J. Functional Programming **11** (2): 155–206, March 2001. Printed in the United Kingdom

Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm

MARGARET BURNETT, JOHN ATWOOD, REBECCA WALPOLE DJANG,
JAMES REICHWEIN
Oregon State University, OR, USA

HERKIMER GOTTFRIED
Hewlett-Packard

SHERRY YANG
Oregon Institute of Technology, OR, USA

Abstract

Although detractors of functional programming sometimes claim that functional programming is too difficult or counter-intuitive for most programmers to understand and use, evidence to the contrary can be found by looking at the popularity of spreadsheets. The spreadsheet paradigm, a first-order subset of the functional programming paradigm, has found wide acceptance among both programmers and end users. Still, there are many limitations with most spreadsheet systems. In this paper, we discuss language features that eliminate several of these limitations without deviating from the first-order, declarative evaluation model. The language used to illustrate these features is a research language called Forms/3. Using Forms/3, we show that procedural abstraction, data abstraction and graphics output can be supported in the spreadsheet paradigm. We show that, with the addition of a simple model of time, animated output and GUI I/O also become viable. To demonstrate generality, we also present an animated Turing machine simulator programmed using these features. Throughout the paper, we combine our discussion of the programming language characteristics with how the language features prototyped in Forms/3 relate to what is known about human effectiveness in programming.

Capsule Review

What is the most widely-deployed use of functional programming? Spreadsheets, of course! Yet the functional programming community pays surprisingly little attention to spreadsheets. This paper is an exception to that rule.

Considered as programming languages, spreadsheets are pretty limited. Forms/3 is an exploration of how far these limitations can be overcome without losing the immediacy and responsiveness that characterises the spreadsheet paradigm. The paper draws insights from both the functional-programming and visual-languages communities; it is an intelligent and articulate discussion of this cross-disciplinary territory.

1 Introduction

A criticism that some have leveled against functional languages is the assertion that functional languages are difficult for many programmers to use. Yet, spreadsheet systems provide evidence to the contrary: even though spreadsheet systems are (first-order) functional programming languages, the success of spreadsheet systems in the commercial market has shown that they are simple enough for a huge number of end users to use. However, while spreadsheet systems are indeed programming languages – they feature at least some degree of composition (through the inclusion within a cell's formula of references to other cells), selection (through a functional if-then-else), and a limited facility for repetition (through replication of the same formula across many rows or columns) – they have historically been rather limited. One limitation has been that spreadsheet systems usually support only a few types, typically numbers, strings, and Booleans. Another limitation has been the lack of abstraction capabilities, which has prevented the kind of expressive power that comes from procedural abstraction, data abstraction, and exception handling. Despite these limitations, however, if the number of people using a programming paradigm is a measure of its popularity, then the spreadsheet paradigm is probably the most popular programming paradigm in use today.

Henceforth, we use the term *spreadsheet languages*[1] to refer to all systems that follow the spreadsheet paradigm, in which computations are defined by cells and their formulas. The essence of the spreadsheet paradigm is expressed well by Alan Kay's *value rule*, which states that a cell's value is defined solely by the formula explicitly given it by the user (Kay, 1984). The value rule disallows devices such as multi-way constraints, state modification, or other non-applicative mechanisms that have sometimes been used to extend spreadsheet languages. When we say a language feature is *consistent* with the spreadsheet paradigm, we mean that it upholds Kay's value rule.

Via the research language Forms/3 (Burnett and Ambler, 1994; Burnett and Gottfried, 1998), a lazy spreadsheet language, we have been experimenting with both programming language and Human-Computer Interaction (HCI) devices to remove spreadsheet limitations without sacrificing consistency with the spreadsheet paradigm. Although we use Forms/3 as a testbed for some techniques intended for spreadsheet languages aimed at end users, Forms/3 also contains techniques intended for trained programmers. In essence, Forms/3 is a 'gentle slope' language, 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 and programmers to create more powerful spreadsheets without having to leave the spreadsheet paradigm to do so.

[1] We have chosen this terminology to emphasize the fact that even commercial spreadsheet systems are indeed languages for programming, although they differ in audience, application, and environment from traditional programming languages. Strictly speaking, a 'spreadsheet system' includes both a spreadsheet language and environmental features. However, in this paper we will not usually differentiate between features present in the language versus the environment, since unlike traditional languages, spreadsheet language features are designed to support tight integration with a particular environment.

1.1 Differences between spreadsheet languages and other functional programming languages

The similarities between spreadsheet languages and more traditional functional languages are obvious: like other functional languages, spreadsheets are applicative, and hence computations are specified by providing arguments to functions and/or operators.[2] Further, like other functional languages, the evaluation mechanisms for spreadsheet languages are declarative, and can be eager, lazy, or a mixture of both. Not surprisingly, given these attributes in common, some spreadsheet languages have been implemented as applications of lazy functional programming (e.g. Wray and Fairbairn, 1989).

There is also an obvious difference between spreadsheet languages and other functional languages: unlike spreadsheet languages, most functional languages support higher-order functions. It is not impossible for a spreadsheet language to do so (e.g. see de Hoon *et al.*, 1995), but since this is not commonly associated with spreadsheets, for the purposes of this paper we will regard only first-order functions as a characteristic of the paradigm.

Another difference between spreadsheet languages and other functional languages is the presence of continuous evaluation in spreadsheet languages, which ensures that all values on the screen are correct reflections of the current formulas in the cells. This difference may, on the surface, appear to be an environmental nicety, but it has more fundamental effects. The continuous evaluator can be described as a simple constraint solver that handles the one-way, equality constraints described by the spreadsheet's formulas. This constraint solver is necessary to provide the immediate feedback (automatic recalculation) feature that is present in spreadsheet languages, but it also enables the use of one-way constraints (expressed as spreadsheet formulas) to support time-based calculations, such as animations and GUI I/O, as we demonstrate later in this paper. Due to the presence of the constraint solver, in some of the literature, spreadsheet languages are said to follow the *one-way constraint* paradigm.

These two differences as well as other language design differences have been due to the fact that the primary intended audience for this paradigm has been end users with no formal training in programming. Examples of such differences in addition to those discussed above include the lack of procedural or data abstraction features, and the use of a very simple model of I/O, consisting only of the ability to enter constant formulas (the only 'input' capability) and to receive immediate feedback (the only 'output' capability).

1.2 Forms/3 design goals

We have already mentioned that our overall goal has been to remove limitations previously associated with spreadsheet languages while still remaining consistent with the spreadsheet paradigm. The motivation behind this goal has been twofold: first,

[2] We will not differentiate between functions and operators in this paper.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

to bring support for more powerful programming capabilities to end users (people who are comfortable with computers but are not formally trained in programming), and second, to leverage some of the ease of programming achieved by spreadsheet languages to professional programming as well.

Any language feature we added could have undermined attributes critical to these ease of programming goals. This has been a consistent problem in the history of programming language design: increasing power has often lead to a corresponding decrease in the language's usability by its intended audience, and therefore its usefulness. Our view is that programming language design is in part a Human-Computer Interaction (HCI) problem. Thus, our design goals include drawing upon what is known about how programming language design attributes affect people's ability to use a language effectively. Background from the HCI literature about the relationship of our particular design goals to research about human productivity in programming and problem-solving is summarized in Appendix A.

Two HCI-related design goals have had a particularly strong influence on Forms/3: directness and immediate visual feedback. In this paper we will use the term *directness* to mean following the principles advocated by Shneiderman; by Hutchins, Hollan and Norman; by Green and Petre; and by Nardi. In short, directness means employing a vocabulary directly related to the task at hand; see Appendix A for more details. For example, for programming graphics, the ability to directly draw the desired graphics instead of textually describing the desired graphics would be an example of directness. Directness is one of the language design goals of Forms/3.

In the context of programming, *immediate visual feedback* refers to automatic display of semantic effects of program edits, and HCI researchers have revealed important ways it can improve programmers' effectiveness. Immediate visual feedback is supported in spreadsheet languages via the continuous evaluator. Tanimoto has coined the term *liveness* to categorize the immediacy of semantic feedback that is automatically provided during the process of editing a program (Tanimoto, 1990). Tanimoto described four levels of liveness. At level 1 no semantics are communicated to the computer by the user's edits, and hence no semantic feedback about the edits is ever provided to the user. An example of level 1 is an entity-relationship diagram for documentation. At level 2 the user can obtain semantic feedback about a portion of a program after an edit, but it is not provided automatically. Some compilers support level 2 liveness only for final output values; other compilers and most interpreters do so for a wide range of program attributes. At level 3, incremental semantic feedback is automatically provided whenever the user performs an incremental program edit, and all affected on-screen values are automatically redisplayed. This ensures the consistency of display state and system state if the only trigger for system state changes is user editing. The automatic recalculation feature of spreadsheet languages supports level 3 liveness. At level 4, the system responds to program edits as in level 3, and to other events as well such as system clock ticks and mouse clicks over time, ensuring that all data on display accurately reflects the current state of the system as computations continue to evolve. Forms/3 is an example of a spreadsheet language that supports time-related computations

and provides feedback about them at liveness level 4. In this paper, the terms *live* and *liveness* refer to liveness level 3 or higher.

Immediate visual feedback is facilitated when concrete objects are present in the programming environment, because in that case feedback about semantics can be concretely based upon those specific objects. Because immediate visual feedback is emphasized in Forms/3, concreteness is a goal as well.

1.3 Organization of this paper

In this paper, we use Forms/3 to show that the limitations previously associated with the spreadsheet paradigm are not inherent, and how they can be removed without loss of consistency with the spreadsheet paradigm. A parallel thread throughout this paper is how the design goals of section 1.2 are realized in Forms/3.

We begin in section 2 with two basic ways Forms/3 extends traditional spreadsheet languages: graphical types and dynamic grids. The way the mechanisms supporting these features are related to the design goal of directness is also discussed. Section 3 generalizes upon linked spreadsheets to support procedural abstraction and data abstraction. An emphasis in section 3 is on how abstraction capabilities can be supported without sacrificing concreteness. Section 4 introduces time-oriented calculations, GUI I/O and animation, and shows how these features can be leveraged for program comprehension and debugging purposes. Immediate visual feedback, particularly when coupled with concreteness, is a key that makes possible many of the features presented in that section. Section 5 relates our work to other spreadsheet languages and visual languages. Following sections 6–8, which present future work, implementation status, and conclusions, Appendix A presents a discussion of relevant HCI research, and Appendix B demonstrates a Turing machine implemented using dynamic grids and time.

2 Basic features of Forms/3

Definitions for the elements of the Forms/3 language are given in Table 1. As the definitions imply, Forms/3 programs (Definition 1) are forms (spreadsheets) containing cells. A form is a flexible organizational unit, analogous to what might be described as a subprogram or a module in some traditional languages. An example of a form (Definition 2) that is also a type definition form (Definition 3) is primitiveCircle in figure 1.

Unlike in traditional spreadsheet languages, Forms/3 cells need not be elements of grids (matrices). A Forms/3 user can place the individual cells (Definition 5) in the form's cellSet (Definition 4) anywhere on the form. This allows flexibility in achieving visual results and documentation simply by placement of the cells. Figure 1's radius, thickness and lineStyle are examples of simple cells (Definition 7) that are not in any grid. A simple cell is analogous to a first-order zero-arity function (a function with no formal parameters, thus referring only to free variables). Forms and simple cells are the basic language elements; discussion of the remaining elements will be deferred until sections 2.1–2.2.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 1. *Language elements of Forms/3. Formulas are as defined in Table 2*

Defn 1.	A *program* is a set of forms.
Defn 2.	A *form* in a program P is the tuple (ID, modelName, cellSet), where ID uniquely identifies the form within P, and modelName $= \begin{cases} F.\text{modelName} & \text{if this form is a copy of form } F \\ \text{ID} & \text{otherwise.} \end{cases}$
Defn 3.	A *type definition form* is a form whose cellSet includes a simple cell with ID Image, one abstraction box with ID MainAbs, and zero or more additional cells.
Defn 4.	A *cellSet* is a set of cells.
Defn 5.	A *cell* is a simple cell or a cell group.
Defn 6.	A *cell group* is a dynamic matrix or an abstraction box.
Defn 7.	A *simple cell* on a form F is the tuple (ID, formula, value, visual attributes), where ID uniquely identifies the simple cell within F.
Defn 8.	A *dynamic matrix* on a form F is the tuple (ID, cellSet, formula, value, visual attributes) whose cellSet contains only simple cells, including one whose ID is *MID* [NumRows] and one whose ID is *MID* [NumCols], where *MID* is the dynamic matrix's ID and uniquely identifies the dynamic matrix within F.
Defn 9.	An *abstraction box* on type definition form F is the tuple (ID, cellSet, formula, value, visual attributes) whose cellSet contains only simple cells and dynamic matrices, and that is an element of a type definition form's cellSet, where ID uniquely identifies the abstraction box within F.

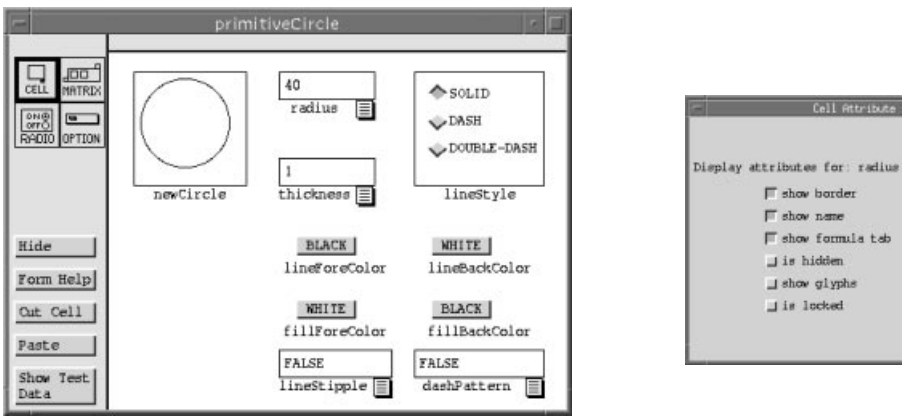


Fig. 1. (Left) a portion of a Forms/3 form (spreadsheet) that defines a primitiveCircle. The primitiveCircle in cell newCircle is specified by the other cells, which define its characteristics. A user can view and specify formulas by clicking on the formula tabs attached to the bottom right of each cell. Radio buttons and popup menus are equivalent to cells with constant formulas. (Right) visual attributes of cell radius.

Each cell has a formula as well as some visual attributes controlling its appearance, and the program's outputs are entirely determined by the combination of these formulas and attributes. A cell's value is the result of the execution of the formula. The value is well defined prior to computation (since it is simply the result of the

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 2. *The grammar for Forms/3 formulas.* (*Note that subexpressions are fully parenthesized, thereby avoiding ambiguity.*) *As the top section shows, it has the usual spreadsheet formula operators and also some operators supporting computations on grids* (*dynamic matrices*) *and on graphics. The bottom section shows cell reference syntax, which includes row/column referencing for cells that are in a grid* (*Matrix*)

formula	::=	*Blank* | expr
expr	::=	*Constant* | ref | infixExpr | prefixExpr | ifExpr | composeExpr | (expr)
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	unaryPrefixOperator subExpr | binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr | IF subExpr THEN subExpr
composeExpr	::=	COMPOSE subExpr AT (subexpr subexpr) composeWithClause | COMPOSE subExpr AT (subexpr subexpr)
composeWithClause	::=	WITH subexpr AT (subexpr subexpr) composeWithClause | WITH subexpr AT (subexpr subexpr)
subExpr	::=	*Constant* | ref | (expr)
infixOperator	::=	+ | − | * |/|AND|OR| = | > | < |...
unaryPrefixOperator	::=	- |ROUND|ABS|WIDTH|HEIGHT|ERROR?|...
binaryPrefixOperator	::=	APPEND | MATRIXSEARCHROW WHERE |...
ref	::=	cellRef | *Form.ID*: cellRef
cellRef	::=	*SimpleCell.ID* | *Matrix.ID* | *Matrix.ID* [subscripts] | *Abs.ID* | *Abs.ID* [*SimpleCell.ID*] | *Abs.ID* [*Matrix.ID*] | *Abs.ID* [*Matrix.ID*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

formula), but Forms/3 is a lazy language, and hence each value is actually computed only as needed, and may be saved or discarded according to any arbitrary caching strategy.

Some spreadsheet languages allow a cell's visual attributes to be defined, like values, via formulas. However, this is not necessary in Forms/3, because cell values themselves can be highly graphical; hence cell attributes are defined solely via constants. Cell attributes relate to a cell's appearance and availability for user editing.[3]

The name attribute raises the issue of scope. In this paper, most cells have been given names, because this contributes to readability of the formulas. However, in the absence of a name, a cell can still be referenced (by clicking on it); such a reference is then reflected textually in a formula via the system-generated ID. The scope of cells'

[3] The attributes are: cell position and cell size, specified by directly manipulating the cell's position and size, an optional cell name specified by typing the name under the cell, optional dataflow arrows' visibility toggled by clicking on the cell, and the attributes on the pop-up checklist at the right side of figure 1.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

names and IDs is local to the form unless qualified by the form's ID; if qualified by the form's ID, they are accessible globally, in the spreadsheet tradition, unless the visibility/information hiding mechanism discussed in the next section is employed.

The textual syntax of formulas is given in Table 2; some formulas can alternatively be entered using a graphical syntax, as will be seen in the next subsection. Most of the operators are straightforward, but a few require some explanation.

A formula of *Blank* results in 'no value'. In some spreadsheet languages, 'no values' are treated as being absent, so that additions, etc., can continue without type errors. In Forms/3, however, 'no value' is actually a value of type noValue, with the advantage (in our opinion) of raising type errors if inappropriate operations are performed on it.

In a functional setting, the else-less 'If subExpr Then subExpr' syntax is unusual. For now, we will slightly oversimplify and say that an else-less if is the equivalent of the syntax 'If subExpr Then subExpr Else *Blank*'. This simplification will be revisited when we introduce the Forms/3 model of time.

There are four 'pseudo references' not shown – I, J, LASTROW and LASTCOL – that can be used in grid formulas. Including these in the grammar is straightforward but tedious, and we have omitted them for brevity. I and J are ways for a cell in a grid to refer to its own row number and column number respectively, as will be seen in section 2.2.

2.1 Graphics as first-class types

Spreadsheet languages have not traditionally supported graphics, except as certain kinds of output (namely, charts and graphs) and as non-semantic documentation devices. Support for more sophisticated uses of graphics has been provided primarily through macros or trapdoors to other languages. However, as this section demonstrates, there is no inherent limitation in the spreadsheet paradigm that requires such measures.

2.1.1 A simple programming example of graphical types

Forms/3 supports both built-in graphical types[4] and user-defined graphical types as follows. Types are defined on type definition forms. The type is defined by formulas in cells on type definition forms, and an instance of a type is the value of an ordinary cell that can be referenced just like any other cell. Built-in types are provided in the language implementation but are otherwise identical to user-defined types. For example, the built-in circle object shown in figure 1 is defined by cells defining its radius, line thickness, color, etc.

Suppose a spreadsheet user such as a population analyst would like to define a visual representation of data using domain-specific visualization rules that make use

[4] Forms/3's current implementation uses dynamic typing, and that is the version underlying discussions of types in this paper. Dynamic typing is used by almost all spreadsheet languages. We also have work in progress on an implicit static type system.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

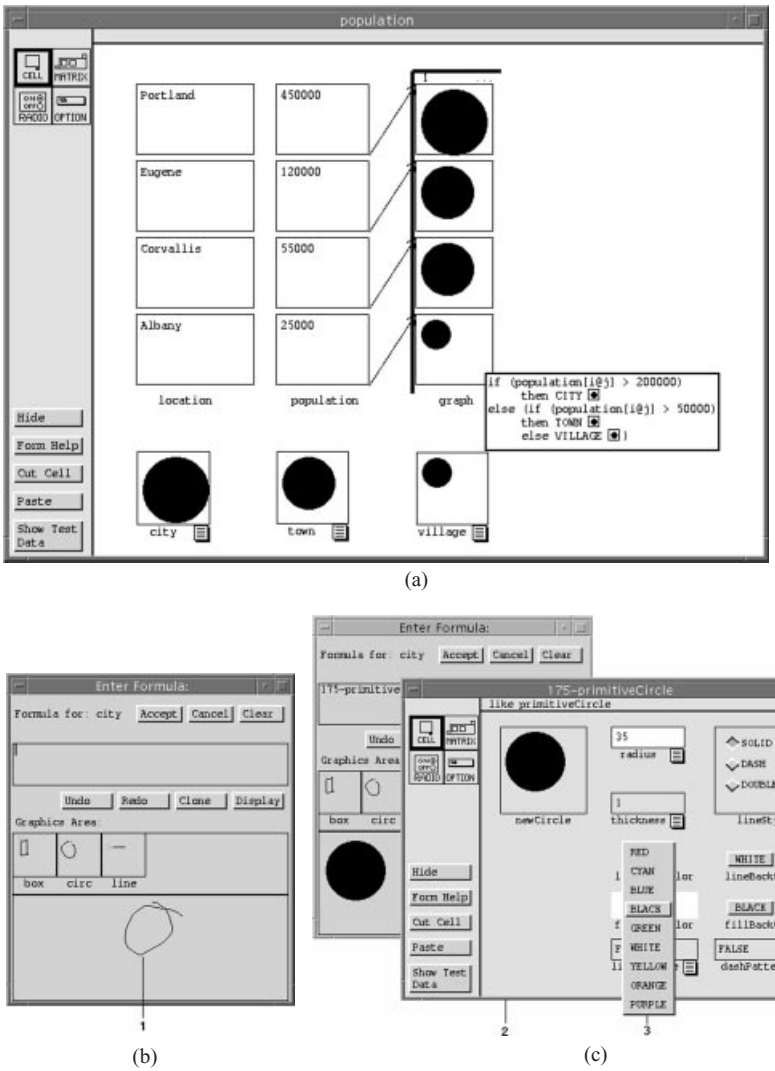


Fig. 2. (a) A spreadsheet under development to visualize population data. The formula shown is shared by the 4×1 dynamic matrix labeled graph. (The •s in the formula are miniaturized drawings of the cells' current values, which can optionally be displayed in formulas.) The optional arrows show how the cells in graph depend on population. (b) To define the circle for cell city, the population analyst first draws a circle gesture (**1**) in city's formula edit window, and then, (c) after clicking on the resulting circle to display its definition form (**2**) (in gray because it is a copy; white indicates formulas different from the original), the population analyst specifies the fillForeColor formula via a popup menu (**3**). Each manipulation is immediately reflected textually and graphically in city's formula edit window (the left window in (c)).

of the built-in primitiveCircle type of figure 1. Figure 2(a) 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.

One valid syntax for the formulas in this example is the conventional textual

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

formula syntax of Table 2. To use this syntax, the population analyst would make a copy of the form shown in figure 1 and edit formulas on the copy as needed. However, recall our design goal of directness. Defining a circle using a '•' would be more direct (i.e. closer to the task to be accomplished) than defining it using integers and math. Thus, Forms/3 includes a graphical syntax for defining such formulas, which includes sketching and direct manipulation. We term this alternative syntax *graphical definitions*, to emphasize that it is a graphical way of defining formulas.

In the example of figure 2, the population analyst defines the formulas for cells city, town, and village by entering circle-shaped gestures in the formula window for each, resizing as necessary to fine-tune the sizes. For example, to define the large city circle, the population analyst first draws a circle gesture as in figure 2(b). This defines the cell's formula to be a reference to cell newCircle on a copy of the built-in primitiveCircle definition form whose radius formula is defined to be the radius of the drawn circle gesture. However, the analyst wants the circle to be solid black. There are no gestures provided to specify fill color, because no obviously appropriate gesture seems to exist for that characteristic of circles. In such cases, the population analyst clicks on the circle to display its definition form, and then enters whatever additional formulas are needed, in this example for cell fillForeColor as in figure 2(c).

There is an apparent similarity between some commercial programming environments' 'property sheets', which allow maintenance of properties of visual objects, and the spreadsheet in figure 2(c), but this similarity does not go beyond the surface. The essential difference is that the cells in figure 2(c) can have arbitrarily complex formulas that specify relationships, not just values as in property sheets.[5] Thus, there is a gentle migration path from the simple formulas that can be specified by an end user via sketching and constant-valued formulas to the more complex formulas that sophisticated programmers might want to use.

Referring again to figure 2(b), alternative graphical syntaxes are to click on the circle icon, which produces a 'representative' value (here, a circle with radius 25), which can then be resized via direct manipulation, or to refer to an existing circle and then manipulate it to demonstrate how it differs from the existing one.

All three graphical ways of specifying the circles are syntactic sugar for the more conventional way of entering formulas textually. However, they feature greater directness by allowing the population analyst to define the desired graphics using a syntax of graphics. An empirical study showed that use of this syntax was linked with both significantly greater programming speed and significantly greater programming accuracy than was use of the equivalent textual syntax (Gottfried and Burnett, 1997).

[5] If a circle depends on a cell whose value is time-varying, such as a reference in the radius cell's formula to the built-in cell containing the system clock, the result will be an animated circle. (This principle also underlies the animated graphics of Fran (Elliott and Hudak, 1997), an add-on to Haskell (Hudak *et al.*, 1992).) We will return to a discussion of animations and other time-varying values later in this paper.

2.1.2 The model for (graphical) types

The above example shows how graphical types work in the case of circles. In fact, in Forms/3, *all* types are considered to be graphical: in addition to the usual attributes of types, all have appearances and optional interactive behaviors. In keeping with this philosophy, in Forms/3 a type is the 4-tuple: (components, operations, graphical representations, interactive behaviors). As the definitions in Table 1 suggest, a type τ is defined via a type definition form F_τ. The form contains at least two cells: an abstraction box with ID MainAbs, which is a cell group that defines the structure of the type as the composition of cells placed inside it (the first element of the 4-tuple); and an *image cell* whose ID is Image and whose formula defines the type's appearance(s) (the third element of the 4-tuple). The other two elements of the 4-tuple, operations and interactive behaviors for type τ, are specified by additional cells on F_τ. All cells inside abstraction boxes are hidden (private), and it is possible to explicitly hide other cells as well.

Note that, in this model, there is no theoretical distinction between built-in and user-defined types. Both are theoretically defined by the above 4-tuples, and practically defined by their accompanying type definition forms. The only distinction is implementation; that is, whether the type's definition form has already been provided by the language implementer.

F_τ's distinguished abstraction box defines as its value a representative instance of type τ, and each additional instance τ_i of τ is defined by the distinguished abstraction box on a copy of F_τ, denoted F_{τ_i}, upon which formulas different from those on F_τ can be defined to allow individual differences among instances of type τ. Instances of type τ can be referred to by any cell but, except for cells on copies of F_τ, can only be operated upon in more substantive ways via the non-hidden cells (public operations) that have been defined on F_τ.

Form primitiveCircle[6] in figure 1 is one example of a type definition form F_τ, where τ is primitiveCircle. Because circles are a built-in type, primitiveCircle is provided in the language implementation. The abstraction box is newCircle, and the image cell is hidden because it is not useful to the user – its formula consists of non-editable low-level code that draws a circle with the characteristics specified by the other cells and formulas on the form. If the user copies $F_{primitiveCircle}$ and changes some formulas on the resulting form $F_{primitiveCircle_1}$'s cells, a different instance of a circle is defined in $F_{primitiveCircle_1}$'s abstraction box newCircle. 175-primitiveCircle in figure 2(c) is an example of $F_{primitiveCircle_1}$.

The mapping from gestures and icon clicks to textual spreadsheet formulas defined using this model is given in Table 3. The mapping from direct manipulation of an existing graphical object to textual formulas is given in Table 4.

[6] In the previous section, we took some liberties with notation for the purpose of brevity. For example, 'Form primitiveCircle' really means 'the form whose ID is primitiveCircle'. In general, we take advantage of the notation of Table 2 as follows. Unless specifically used in the context of a formula syntax example, we will use Table 2's 'ref' syntax as an abbreviation for the forms and cells themselves. For example, 'form F' will be used as an abbreviation for 'the form whose ID is F', and 'F:A' will be used as an abbreviation for 'the cell whose ID is A that is an element of the form whose ID is F'.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 3. *The mapping from gestures and icon clicks to formulas for built-in types. In each case, the result of the gesture is the formula that is a reference to an abstraction box* χ *on a definition form copy* F_{τ_β}*, where* $F_{\tau_\beta} = F_\tau$(DefSet)*, and* DefSet *is the set of formula definitions for each cell defined differently on form* F_{τ_β} *than on* F_τ*. The notation for each element of* DefSet *is* (X.*formula* $= \phi$)*, denoting that cell* X *has the formula* ϕ

Graphical type	Action	Textual formula
primitiveCircle	draw circle of radius ρ	primitiveCircle(radius.formula $= \rho$): newCircle
	click on circle icon	primitiveCircle (radius.formula $= 25$): newCircle
primitiveBox	draw box of width ω and height η	primitiveBox(width.formula $= \omega$, height.formula $= \eta$): newBox
	click on box icon	primitiveBox (width.formula $= 50$, height.formula $= 50$): newBox
primitiveLine	draw line with dx ξ and dy ψ	primitiveLine (deltax.formula $= \xi$, deltay.formula $= \psi$): newLine
	click on line icon	primitiveLine (deltax.formula $= 50$, deltay.formula $= 50$): newLine

Table 4. *The mapping from direct manipulation of an object* α *to formulas for built-in types. As in Table 3, the result of the gesture is the formula that is a reference to an abstraction box on a definition form copy* F_{τ_β}*, where* $F_{\tau_\beta} = F_{\tau_\alpha}$(DefSet)*, and* DefSet *is the set of formula definitions for each cell defined differently on form* F_{τ_β} *than on* F_{τ_α}.

Graphical type	Action	Textual formula
primitiveCircle	stretch edge of circle α to radius ρ	primitiveCircle$_\alpha$ (radius.formula $= \rho$): newCircle
primitiveBox	stretch corner of box α to width ω and height η	primitiveBox$_\alpha$ (width.formula $= \omega$, height.formula $= \eta$): newBox
primitiveLine	stretch line α's endpoint to position (ξ, ψ)	primitiveLine$_\alpha$ (deltax.formula $= \xi$, deltay.formula $= \psi$): newLine

In defining a mapping from direct manipulation of concrete values to general formulas, there are three issues to be addressed: the basic strategy of such a mapping, how to generalize direct manipulations into parameters that are more complex than simple constants, and how to support direct manipulations on types that are not built-ins. This section has discussed the basics of the approach, which

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

demonstrates only the first of these three issues; in section 3, the remaining two of these issues will be covered.

2.2 Dynamically-sized grids

As figure 2 shows, Forms/3 is not tied to the use of a grid – individual cells can be placed in any location, and no grid is required. However, as the example also shows, it is possible to include one or more grids on a form: location, population and graph are all grids. In Forms/3 these grids are dynamically-sized matrices.

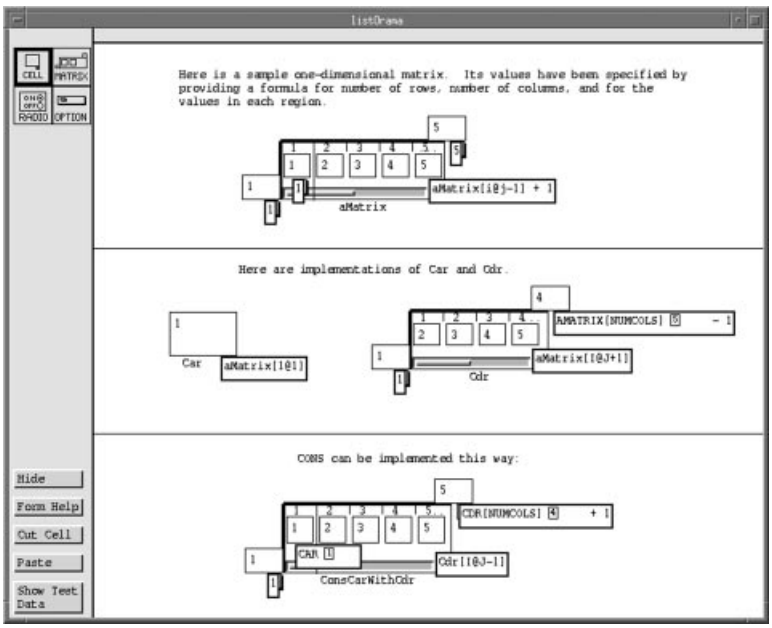
Forms/3 dynamic grids are similar to traditional matrices and to traditional spreadsheet grids in that they are two-dimensional groups of cells that can be referred to in terms of their relative or absolute position. However, they are different from traditional matrices in that they do not have a statically-determined contiguous internal layout; instead, they are created dynamically and lazily. More to the point from the spreadsheet user's perspective, they are different from traditional spreadsheet grids in these ways:

(1) The number of rows and columns in a dynamic grid is determined dynamically by the formulas of its distinguished NumRows and NumCols cells.
(2) The size of a dynamic grid can be queried dynamically through references in other formulas to the dynamic grid's distinguished NumRows and NumCols cells.
(3) Formulas can be specified for a contiguous *region* of the dynamic grid (which contains zero or more cells), and this formula is shared by all the cells in the region.
(4) Alternatively to item (3), a formula can be specified for the entire dynamic grid.

How this combination of features affects spreadsheet programming warrants some discussion. The first two features allow grid size to be both determined by and referred to by formulas. The third feature replaces the traditional 'replicate' mechanism common in commercial spreadsheet languages. The fourth feature is simply an alternative to the third feature, useful for explicitly expressing relationships at the granularity of entire grids. Advantages of the third (and fourth) feature directly visible to the user are that it makes explicit the relatedness of cells with essentially the same formula, and that it removes the maintenance problem of replicating formulas (i.e. duplicating code). In combination with the first feature, it allows cells in a large dynamic grid to be created lazily – since the regions determine the formulas, enough information is present in the regions to dynamically create a cell in a region only if and when it is actually needed. This advantage in turn allows the size of dynamic grids to be time-varying, which will be discussed further in section 4.

Because of these features, Forms/3's dynamic grids have the same functionality as the lists commonly found in functional languages. Figure 3(a) shows an implementation of the basic list operations to demonstrate this. Forms/3 supports recursion, as will be demonstrated in section 3, and these basic list operations can be combined in the usual way with recursion to write more elaborate list operations. However, for many list operations, recursion is not actually necessary, as is demonstrated by

(a)



(b)

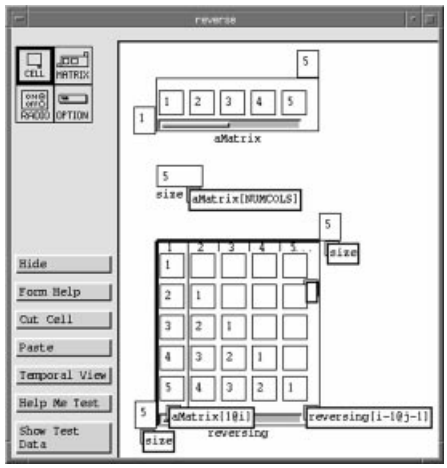


Fig. 3. (a) Forms/3's dynamic grids can implement the basic list operations. The sample input, aMatrix (top portion of the window), has been defined to have 1 row, 5 columns, and to consist of two regions, which the user established by dragging the vertical bar from the left border rightwards. The first region consists of the first cell, and the second consists of all remaining cells. i and j in the second region's formula are pseudo-references that refer to a cell's own row number and column number. If cell aMatrix[NumCols] (attached to the upper right of aMatrix) is given a different formula such as "3 * 2", then all the dependent cells will (lazily) adjust themselves appropriately. (One formula in Cdr and two formulas in ConsCarWithCdr have been displayed with the miniaturized drawings of the referenced cells' current values showing, such as the 5 for the reference to aMatrix[NumCols].) (b) The 2-dimensional grid reversing reverses the single-row grid aMatrix. Reversing consists of three regions – the left column, the top row except for the cell in the left column, and the rest – and thus three formulas define the calculations of its interior cells. (One of these formulas, the one for the top row, is blank.) In addition, the number of rows and number of columns each has a formula. The answer is in the bottom row.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

figure 3(b). In fact, dynamic grids in combination with time-varying operations can be used to program a Turing machine simulator, without using either recursion or traditional forms of iteration (see Appendix B). We chose dynamic grids over lists because dynamic grids are based on traditional spreadsheet grids, thus allowing end users familiar with spreadsheet grids a gentle slope to the advanced functionality supported by dynamic grids.

Regions themselves have some similarity to the list comprehensions (Wadler, 1987) used in some functional languages, but regions are less powerful than list comprehensions. List comprehensions consist of a generator and a filter. They produce a list containing all the elements from the generator that satisfy the predicates in the filter. Hence, the resulting list can be smaller than the input list from the generator. Regions do not share this attribute; they include generator functionality, but no filter – they always specify *n* output cells from *n* input cells, because a region formula is simply a specification for the value of every cell in the region. The region's size itself is specified through mechanisms external to the region, namely by the user's manipulations of region boundary lines to establish the static position and size of each region within a dynamic grid G, and by the evaluations of the formulas of G[NumRows] and G[NumCols]. Hence, it is possible to achieve list comprehension functionality by combining region formulas with recursion and dynamic grid sizing, but it is not possible to do so using region formulas alone.

The three dynamic grids in the population example in figure 2 were set up as follows: location is a dynamic grid with four rows and one column, with each cell inside the grid having its own formula (such as 'Portland'); population is a similar grid, but with population[NumRows]'s and population[NumCols]'s formulas set up as references to location's corresponding cells; and graph is a dynamic grid with the same number of rows and columns as population, and with the four interior cells sharing the single region formula shown.

The pseudo-references i and j in the figures provide a general way for a cell to refer to its own row and column; in object-oriented languages with a *self* pseudo-variable, such a reference might be expressed as self.i. and self.j. Like *self*, these are placeholder references that are general enough to allow a single region formula to be applicable to all cells in that region. This is in contrast to zero-argument functions, because if they were zero-argument functions, referential transparency would be lost.

Our approach to dynamic matrices has several features that are similar to an approach proposed for Forms/3 by Viehstaedt and Ambler (1992). The Viehstaedt/Ambler version is more powerful, allowing region sizes to be specified via formula, and allowing multiple views as to how a single dynamic matrix is divided into regions. We did not include these two capabilities in Forms/3 because we thought they might detract from the understandability of dynamic matrix programs. The static representation of dynamic matrix formulas is also different in the Viehstaedt/Ambler version. The Viehstaedt/Ambler approach to dynamic matrices has since been developed further (Wang and Ambler, 1996) in the context of the spreadsheet language Formulate (Ambler and Broman, 1998; Ambler, 1999).

When the approach to dynamic matrices was still in the design stage, we conducted an empirical study comparing construction of matrix manipulation programs in

Forms/3 (using a variation on the Viehstaedt/Ambler static representation) with the same task in two textual programming languages (Pandey and Burnett, 1993). The study was done using only pencil and paper. Its goal was to determine whether the approach to dynamic matrices of Forms/3 would be used by the subjects more accurately than when using more traditional approaches to writing matrix manipulation programs. To do this, we compared 60 subjects' correctness in writing matrix manipulation programs in Forms/3 with their ability to write the same programs in two textual languages. One of these languages was Pascal, because it was representative of the most widely-used paradigm (imperative) and because it was the language best known by the subjects (CS juniors) at the time the study was done. The other language was a version of APL that had been modified to use an English-like syntax. We chose APL because it was the most matrix-specific textual language available, but we modified the syntax to use common words and symbols and left-to-right reading order to allow the subjects to learn it quickly. Each subject constructed two small matrix manipulation programs in all three languages, for a grand total of six programs by each subject, done in varying order to balance any learning advantage.

In total, significantly more of the programs were constructed correctly in Forms/3 than in the other two languages. This total came from the fact that in one of the two problems, the Forms/3 and Pascal solutions were approximately the same in terms of correctness and were significantly more correct than the APL solution; and in the other problem, the Forms/3 and APL solutions were about the same degree of correctness and significantly more correct than the Pascal solution. The extent to which Forms/3 compared favorably with the other two languages was actually quite remarkable, given that the subjects were already experienced in Pascal, and that APL contains a built-in primitive that entirely solved one of the problems. Our belief is that these results are due to directness and concreteness; that is, that subjects programmed most correctly in Forms/3 because they were able to program using a vocabulary consisting of matrix-oriented operators and concrete, visible matrices and matrix elements, rather than using a vocabulary of loops, subscript arithmetic and variables representing arbitrary matrices.

3 From concrete linked spreadsheets and graphical types to generalized abstractions

3.1 Procedural abstraction and automatic generalization

Applying dynamic grids to the population example yields a certain amount of generality. For example, the original form, which includes Oregon cities, can be copied for use on Nevada cities. Suppose three cities in Nevada are to be included in the analysis. The user changes the formula for location[NumRows] on this copy to '3', enters the city names in the three remaining regions of the grid, enters the corresponding figures in the population dynamic matrix, and the graph automatically comes out correctly. This degree of generality is due to the dynamic sizing capability, and to the use of regions to specify a formula for an entire section of a dynamic matrix (see figure 4).

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

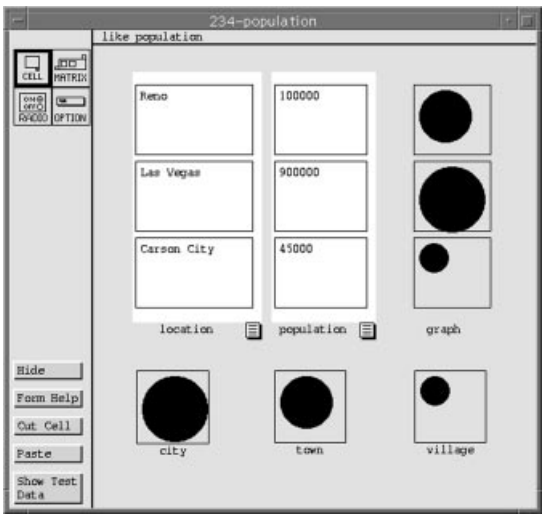


Fig. 4. Copying the Oregon population form applies the graphical depiction to a different state's cities. The term 'copy' does not perfectly describe the relationship with this form and the original: the non-white matrices on the 'copy' share the same formulas as the original; the user edited the white cells to enter Nevada information, and hence their formulas are no longer shared. If a bug is fixed in the original form, the fix will also be propagated to the unedited corresponding (gray) cells on any copies.

As this example demonstrates, a form provides functionality similar to both a (first-order) function and an instance of that function (i.e. an activation record), and cells whose formula tabs have been left visible provide parameter-like functionality. However, the approach does not seem to afford as much generality in expressiveness as is usual with approaches to procedural abstraction in first-order languages. The formulas for the cells in the example all refer to values that the spreadsheet creator explicitly instantiated, either by entering them explicitly via constant formulas, or by referring to cells on forms (analogous to visible activation records) that he or she manually created. In contrast to this, conventional first-order functions' parameters can automatically generate the needed activation records at runtime.

3.1.1 Generalization example: What the user does

Our solution to providing as much generality as is present with conventional first-order functions is through automatically generalizing formulas through deductive reasoning. Suppose that, instead of referencing only the pre-existing forms that were set up 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. To create this program, the analyst copies the primitiveCircle form to create a new copy (say, 250-primitiveCircle), edits cell radius on that copy to be '1 +

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

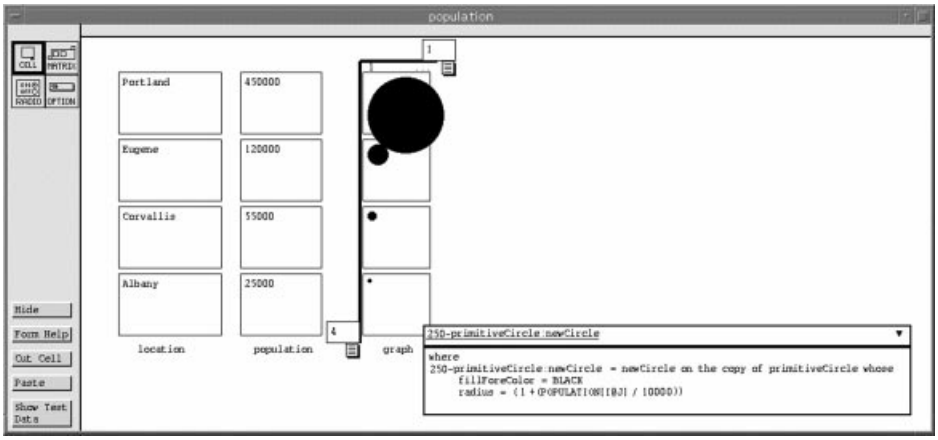


Fig. 5. A more general version of the population program is in progress. The concrete formula (i.e., the way the user programmed it) is shown in the top half of the formula window; it is underlined to indicate that generalization has occurred. To see the results of generalization, the user clicked the arrow at the right of the formula, causing the generalized formula to be shown below the concrete one.

(population:population[i@j]/10000))'[7] and references the resulting circle in graph's region formula. The system immediately responds by displaying a sample result calculated using population[1@1].

The analyst's task is finished, but the system still needs to generalize further. If it did not generalize, all the cells in graph would be the same size, because they would all refer to newCircle on *the same* copy, namely 250-primitiveCircle. After the system generalizes, using the method described next, the formula shown at the bottom right of figure 5 is produced, which says that each reference in graph's formula is to cell newCircle on *an appropriate* copy of primitiveCircle.

3.1.2 The generalization method

Concrete sample values and directly pointing to the objects of interest are strategies common in programming languages that aim to promote directness, especially languages using demonstrational techniques (Cypher, 1993), and these features usually lead to the need for generalization. Forms/3 shares this need because it makes use of concrete programming features, which are central to its ability to provide immediate visual feedback incrementally after every formula edit (i.e. liveness).

An approach to generalization in programming languages can be either explicit or implicit. In an explicit approach, the user would provide the generalized interpretation explicitly, such as by manually typing in the legend shown in figure 5. Implicit approaches, which are common in demonstrational languages, derive the generalized version (such as that shown in the legend) automatically. If an implicit approach

[7] Another alternative would be to have the area be directly proportional to population via a radius formula along the lines of 'ceiling (sqrt (population : population[i@j]/1000))'.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

for generalization employs inference,[8] which is the case in many demonstrational languages, there is a possibility of guessing wrong. The probability of doing so is often reasonable in domain-specific languages, in which the number of possibilities are relatively small, and a number of domain-specific languages have successfully employed this technique. (See Cypher (1993) for several examples.) However, this kind of inference has not proven to be viable in general-purpose languages, because the probability of guessing wrong has been too high.

Fortunately, in spreadsheet formulas, the operators are already fully general; only the operands must be generalized. This makes the implicit generalization problem much easier than in entirely demonstrational languages; in fact, there is enough information to allow generalization to be entirely implicit[9] while still requiring only deductive reasoning, without the need for inference that employs guesswork.

Even the operand is already partly general: the cell part of the operand has been specified in a general way by the user. The only aspect of an operand that actually needs to be deduced is an abstract specification of how to generate an appropriate copy of a form when needed. Let F_α be a form, let F_{α_i} be a copy of F_α instantiated directly by a user pressing the copy button, and let $DefSet_{\alpha_i}$ be a set of elements of format 'X.formula=ϕ', where each X is a cell on F_{α_i} whose formula has been edited to be ϕ. Thus, just as before in Table 3, it is possible to abstractly specify copy F_{α_i} by enumerating how its cell relationships differ from those in F_α:

$$F_{\alpha_i} = F_\alpha(DefSet_{\alpha_i})$$

Given F_α, this description is sufficient for the system to automatically generate copies exactly the same as F_{α_i} at future runtimes. More important, by substituting IDs of different copies and/or different grid element subscripts in $F_\alpha(DefSet_{\alpha_i})$, this description is sufficient to generate additional, similar, copies of F_α such as the additional copies of primitiveCircle needed to support rows 2–4 of grid graph in figure 5.

3.1.3 The granularity issue

The above generalization reasoning is at the granularity of entire forms, and thus suffices for supporting the traditional call-return structure of one function invocation calling another, as found in traditional applicative languages, allowing even recursive programs to be programmed in the above concrete manner and then generalized correctly. If the technique were intended only for programmers, it would not be necessary to improve upon this level of support.

[8] 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 this latter convention.

[9] Commercial spreadsheet languages are partially explicit; the user must enter a special character (often a '$') with a cell reference to make it an 'absolute' reference. The implicit ('relative') references are generalized based solely on spatial relationships in the grid. For spreadsheet languages not tied to a single grid, it is necessary to base generalization of implicit references at least in part on logical relationships.

However, one of the goals of this research is to explore ways to support end user programming of spreadsheets, and it may not be reasonable to expect end users to structure their programs to follow such a traditional call-return structure. The use of multiple forms that reference one another (the same idea as linked spreadsheets) supports not only call-return structures, but in fact any arbitrary non-circular cell referencing pattern. For example, dataflow paths that follow a linear pipeline of spreadsheet cells are possible in linked spreadsheets, as in the relationship among the three N cells in figure 6. Some of these non-traditional structures require reasoning at a finer granularity than entire forms, because when cells in two copies of the same form reference each other, the form referencing pattern appears circular even when the cell referencing pattern is not. In this example, the pipeline of Ns combined with recursive formulas of Ans and tree shows such apparent circularity at the granularity of forms – 137-Fibonacci:N references 127-Fibonacci:N, yet 127-Fibonacci:tree incorporates 137-Fibonacci:tree. Here, reasoning at the granularity of forms would be problematic if the user edited cell Ans in the two copies, because then each copy's *DefSet* would have to be described in terms of the other copy's *DefSet*.

Our solution is to reason only about the portions of a form that actually affect the cell whose formula is currently being generalized. Suppose X is the cell whose formula is currently being generalized. Let $AffectsSet_{\alpha_i}$ be $\{(Y.\text{formula} = \phi) \mid Y \xrightarrow{+} X\}$, where $Y \rightarrow X$ denotes a reference in X's formula to Y (the arrow is drawn in the direction of dataflow), $\xrightarrow{+}$ is the transitive closure of $\rightarrow$, and ϕ is any formula. Using this definition, we modify the description of a generalized version of some concrete reference $F_{\alpha_i}:Y$ in X.formula to be:

$$F_{\alpha_i}: Y = F_{\alpha}(DefSet_{\alpha_i} \cap AffectsSet_{\alpha_i}): Y$$

We will say that the generalized version of a reference in cell X's formula is *correct* if replacing the concrete reference with the generalized reference results in the same value in cell X as with the concrete reference. As we have pointed out before, replacing every reference $F_{\alpha_i}: Y$ with $F_{\alpha}(DefSet_{\alpha_i}): Y$ would have been correct in this sense, since $F_{\alpha}(DefSet_{\alpha_i})$ completely describes F_{α_i} by enumerating every difference between F_{α_i} and F_{α}. Further, a cell not in $AffectsSet_{\alpha_i}$ cannot possibly have any effect on X's value; hence a reference to $F_{\alpha}(DefSet_{\alpha_i} \cap AffectsSet_{\alpha_i}): Y$ must produce the same result in X as a reference to $F_{\alpha}(DefSet_{\alpha_i}): Y$. Since the system enforces that references among cells are non-circular (even though relationships may seem circular when viewed at the granularity of whole forms), this approach adds the generality needed to support even non-traditional cell referencing structures such as in figure 6.

Generalization is performed lazily, i.e. is invoked only when the existing concrete formula will not suffice. The concrete formula will not suffice if the formula has grid references of the sort in figure 5; if, left ungeneralized, calculation of the answers on the displayed version cannot terminate due to seemingly circular references, as in figure 6; or if the concrete information is about to be separated from a portion of the program, as is the case when the spreadsheet creator decides to save only some of the forms (part of the program) to disk.

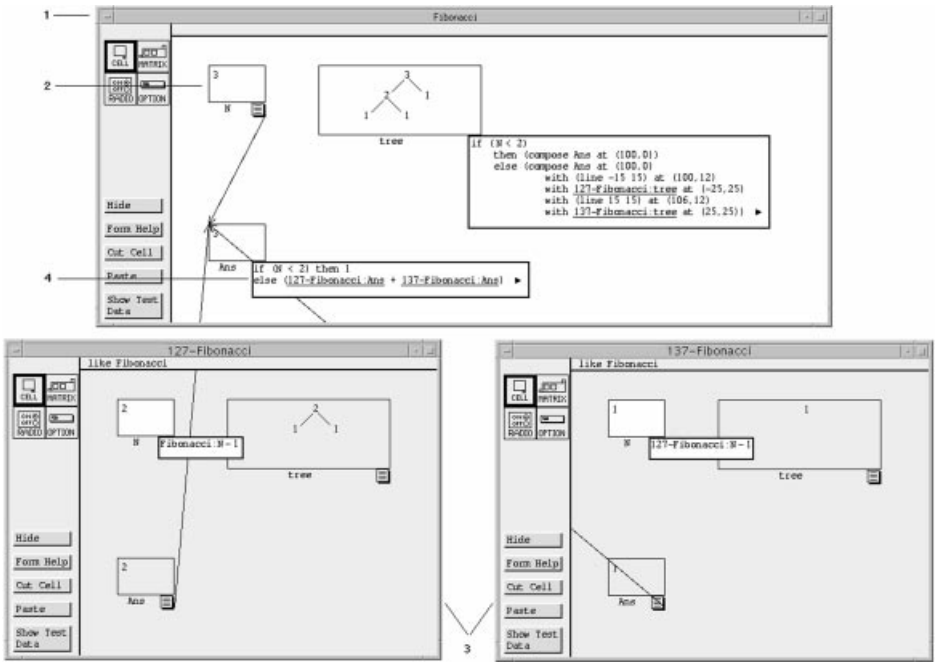


Fig. 6. To program this recursive selection of the Nth number from the Fibonacci sequence, the programmer creates the main form (1) and puts the three cells on it, giving N (2) the formula '3' to provide a sample value. Next the programmer tells the system to copy the form twice, enters the formulas for the N cells on the two copies (3), and then enters the concrete formula for the original cell's Ans (4), which refers to Ans on the copies. The programmer has clicked on Ans to display the optional dataflow arrows depicting Ans's data dependencies. The tree cell sketches the relationships in the Fibonacci sequence. A non-traditional feature of the structure of this program is that it produces two answers, Ans and tree, either or both which can be referenced as needed without the special packaging and de-packaging constructs required in most other types of applicative languages.

Even after generalization, a concrete version of the reference can always be viewed. If no concrete version is already in the system, it is automatically generated upon request. For example, in figure 6, the references to cells on form 127-Fibonacci are concrete, and the user can click on the reference to have the copy (127-Fibonacci in this case) spring into view if it is not already present on the display. If no concrete copy remained in the system, this concrete name would have been changed by the system to an abstract name, such as 'Fibonacci-a'; in that case, if the user clicks on the reference, a concrete example of form Fibonacci-a with the same $DefSet \cap AffectsSet$ as that of form 127-Fibonacci will spring into view.

3.2 Data abstraction in the spreadsheet paradigm

The graphical model of types described in section 2, combined with the information hiding and generalization capabilities, provides the features necessary for a spreadsheet-based approach to data abstraction.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

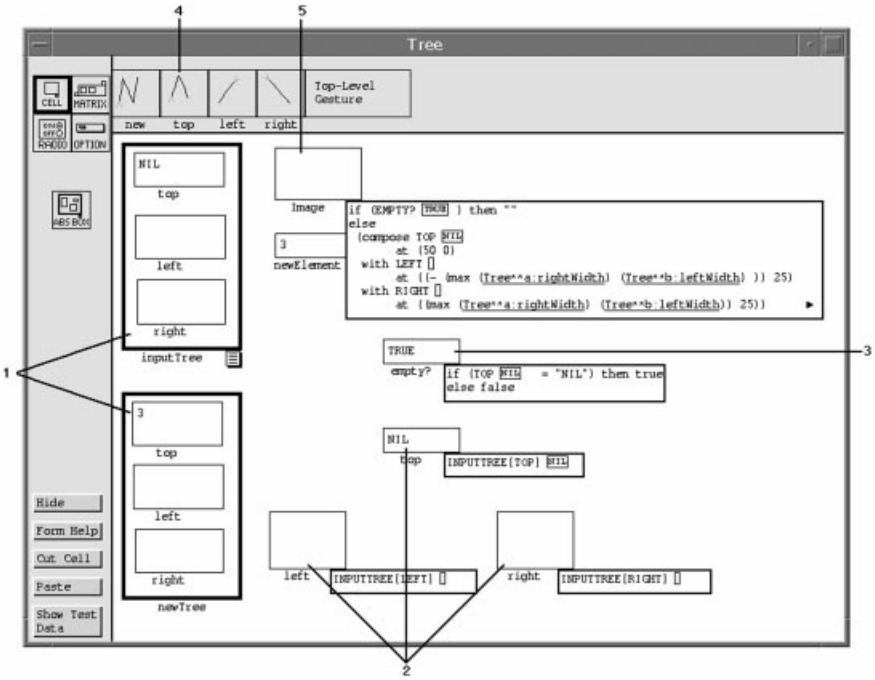


Fig. 7. The Tree definition form's accessible cells (**1**–**3**) and gestures (**4**), plus the hidden cells used in the implementation of these accessors. Cells inside abstraction boxes (**1**) are by definition hidden (private). The image cell (**5**), which is also hidden, defines the appearance of instances of this type; whenever an instance t of type Tree needs to be displayed, a demand is generated for the image cell on t's copy of form Tree (namely, Tree(inputTree.formula = t):Image). Cell Image was specified by arranging the cells and rubberbanding the arrangement, and then editing the x-coordinates to refer to widths of the components. The underlined references refer to the (generalized) instances of the Tree form that recursively construct the left and right subtrees.

3.2.1 An introductory example: implementing a tree type

We have said that Forms/3 is a 'gentle slope' language. This implies that the linked-spreadsheet-like mechanism supporting built-in types an end user might wish to use, such as circles and boxes, needs to extend to the kinds of user-defined types that a sophisticated programmer might like to use, such as binary trees. The example in this section demonstrates this end of the slope.

To implement a new type, the programmer creates the type definition form, placing abstraction boxes and ordinary cells on it as needed and defining their formulas. Programmers will often use more than one abstraction box, placing an input abstraction box, other cells, and one or more output abstraction boxes on the definition form. However, recall (from Table 1, Definition 3) that there is always one distinguished abstraction box on the definition form, and it is known behind the scenes by the ID MainAbs.

For example, the way the tree's implementer implements a binary search tree type is shown in figures 7 and 8, and the view of the Tree definition form as seen by

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

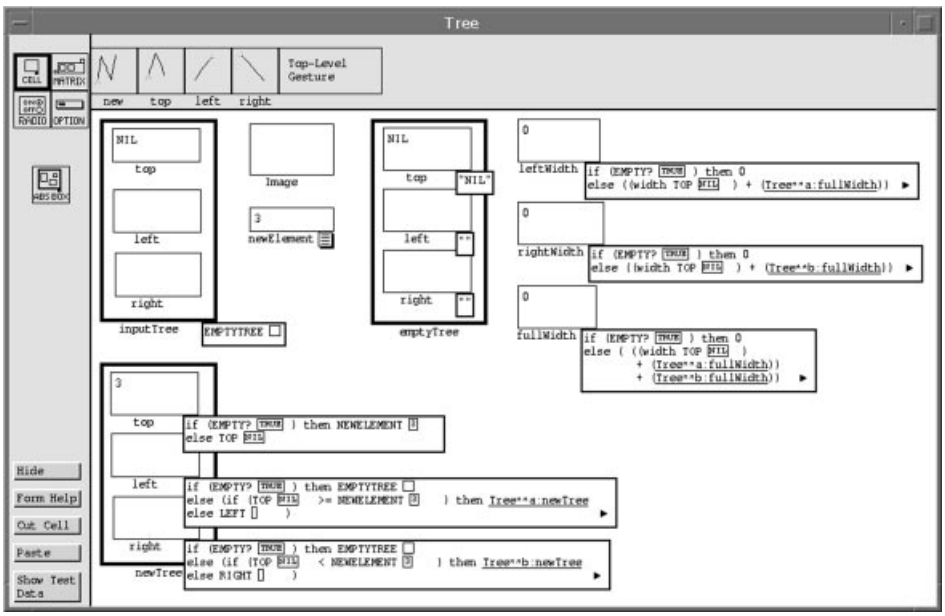


Fig. 8. The formulas defining how trees are constructed. (The accessor cells have been moved aside to make room for these formulas to be displayed.) leftWidth, rightWidth and fullWidth will be hidden; they are helper cells used by Image's formula.

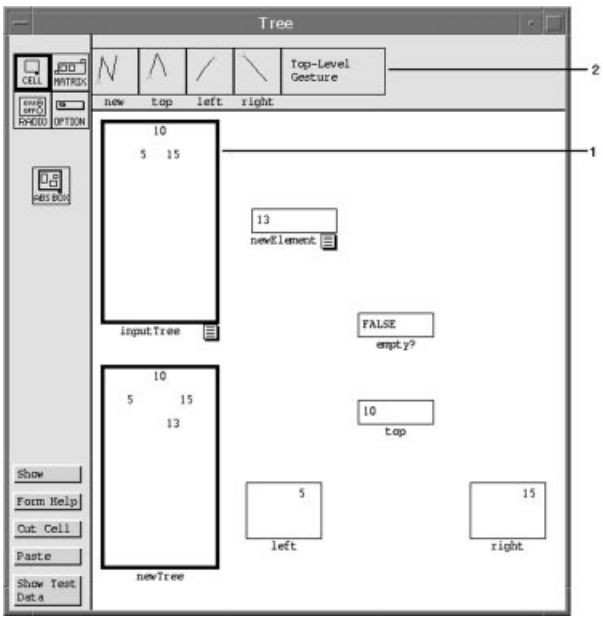


Fig. 9. View of Tree for use by other spreadsheets. The hidden cells are no longer visible because they are not accessible outside this form. Most of the cells shown here report information about the incoming tree (**1**). Tree gestures are enumerated at the top (**2**).

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 5. *The mapping from an action applied to object* α *of type* τ *to formulas.* A *represents the distinguished abstraction box* (MainAbs) *and* χ *represents the cell to be referenced on* F_{τ_β}*, which is a copy of* F_{τ_α}*. The programmer explicitly specifies which cell is* χ *when defining the new gesture's semantics.* InputDefSet *is the set* $\{\text{cell}_i.formula = \text{formulaSpec}_i\}$*, for all* cell_i *that are 'input cells'* (*non-hidden cells whose formula tabs have been left visible*) *in* $F_{\tau_\beta}.cellSet - \{A\}$*, and* formulaSpec_i *is a formula the programmer defines using the formula specifications in Table 6*

Action	Textual Formula
draw gesture, or click on gesture icon	$F_{\tau_\beta}(InputDefSet \cup \{A.\text{formula} = \alpha\})$: χ

other programmers who may wish to use the Tree type (i.e. the public interface to type Tree) is shown in figure 9. As these figures show, the form contains an input abstraction box inputTree (the distinguished abstraction box) intended to contain an incoming tree, input cell newElement for an element to be inserted into the tree, and output abstraction box newTree to define a tree into which the new element has been inserted. Other cells providing operations for the tree (such as the predicate reporting whether the incoming tree is empty, and a cell reporting the top element) are also present. Just as with the primitiveCircle type, multiple instances of type Tree can be instantiated using multiple copies of the Tree form.

3.2.2 *Direct manipulation and gestures as operations on user-defined types*

In section 2, we described how a user can use direct manipulation and gestures to program directly in the graphical vocabulary of circles, such as by selecting a circle and stretching it. Supporting these capabilities for built-in types such as circles was a way of addressing our language design goal of directness. Here we describe how to extend the same directness to user-defined types.

As one would expect, the semantics for gestures on user-defined types are a generalization of the semantics for built-in types, as can be seen by comparing Table 5 with Table 3 and Table 4. (Direct manipulations can be viewed as gestures in the context of an existing instance of a type, and hence their semantics do not really need to be separated, although they were in Tables 3 and 4 for clarity.)

To provide the *formulaSpec*s in Tables 5 and 6 that map the desired gestures to the Tree definition form's cells and formulas, the programmer must first train the gesture itself by drawing several examples of it. When the gesture training is complete, our implementation adds a miniature of the gesture to the top of the type definition form, such as the gesture miniatures at the top of figure 9. The programmer then specifies the gesture's semantics, i.e. the mapping from the gesture to a collection of cell formulas, such as in figure 10.

We have mentioned an empirical study evaluating use of gestures mapped in this way to formulas (Gottfried and Burnett, 1997). For one of the tasks in that study, subjects were required to use the tree data structure described in this section

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 6. *Explicit formula specifications. The programmer defines the* formulaSpec *of Table 5 as a one-to-many mapping from a gesture* G *on some graphical object* α *to formulas for cells* (*one of which is* X) *on form* F_{τ_β} *using the specification types shown in this table.* X_α *is the cell on form* F_{τ_α} *corresponding to cell* X *on form* F_{τ_β}. *For the user dialog formula specification* (*bottom row*), *the keyword* ask *followed by the prompt 'string' causes a dialog box to be displayed when the user makes the gesture; the user's response becomes the formula for cell* X

Type of formula specification for a cell X on form copy F_{τ_β}	Formula specification	Formula defined for X
gesture attribute	height width radius dx dy	height of user's gesture width of user's gesture radius of user's gesture dx of user's gesture dy of user's gesture
'same'	same	X_α
constant	*anything*	formula specification value (i.e. same as previous column)
user dialog	ask '*string*'	the user's response

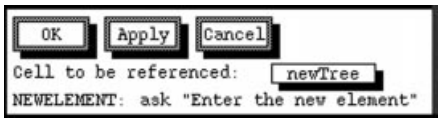


Fig. 10. Defining the 'new' gesture for type Tree. In this figure, the programmer is specifying that the new gesture means the same as a reference to cell newTree on a copy of the Tree definition form whose newElement formula is the user's input from a dialog.

to program a search. The advantages of using the graphical techniques over using strictly textual referencing were particularly pronounced in this task. In fact, each of the subjects who used the graphical programming techniques completed the tree-based program faster than *any* subject who used strictly textual referencing, and detailed analysis of the data showed that this speed was due at least in part to greater avoidance of logic errors.

4 Time and GUI I/O

We have been somewhat vague to this point about the values of cells. Rather than each cell having an atomic value, the 'value' element of cells provided in the definitions prior to this point is actually a sequence of values over logical time.

This time-oriented concept of cells and their values adds significant power to the dynamic grid and graphics capabilities. To show this, we first present the model

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

of time, and then show how it can be exploited, particularly regarding spreadsheet programming of interactive and animated graphics.

4.1 A simple model of time

Forms/3's concept of time is based on a discrete, global notion of logical time. In Forms/3, logical time is viewed as a dimension, and each computed value has a fixed, permanent position along that dimension's axis. Thus, a cell's formula defines a sequence of values positioned along that axis. Even a constant formula such as a text string is formally defined as a one-element sequence first defined at logical time 1.

Let *logical time* be defined by (T, t_1, t_{max}, *next, onOrAfter*), where time axis T is a sequence of 'time' elements from any domain in which binary operators $<, =$ and $>$ are total functions (e.g. N^+); where $\forall t_i, t_j \in T$, $t_i < t_j$ iff $i < j$; t_1 is the minimum and t_{max} the maximum element of T; *next* is a function that, given a time t_i, returns t_{i+1}; and *onOrAfter* can be any function that, given a subsequence S of T, returns a time no smaller than any $t_j \in S$. t_{max} is not really necessary, but is present for clarity and flexibility; in Forms/3, *max* is infinity.

Under this model, we redefine a simple cell to be a tuple (*cellID, formula, tv, visual attributes*), where the cell's temporal vector, *tv*, replaces the *value* element in our previous definition. Similar redefinitions are made for radio button cells, option cells, dynamic matrices, and abstraction boxes. A *temporal vector*[10] is a set of *vt-tuples* (v, $defTime$), where v is a value and $defTime \in T$ is the time at which v becomes defined. A vt-tuple (v, $defTime$) and its value v for a cell X are said to *expire* at time $defTime'$ if there exists another vt-tuple (v', $defTime'$) for cell X such that $defTime' > defTime$. A vt-tuple is *valid* from the time v is defined until it expires. Since a vt-tuple is valid until it expires, a temporal vector can be sparsely populated while still containing vt-tuples valid for every moment in logical time.

The vt-tuples that make up a temporal vector are subject to the following constraints:

(1) Uniqueness of $defTimes$: if (v, $defTime$) $\in$ cell x's temporal vector, then there exists no other vt-tuple (v', $defTime'$) $\in$ cell x's temporal vector such that $defTime = defTime'$.
(2) Constants' $defTimes$: if cell x's formula is a constant, then its temporal vector consists of a single vt-tuple with $defTime = t_1$.
(3) The past does not depend on the future: given a vt-tuple (v, $defTime$), then $\forall(v', defTime')$ such that v' is needed to compute v, $defTime \geqslant defTime'$.

The purpose of constraint (1) is to prevent a cell from having two values at the same time. Constraint (2), which says constants are defined immediately, provides a base for defining values that depend on other values. Constraint (3) is a general

[10] The term 'temporal vector' was chosen to emphasize the static association between values and indices along axis T, as distinguished from the implications of element movement, production, and consumption, normally assumed under stream-based terminology.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

Table 7. *Extending the grammar from Table 2 to include the Forms/3 operators related to time*

formula	::=	*Blank* | expr
expr	::=	*Constant* | ref | infixExpr | prefixExpr | ifExpr | composeExpr | timeExpr | (expr)
… (all the expr's specified in Table 2) …		
timeExpr	::=	EARLIER subExpr | EARLIER subExpr optionalParts | subExpr FBY subExpr | subExpr FBY subExpr untilPart
optionalParts	::=	initialPart | untilPart | initialPart untilPart
initialPart	::=	INITIALLY subExpr
untilPart	::=	UNTIL subExpr

constraint for defining values that depend on other values: it prevents defining the past or present in terms of the future. This property is useful in supporting debugging using time travel, which will be discussed in section 4.6. It also prevents cyclical relationships across time, i.e. involving more than one time in T. 'Spiral' relationships are, however, allowed, such as a vt-tuple of a cell X depending on another, earlier vt-tuple of X. Note that constraint (3) is not sufficient to prevent cyclic relationships involving multiple cells' vt-tuples at one time t, and another language following this model could choose to allow them, but the Forms/3 language implementation prevents them through other mechanisms.

The only information that can be extracted from two elements of T is whether one element is before or after the other. Thus, in Forms/3, there is no guarantee that the time steps are equally spaced, and it is not possible to subtract one from another to compute a length of time. Rather, a logical time step occurs when it is needed, either due to the arrival of some event of interest, or due to formula dependencies on previous moments in time. However, there is a built-in cell whose temporal vector reports the value of the system clock at each position on the T axis, and this cell's vt-tuples' values can be subtracted to compute elapsed clock-on-the-wall time.

Forms/3 provides two formula operators related to logical time. The syntax for these operators is shown in Table 7. The earlier operator allows reference to the value of a cell that was defined at an earlier moment of logical time, thus supporting time shifting as well as non-destructive, single-level iteration. The optional initially modifier allows specification of a value for the cell's vt-tuple at t_1 and the until modifier allows a specification that, at the first time t at which the test in the until clause becomes true, the expression's vt-tuple that is unexpired as of time t will never expire. For example, if a cell named foo had the formula 'earlier (foo + 1) initially 1 until (foo > 5)', foo's temporal vector would be $\langle(1, t_1), (2, t_2), (3, t_3), (4, t_4), (5, t_5), (6, t_6)\rangle$, and its vt-tuple at time t_6 would never expire.

Fby is a syntactic alternative to earlier inspired by Lucid (Wadge and Ashcroft, 1985), and in fact is internally implemented using earlier. It simply allows the initial value to precede the operator without a keyword, thus specifying an initial value for time t_1 and a sequence beginning at time t_2. For example, '1 fby earlier (foo + 1) until (foo > 5)' specifies the same temporal vector as the example in the previous paragraph, and '1 fby 2' would define the temporal vector $\langle(1, t_1), (2, t_2)\rangle$.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

In the context of this model of time, the complete behavior of the else-less version of if, namely 'If subExpr Then subExpr', now can be presented. For a cell foo with such a formula, if the predicate subexpression defines a vt-tuple (false, t), then foo's temporal vector is defined to contain no vt-tuple at time t. Hence, foo's preceding vt-tuple remains valid at time t, because the absence of a vt-tuple at time t means the previous one does not yet expire. This is one way temporal vectors that are sparse can be defined. Through this facility of one temporal vector being sparser than another, a programmer can control the relative rates of speed of different cells' computations, which is useful for applications such as animations and GUI I/O.

4.2 GUI input

In traditional spreadsheet languages, inputs are not delayed until the execution of some get-like function; rather, input values are simply constants specified by static formulas. Forms/3 supports this input model. In addition, in order to support spreadsheet programming about sequences of event inputs such as mouse events, there is a temporal form of input.

In Forms/3, event queues record sequences of GUI events. An *event queue* is a special kind of cell that resides on the distinguished form System, and has the constraint that for any two vt-tuples $(event_i, defTime_i)$ and $(event_j, defTime_j)$ in its temporal vector, $defTime_i < defTime_j$ iff $event_i$ happened at a 'real' (clock-on-the-wall) time before $event_j$. Event queues are activated when they are associated with other cells by virtue of a cell's formula referencing an *event receptor*, whose purpose is to define and report activity in the associated event queue. An event receptor is similar to some languages' non-blocking input operators, such as those used for input polling.

The events an event receptor can report are determined by a tuple: (*name, eventsOfInterest, transparent, shape*), where *name* associates the event receptor with an event queue, *eventsOfInterest* is the collection of event types that should be considered 'interesting' to the associated event queue, *transparent* is a Boolean specifying whether events are allowed to propagate to event receptors that are spatially covered by this event receptor, and *shape* defines the event-sensitive area (note: this is geometrical area, not screen location). By the principle of referential transparency, if two instances of event receptor have identical tuples, then they are identical; hence they are associated with the same event queue and report identical events.

Event receptors, like other primitive types such as primitiveCircle, are defined on built-in type definition forms. Thus, as with other types, multiple instances of event receptors can be created by making copies of the eventReceptor form, and instances of event receptors can be composed with values of other types. Figures 11 and 12 show a thermometer application that makes use of event receptors. The thermometer displays the temperature entered in the input cell. The user can press the F<–>C button in order to toggle the thermometer between displaying in Fahrenheit or Celsius. The formula for the button contains a reference to cell eventReceptor, which is an abstraction box on the primitive eventReceptor form

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

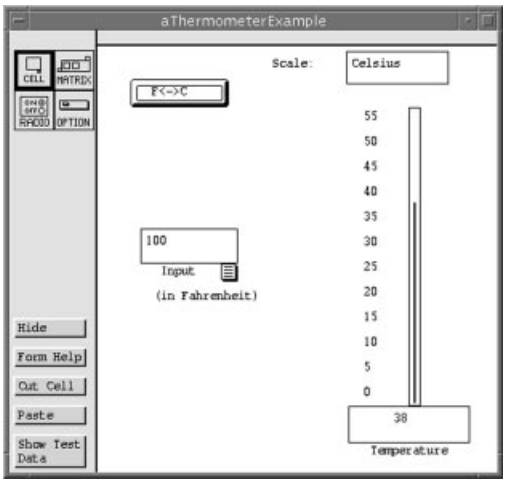


Fig. 11. The thermometer application, shown from the user's point of view. A formula tab has been left visible to show where the user is expected to provide an input formula.

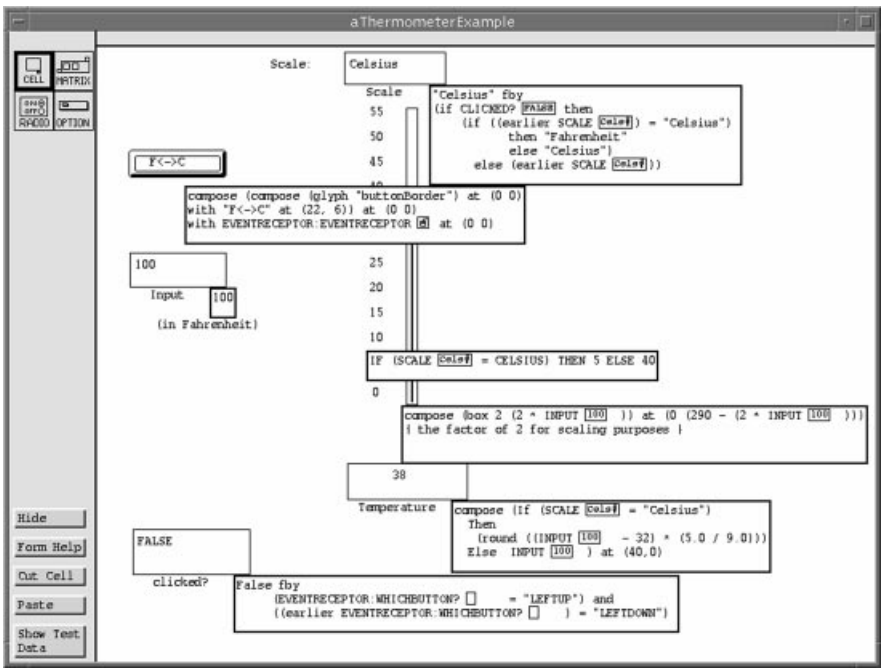


Fig. 12. The thermometer form with all formulas shown. Button (cell) F<–>C references eventReceptor, shown in figure 13.

(shown in figure 13). The clicked? cell in figure 12, which is normally hidden from the user, detects F<–>C button clicks given the low-level event information reported by the event receptor, which is used in turn by the Scale cell to toggle the temperature scale.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

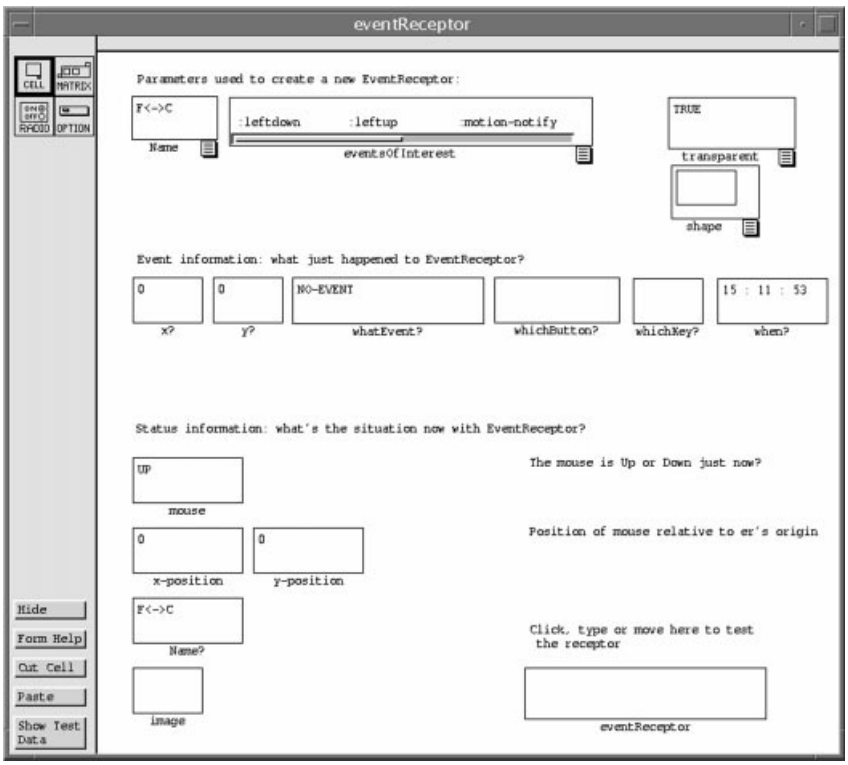


Fig. 13. Mouse and keyboard events can be referenced by referring to cells on this event receptor form. Formula tabs are present on the modifiable cells. Cell eventReceptor (bottom right) is the abstraction box.

4.3 Sequencing interactive I/O: an 'interactive deadlock' problem

Appropriate sequencing of I/O is often necessary for successful communication with the user. However, as Wadler eloquently explains in his discussion of functional I/O, correctly interleaving the sequence of interactive I/O has long been a problem for applicative languages (Wadler, 1997). The earliest 'pure' approach, synchronous streams (Landin, 1965; Stoye, 1986), used for example in one version of Haskell, relied upon dependencies to implicitly control sequence. Unfortunately, it was often difficult for programmers to interleave inputs and outputs correctly with only this implicit mechanism for controlling sequence, resulting in waits for input before the prompts appeared and similar problems. Because of this difficulty, many other approaches to I/O have been developed for applicative languages.

Some of these approaches have been: imperative side-effecting constructs as in SML (Milner *et al.*, 1990), linear logic (Wadler, 1990; Achten and Plasmeijer, 1995), continuations (Perry, 1989; Hudak *et al.*, 1992) and monads (Peyton Jones and Wadler, 1993; Launchbury and Peyton Jones, 1994; Wadler, 1997). Monads in combination with concurrency can be used to extend monadic I/O sequencing to the concurrent needs of GUIs, as has been shown by Haggis (Finne and Peyton Jones, 1996), a framework for writing GUIs in Concurrent Haskell (Peyton Jones

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

et al., 1996) via explicit use of monads. A related approach is that demonstrated by the Fudgets system (Carlsson and Hallgren, 1993; Hallgren and Carlsson, 1995). Fudgets are processes that communicate by message passing, via communication paths created using combinators.

Despite their successes for some situations, none of these approaches seemed viable for Forms/3. ML's side-effecting approach violates referential transparency, which we would like to preserve in Forms/3. Linear logic works by allowing no more than one interaction on a single state variable, thus enforcing a linear sequence on I/O actions. However, if placed in a spreadsheet setting, a 'state-oriented' cell could only be referenced by at most one other cell, which would be inconsistent with common spreadsheet practices. Monads and continuations do not seem viable for spreadsheet languages because many programmers of spreadsheet languages are end users, who are not likely to have experience working with continuations, monads or the higher-order functions employed by these approaches.

In understanding the possible solution space for the problem of interleaving interactive I/O correctly, it is useful to view the problem as a deadlock problem. If I/O operations are not sequenced correctly, a situation we term *interactive deadlock* can occur. To understand how the concept of deadlock applies to interactivity, think of the user as filling the role traditionally held by a process running in a computer system, with the application program being another process in the same system. Recall the four classical conditions necessary for deadlock: mutual exclusion, no preemption, hold and wait, and circular wait (these can be found in most textbooks on operating systems (e.g. Silberschatz and Galvin, 1998). Deadlock in the context of interactive I/O then, can occur if an incorrect program waits for input before producing any output (holding all output while waiting for input), while the user waits for some prompt-like output before realizing that input is expected (holding the input while waiting for the prompt).

Viewed from the perspective of interactive deadlock, previous I/O sequencing mechanisms have all been aimed at removing circular wait. If sequencing can be explicitly controlled by the programmer, the application program can be written to always release the prompt before waiting for the input, thereby preventing circular wait.

Giving the programmer mechanisms to prevent circular wait is not the only possible solution to interactive deadlock. Removing any of the other three conditions can also solve the problem, and Forms/3 does so by removing the 'hold and wait' condition. Although Forms/3's temporal vectors are much like synchronous streams, user interaction can be supported without requiring the programmer to explicitly sequence operations. The Forms/3 approach is to spread the interaction over space concurrently – via multiple streams, each monitored in the language implementation with its own concurrent thread, at least in theory – instead of solely over time implemented by only one thread. (Forms/3's implementation does not really create a different thread to monitor each cell, but that would be one possible way to implement the behavior we describe here, and pretending that it is implemented in this way helps to illustrate the essence of the approach.) This allows the 'fill in the blanks' approach of commercial spreadsheet input entry to be generalized to

event-based I/O, i.e. giving control over most I/O sequencing to the ultimate user of the spreadsheet instead of to the spreadsheet's programmer, as advocated in Dix's proposal (Dix, 1987).

Key to the Forms/3 approach is the fact that Forms/3 input is non-blocking. Even computations dependent on an un-entered input are not blocked, because in our model of time there is always a value defined. From this and from the presence of liveness, it follows that output is always possible, even before important inputs have been entered.

For example, first consider the traditional spreadsheet approach to input. Suppose some cell A's formula is 'Enter total income on line 1' and cell line1's formula is blank. Here, the prompt is always present when cell A is on the screen, and the user can provide the input in cell line1 whenever it is convenient, by modifying line1's formula. One reason it is possible for the user to choose the time to provide the input is because in Forms/3, as in traditional spreadsheets, cell line1 has a value even before the user changes its formula – in Forms/3 it is the distinguished value noValue. Now suppose cell A's formula is 'Click the F<–>C button to toggle between Fahrenheit and Celsius', and cell F<–>C's formula is as in figure 12. As in the 'Enter total. . . ' example, the prompt is always present, and the user can provide the input whenever it is convenient to do so, this time by clicking the F<–>C button rather than by changing a formula. Also as in the 'Enter total. . . ' example, all the cells have values even before the user provides inputs: for example, the value in the visible vt-tuple of figure 13's cell whatEvent?, whose temporal vector reports the user's interactions with the F<–>C cell, is NO-EVENT. This vt-tuple's *defTime* is t_1, and it will not expire until the user interacts with the button at some time t_i ($i > 1$ by the constraint of section 4.2). If the user's interaction with the button was a leftdown event, then a new vt-tuple (leftdown, t_i) will be defined for cell whatEvent?.

As is demonstrated in both of the examples in the above paragraph, the system neither holds (the output) nor waits (for input). This technique allows straightforward programming of GUI objects, such as the use of the F<–>C button, and of a variety of business applications, in which fill-in-the-blanks paper forms are simply mimicked by a spreadsheet.

There are some programs, however, in which interleaved sequencing of input versus output over time is required due to direct dependencies between them, such as needing to vary a particular cell's value after each mouse movement. This case is handled in a straightforward way via formula dependencies on the event receptors. For example, to produce output dependent on interactive input, a cell could be given a formula such as:

```
if (123-EventReceptor:whatEvent? <>  ":motion-notify")
    or (123-EventReceptor:when? > System:time[11] + 10)
        then "Please move the mouse" else "Thank you!"
```

[11] Form System's cell named Time provides access to the system clock: Time's temporal vector is the

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

4.4 Dynamic graphical output: the implications of liveness

As the examples to this point have shown, like other live spreadsheet languages, Forms/3 automatically maintains the display of the values of all on-screen cells. Thus, output is implicit: there is no call to a put-like function; rather the language automatically evaluates visible cells whenever doing so is necessary to ensure that the display is up-to-date. For example, since figure 12's cell Temperature directly references cell input and transitively references cell whichButton?, then whenever input has a new vt-tuple due to a formula edit or whenever whichButton? has a new vt-tuple due to a mouse click, then Temperature by definition also has a new vt-tuple, which will be computed if needed. If input's formula were changed to monitor temperature readings coming in from a satellite, then cell Temperature's display would be animated. Thus, Forms/3's output is simply a by-product of level 4 liveness.

Level 4 liveness can be modeled abstractly by the tuple (S, E, M_T, *Whenever*), where S is the current state including a program as defined in Table 1, all values, and a display state; E is a sequence of input and/or edit events such as mouse clicks and formula edits; M_T is a model of time such as that presented in this paper; and *Whenever* is a never-ending function that takes state S and the most recent event of E and produces a new state S' according to M_T, and then invokes itself again on S' when the next event arrives. If M_T were not included, this model of liveness would also describe traditional spreadsheet languages' liveness level 3. This model makes clear that the effects of liveness on a language are fundamental, since liveness's *Whenever* function both supercedes the use of traditional output constructs and generates computational behaviors for the purpose of maintaining the display state.

At first glance, the *Whenever* function may seem to be inherently eager, but this is not the case – it is entirely compatible with lazy evaluation. As in non-live lazy languages, all demands for computation start at the outputs, but in a live language, everything on the screen is an output. Hence, demands are concurrently generated for all on-screen values, which then propagate backwards through dataflow paths in the usual way. Thus, everything on the screen is demanded, plus the off-screen values needed to produce those on-screen values, but off-screen values not needed for the on-screen values are not demanded.

It is the *Whenever* function that transforms a spreadsheet's collection of formulas from a single-threaded sequence of 'function calls' into a partially-ordered network of one-way, equality constraints. This relationship between spreadsheet programming and one-way equality constraint programming, when considered in the realm of time-varying interactive graphics, suggests that the successes in using one-way equality constraints for straightforward GUI specification (e.g. Bharat and Hudson, 1995; Carlson *et al.*, 1996; Hill, 1993; Hudson, 1994; Myers *et al.*, 1990, 1996; Vander Zanden and Myers, 1995; Vander Zanden and Venckus, 1996) can potentially be brought to bear on the problem of functional I/O.

sequence of times reported by the system clock. (This approach to the system clock preserves referential transparency for all programs run within a single Forms/3 session.)

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

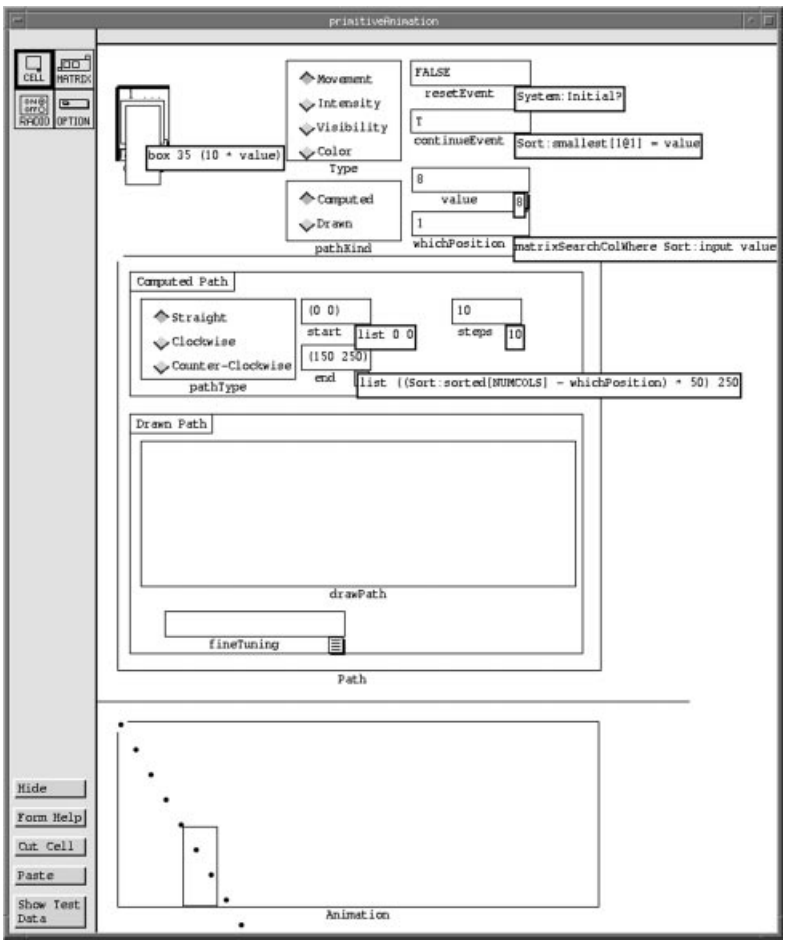


Fig. 14. An animation form for one element of the selection sort animation. The parameters are established in cell formulas at the top and middle (through a flexible combination of text and/or drawing), and the result is at the bottom. Automatic generalization of the formula that references this form causes, on a lazy basis, a copy of this form to be created to animate whichever element is actively being sorted.

4.5 An application of dynamic graphics: software visualization

We have pointed out that the presence of temporal vectors and the ability to see them evolve over time on the screen leads naturally to the support for animated graphics. A primary interest to us in supporting animation has been as a dynamically-computed documentation mechanism for supporting program understanding. This is an example of the subarea of research known as *software visualization.*

Forms/3 has several graphical devices intended to aid in program understanding, but we focus here only on animation as dynamically-computed documentation. Animation in Forms/3 is straightforward, due to the full support of graphical types in combination with the model of time and liveness. As has already been demonstrated, some animation is possible in Forms/3 without additional features,

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

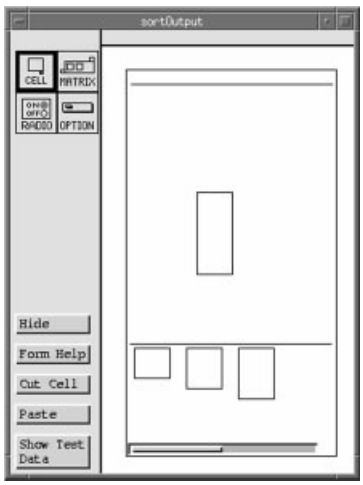


Fig. 15. A sort animation shows the elements of the unsorted group at the top being moved one at a time to the sorted group at the bottom. The final element is moving from the top left corner to the bottom right at this point in the animation.

but Forms/3 provides additional animation functionality through an animation type via an animation form (figure 14). For example, to provide animated documentation of a selection sort, a programmer may wish to emphasize the 'move' portion of the algorithm, having each element step across the screen to its new location. To specify such an animation, the programmer gives formulas for the intermediate positions through which a graphical depiction of an element should travel, either by specifying straight/clockwise/counter-clockwise and the start, end, and number of steps, or by directly drawing the path (middle of figure 14). When this form is used to create an animation of one element of a dynamic matrix, it is automatically generalized for the other elements of the dynamic matrix (Carlson *et al*., 1996). After this generalization of figure 14, the result is as shown in figure 15. For animation effects other than spatial movement, the programmer can select options on the animation form to specify paths through 'visibility space' (for fade-in/fade-out sequences), through 'color space' (for gradual color transitions), or through 'intensity space' (for brightening/dimming transitions). It is possible to imagine additional options for 'orientation space' (for rotations) and 'magnification space' (for scaling), but we have not implemented these.

Since the 'input' cells for one animation (upper left of figure 14) can reference the result cell (bottom of figure 14) of another animation, animation effects can be composed in arbitrary ways. This effect is transitive; that is, other cells referring to a cell whose (textual or graphical) values vary over time will also be animated over time. Thus, any cell referencing the result cell on an animation form, or in fact referencing any other time-varying cell, will also vary over time.

This transitivity is also present in other time-varying languages such as those termed 'synchronous' or 'reactive' languages, such as Lucid (Wadge and Ashcroft, 1985; Du and Wadge, 1990) and related languages such as Chronolog, Esterel

and LUSTRE (Orgun and Wadge, 1992; Liu and Orgun, 1996; Halbwachs, 1993); however, these languages did not extend support to the realm of graphics and animations. Fran (Elliott and Hudak, 1997) is a recent Haskell-based system that supports graphics and animations through constraint-like relationships such as Forms/3's, but is based upon a continuous model of time as opposed to our discrete approach. The Fran approach, as well as Haskell's earlier approaches to I/O, have some similarities to Arya's seminal work on functional animation (Arya, 1989). Fran, Haskell and Arya's work all use devices not present in the spreadsheet paradigm such as higher-order functions and monads. The visual dataflow language Viva (Tanimoto, 1990) was perhaps the first first-order language specifically aimed at visually working with images that vary over time, though Viva was not aimed at generating animations, but rather at responding to changes in image data as they arrived from the data source.

4.6 Time travel and steering

The model of time just presented, when combined with referential transparency and liveness, provides an opportunity for an environment to support *time travel*, the ability for a spreadsheet programmer to return to (or move ahead to) a previous (future) step of a time-based computation. Forms/3 takes advantage of this opportunity, and with it provides the ability to *steer* programs. This term comes from the scientific visualization community, and means the ability for the programmer to interactively modify *any* portion of the source code at any time, and immediately see the effects without restarting the computation (McCormick *et al.*, 1987).

Steering can be thought of as an extension of interpreter functionality. Standard interpreters allow code to be replaced and execution to be resumed, but the underlying system state after such a change may be contradictory, and worse, the display screen after such a change is usually inconsistent with the underlying system state. Steering in Forms/3 eliminates these inconsistencies through the liveness that implements each formula as a live constraint that must be maintained. It also eliminates programmer effort switching between programming mode and debugging mode (see the discussion of viscosity in Appendix A).

Note that, following the model of time of Forms/3, previous moments in time are not historical (like 'undo'), but rather the way values at previous positions in logical time would have been under the current collection of formulas. The ability to traverse historical time is the kind supported by version control systems, by a few visual programming languages' undo capabilities such as KidSim/Cocoa (Cypher and Smith, 1995), and by a few visual debuggers such as PROVIDE (Moher, 1988). On the other hand, Forms/3's time travel backward through logical time is closer to that of the debugger for Tolmach and Appel's concurrent extension of Standard ML, which has reversible logical time (Tolmach and Appel, 1991, 1993). Other related approaches include Baker's reversible Lisp (Baker, 1992), the Transparent Prolog Machine (Brayshaw and Eisenstadt, 1991), which provides graphical visualizations of Prolog queries that can be viewed at variable speeds forward and (if viewed post-mortem, but not live) in reverse, and SPYDER (Agrawal *et al.*, 1993), which

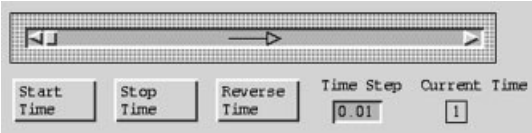


Fig. 16. The slider used for time travel in Forms/3. This device allows interactive navigation through the on-screen cells' temporal vectors.

is an example of how backwards time travel can be supported for debugging in an imperative language. However, the closest approach to Forms/3's time travel capability is ZStep (Lieberman and Fry, 1995, 1997), a visual debugger for a subset of Common Lisp, which provides support for time travel, for viewing how values and code are related, and for live graphical stepping.

From the view of debugging as a 'locate-fix-verify' process, the differences between these prior approaches and Forms/3's are that prior systems use time travel but not steering, thereby supporting only the 'locate' step of debugging, whereas Forms/3 also facilitates the 'fix' step by allowing the bug to be corrected in context, and facilitates the 'verify' step by automatically and immediately redisplaying values of all on-screen cells affected by the 'fix' step.

For example, consider again the thermometer example in figures 12 and 13. It contains a bug: some mouse clicks do not cause the Scale value to toggle. A spreadsheet programmer can begin debugging by using time travel to try to understand this behavior. Having gained an understanding of the cause (the 'locate' step), the programmer can, via the steering capability, edit in the necessary changes without losing context (the 'fix' step), receiving immediate feedback as to the effects of these changes (the 'verify' step). The details of such a debugging session might proceed as follows.

The formulas for Scale and the F<–>C button are hidden from end users, but the programmer can interactively unhide the formulas. The programmer examines the two formulas, and sees that the Scale cell depends on a hidden cell, named clicked?, and that both clicked? and the button make use of the eventReceptor in figure 13. Bringing this form onto the screen, the programmer travels backward and forward through time using the slider shown in figure 16 to explore how the behavior of the eventReceptor might be affecting the Scale cell. Looking at the eventsOfInterest cell, the programmer sees that an irrelevant event type – Motion-Notify – is being attended to by the button, separating Button-Press, the first half of a click, from Button-Release, the other half (see figure 17). This is the bug. The programmer edits the formula of eventsOfInterest to remove Motion-Notify.

To find out if this edit fixed the bug, the programmer explores the now-redefined history via time travel. It is inherent for the program's entire history to be redefined according to this change because cells' histories are defined solely by their formulas and attributes. This is another way liveness is used to support debugging – as soon as a change is made, all affected histories are automatically (but lazily) redefined and all affected on-screen values are automatically recomputed and redisplayed, maintaining consistency between the system state and the display state. Thus, time

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

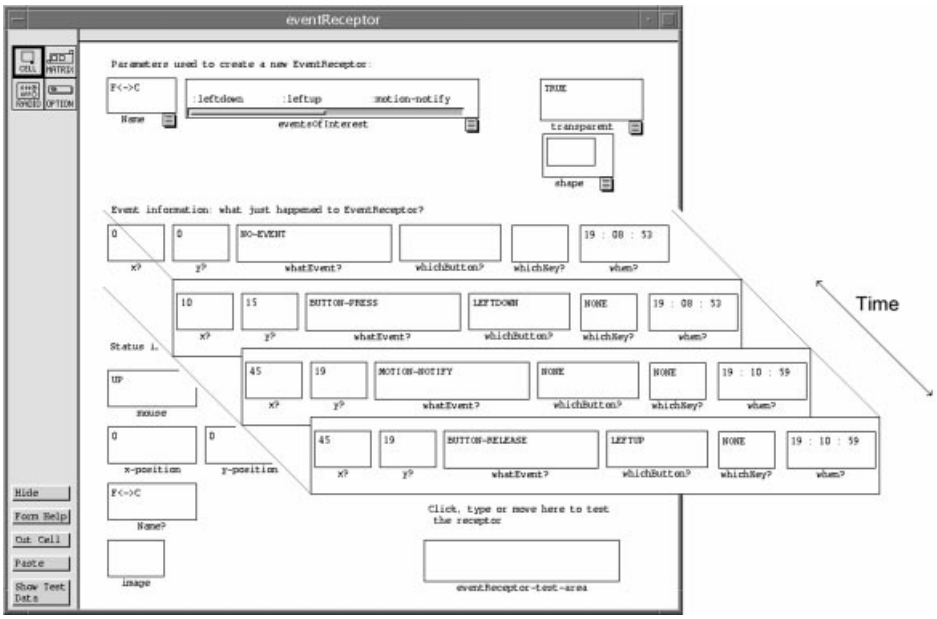


Fig. 17. An annotated sequence of screen shots depicting the sequence of values in time for cells on form eventReceptor. The programmer travels through time by manipulating the time slider.

travel now reflects history as computed under the new version of the program. This allows the programmer to explore the program to determine whether the values changed as expected. The programmer is spared the usual effort of mode switching: re-running the program repeatedly, instrumenting the program with breakpoints or diagnostic statements, recompiling and reconstructing the context in which the bug occurred before. In this example, the programmer sees that the clicks are now all recognized, and the bug is fixed.

To preserve liveness's immediacy of feedback, time travel must be efficient. Our approach allows tunability by the language installer regarding the emphasis on space versus time efficiency. Regarding space efficiency, in Forms/3, all of a cell's sequence (history) is completely defined via its current formula and attribute set, making the storage of the actual values unnecessary for correctness. The only information in addition to a cell's formula and attributes that absolutely must be stored are the user events (mouse clicks, etc.). (In fact, in our implementation, we store only a subset of user events – the user events of types and locations declared to be of interest to an event receptor – which is sufficient unless the formulas are edited to express interest in new types of events or in larger spatial areas, in which case the system loses backward time travel capability for events before the edit.) This definition-based approach means that the history of a Forms/3 program can be stored using only the amount of space required for the source code plus the relevant user events.

Time efficiency is aided by laziness. In the above example, when the programmer edits eventsOfInterest, Scale is the only cell that needs to be recomputed from

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

the beginning of time, because it is the only one that depends upon its own earlier values. There are other cells affected by the programmer's edit (via dependencies on eventsOfInterest or on Scale), but their vt-tuples at earlier moments in time are not needed for output and hence are not computed: the only vt-tuples that are needed for these other cells are those on display at the current time and whichButton?'s vt-tuple just before (to decide if a click has occurred).

Caching also provides time efficiency opportunities: although the system need not save unchanged cells' temporal vectors for correctness, clearly response time can be improved if it does cache at least some of them. For example, without caching, a programmer traveling back and forth through time could force the program to re-display the same values many times, generating duplicate computations. To solve this problem, as much cache space as desired can be used to reduce the number of duplicated computations via lazy memoization (Hughes, 1985), an adaptation of memoization (Michie, 1968) for lazy evaluation. Although management of the cache itself requires time, it has been shown both theoretically and practically to be far less than the time required to maintain the display without saved values (Burnett *et al.*, 1998).

5 Related work

5.1 Related work on spreadsheet languages

Two widespread limitations in prior spreadsheet languages have been in the limited types supported and the lack of abstraction capabilities. For example, commercial spreadsheet languages support graphics as decorations or as outputs based upon spreadsheet values, but many do not support interactive graphics or graphical types as first-class values that may be incorporated into other computations, and do not support user-defined graphical types. Commercial spreadsheet languages that do support computations on graphics have done so through devices that are incompatible with the value rule, namely via imperative macro languages and escapes to traditional imperative languages.

One of the pioneering research spreadsheet languages to address graphics in spreadsheets was NoPumpG (Lewis, 1990) and its successor NoPumpII (Wilde and Lewis, 1990), two early spreadsheet languages designed to support interactive graphics without macros or other non-formula devices. The design goal of these languages was to provide the capability to create low-level graphical primitives while adding as little as possible to the basic spreadsheet paradigm. Thus, NoPumpG and NoPumpII include some built-in graphical types that may be instantiated using cells and formulas, and support limited (built-in) manipulations for these objects, but do not support complex or user-defined objects.

Several research projects have aimed at extending spreadsheet language functionality through imperative devices. Penguims (Hudson, 1994) is an environment based on the spreadsheet model for specifying user interfaces. Its goal is to allow interactive user interfaces to be created with little or no traditional programming. Its support for abstraction is similar to Forms/3's – it provides the capability to collect cells

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

together into 'objects' – but unlike Forms/3, it employs several techniques that do not conform to the spreadsheet value rule, such as interactor objects that can modify the formulas of other cells, and imperative code similar to macros. Action Graphics (Hughes and Moshell, 1990) is a spreadsheet language for graphics animations. It too provides some support for complex objects. Animation in Action Graphics is performed through functions that cause side-effects. Smedley, Cox and Byrne have incorporated the visual programming language Prograph and user interface objects into a conventional spreadsheet system in order to provide spreadsheet users with graphical interface capabilities (Smedley *et al*., 1996). The Prograph approach includes imperative devices and side effects. SIV (Spreadsheet for Information Visualization) is a recent spreadsheet research effort aimed at supporting information visualization (Chi *et al*., 1998). SIV formulas are state modification oriented: the syntax for formulas is 'command result–cell arguments'. SIV formulas and cellnames can also employ general Tcl code/variables, an approach also followed by Levoy's Spreadsheet for Images (Levoy, 1994).

C32 (Myers, 1991) is a spreadsheet language that uses graphical techniques to specify user interfaces. Unlike the other spreadsheet languages described here, C32 is not a full-fledged spreadsheet language; rather, it is a front-end to the underlying textual language Lisp used in the Garnet user interface development environment (Myers *et al*., 1990). C32 is a way of viewing one-way constraints, but does not itself feature the graphical creation and manipulation of graphical objects. Instead, this function is performed by the demonstrational system Lapidary (Vander Zanden and Myers, 1995), which is another part of the Garnet package. The combination of C32 and Lapidary (and the other portions of the Garnet package) features strong support for direct manipulation of built-in graphical user interface objects, but not for any other kinds of objects, which must be written and manipulated in Lisp.

Forms/3 is a descendent of two earlier languages that explored ways to expand the spreadsheet paradigm, Forms (Ambler, 1987) and Forms/2 (Ambler and Burnett, 1990). The spreadsheet language Formulate (Ambler and Broman, 1998; Ambler, 1999) is another descendent of these two languages. Formulate has been used primarily as a vehicle to research the support of matrix-oriented computations using multiple levels of formulas (Viehstaedt and Ambler, 1992; Wang and Ambler, 1996). Recent work on Formulate also incorporated the use of voice, handwriting, and gestures as input modalities for fine-grained entry of spreadsheet formula operands and operators, all three of which modalities can be mixed in the entry of a single formula (Leopold and Ambler, 1997). On the other hand, Forms/3's foci have been primarily on abstraction, such as in combining data abstraction with direct manipulation, and on the use of a logical time dimension in spreadsheet programming.

5.2 Related work on visual languages

Forms/3 has been influenced by work in several types of visual programming languages, especially by demonstrational programming languages (Cypher, 1993), which support programming by direct manipulation of objects. Of these, the most

closely related to our work are those featuring a declarative approach, which to date have followed either the rule-based or the constraint-based paradigm. KidSim/Cocoa (Cypher and Smith, 1995) and Visual AgenTalk (Repenning and Ambach, 1996) are demonstrational systems that use direct manipulation to specify declarative graphical rewrite rules. Although the approaches used by these systems have some similarity to ours in their support for directness using a declarative mechanism, they do not provide full-featured, declarative specification of objects and attributes.

The multi-way constraint systems TRIP3 (Miyashita *et al*., 1992) and IMAGE (Miyashita *et al*., 1994) also use direct manipulation as a means of specifying relations declaratively. In these systems a visual example defines a relationship between the application data and its visual representation. One fundamental difference from Forms/3 is that the purpose of TRIP3 and IMAGE is to provide a visual interface to traditional textual programming languages, while Forms/3 aims to extend the power of the spreadsheet paradigm without involving any other programming language. Another fundamental difference is that that TRIP3 and IMAGE use multi-way constraints, which are not consistent with the spreadsheet value rule. To see why, imagine specifying the formula for cell X to be a box whose width is a reference to cell W (whose formula is cell A plus cell B). If the user then selects and stretches the box in X, what does that mean for cells W, A, and B? If any of these were automatically changed, the value rule would be violated for the changed cell(s); if they were not changed, the multi-way nature of the constraints would not be maintained.

6 Continuing and future work

A continuing theme of this research has been scalability, a problem suffered by many visual and end-user languages (Burnett *et al*., 1995), and we have been working on that problem from both language design and software engineering directions. One of our projects in the area of language design has been to devise an approach to exception handling that extends the 'error value' model followed by most commercial spreadsheet languages to allow user-defined exceptions and to support 'replacement value' exception handling (Burnett, Agrawal and von Zee, 2000). Another feature in progress is a new, fine-grained approach to inheritance termed *similarity inheritance* (Djang and Burnett, 1998). Similarity inheritance is similar to copy/paste, but maintains relationships among duplicated formulas; it is intended to bring some of the benefits of traditional inheritance to spreadsheet users who are not trained in traditional inheritance. One of the challenges with this fine-grained, relatively unstructured approach is that extensive support from the language and environment seems necessary to make it usable. As in many other applicative languages, type information can be used to help with this task. The current implementation of Forms/3 is dynamically typed, but we are working on a model of static type inference that can operate at the fine-grained level necessary to support similarity inheritance (Djang *et al*., 2000).

Regarding software engineering issues, we are currently working to explicitly support debugging and testing of programs written in spreadsheet languages. We

began that work by conducting an empirical study to learn more about how liveness affects debugging in this paradigm (Wilcox *et al.*, 1997; Cook *et al.*, 1997). We are in the process of building upon that work by developing a new methodology for testing spreadsheets that can help with both testing and debugging (Rothermel *et al.*, 1997, 1998; Burnett *et al.*, 1999; Reichwein *et al.*, 1999). Our methodology includes several test adequacy criteria and low-cost incremental program analysis techniques, and uses them to track how thoroughly tested each cell is according to the selected criterion. The cell's 'testedness' status is updated automatically after each user action, and the visual feedback mechanism continuously communicates this status using the border color of each cell.

Debugging support relates strongly to the ability to see any value at will, both intermediate and final answers. Spreadsheet languages generally already support this capability over space, and we have shown how Forms/3 also supports it along the time dimension. In fact, another way of looking at support for time travel is that it simply extends the capabilities already available in spatial dimensions in spreadsheet languages to another dimension (time). Looking at time travel in this way has recently led us to develop a continuum of temporal programming and visualization models that make various trade-offs between supporting time as just another dimension in the spreadsheet world, versus allowing programming of specifically temporal attributes such as speed relationships (Burnett, Cao and Atwood, 2000).

7 Implementation status

Forms/3 is currently implemented in Liquid Common Lisp with the Garnet user interface system (Myers *et al.*, 1990). We also have a Java version in process. The Forms/3 implementation is publicly available at:

http://www.cs.orst.edu/~burnett/Forms3/forms3.html

8 Conclusion

One of the primary goals of the research for which Forms/3 serves as a prototype is to test the limits of the spreadsheet paradigm, both from the perspective of language design issues such as computational power and expressiveness, and from the perspective of human-oriented issues such as usability and directness. As the results presented in this paper show, it is possible to leverage the spreadsheet paradigm far beyond the current state of practice to include features such as the following:

- *Graphical types*: the support of both primitive and user-defined graphical types as first-class types allows them to be created and accessed in formula calculations.
- *Gestural programming*: the ability to 'call' operations using contextual direct manipulations and gestures promotes directness. An empirical study indicated that this type of syntax improved programming speed and accuracy.

- *Dynamically-sized grids*: dynamically-sized grids provide the functionality of both lists and traditional matrices, allowing a wider range of calculations to be specified than has been possible using the statically-sized, statically-referenced grids of commercial spreadsheet languages.
- *Generalized abstractions*: both procedural abstraction and data abstraction are possible in the spreadsheet paradigm without employing function definitions or other devices from traditional languages, instead using only a variation of linked spreadsheets coupled with an entirely deductive approach to generalization.
- *Graphical I/O*: combining a simple model of time with graphical types and liveness allows event handling and animations to be supported without the use of higher-order functions, and without the synchronization problems of prior stream-based approaches. This allows a straightforward, yet fully declarative approach to GUI I/O and animated graphics without requiring the addition of higher-order functions to a spreadsheet language.
- *Time travel and steering*: the declarative semantics combined with the live evaluation model makes advanced programming environment features such as steering programs (modifying source code in context while observing resulting changes) viable. These features are of particular relevance to debugging.

Most important, these features are possible without the use of impure solutions such as imperative macro languages or trapdoors to traditional programming languages, thereby opening the possibility of the use of these features by several different kinds of populations, including not only professional programmers, but also end users.

Acknowledgements

Many people have made important contributions to continuing progress on the language and implementation research for which Forms/3 serves as a prototype. We thank Tim Adams, Anurag Agrawal, Allen Ambler, Derrick Boom, Jonathan Cadiz, Mingming Cao, Nanyu Cao, Paul Carlson, Roger Chen, Maureen Chesire, Curtis Cook, Frank Cort, Christopher DuPuis, David Hackenyos, Judith Hays, Lixin Li, Sunanda Mishra, Rajeev Pandey, Gregg Rothermel, Karen Rothermel, Andreas Schoberth, Andrei Sheretov, Gerhard Viehstaedt, Zachary Welch, Eric Wilcox and Pieter van Zee for their creative ideas and their hard work that have helped shape Forms/3. Special thanks are due to Christopher DuPuis for his programming of an earlier version of the Turing machine program.

This research has been supported by Hewlett-Packard, by Pictorius, by Harlequin, by Rebecca Djang's NASA Graduate Student Researcher Award, and by the National Science Foundation under NSF Young Investigator Award CCR-9457473 and grants ASC-9523629 and CCR-9806821.

Appendix A: HCI research and spreadsheet language design

A beneficial side-effect of the focus of the spreadsheet paradigm on end users has been that it has brought extensive Human-Computer Interaction (HCI) research to bear upon spreadsheet language design (e.g. Nardi (1993); Hendry (1995); Hendry and Green (1993)). Four features for which there has been work that is of particular relevance to spreadsheet languages are (1) directness, (2) viscosity, (3) immediate visual feedback, and (4) hidden dependencies.

The term *directness* as used in the HCI community expands upon the term *direct manipulation*, first coined by Shneiderman to describe three principles: continuous representation of the objects of interest; physical actions or presses of labeled buttons instead of complex syntax; and rapid incremental reversible operations whose effect on the object of interest is immediately visible (Shneiderman, 1983). Hutchins, Hollan and Norman expand upon these notions, suggesting that the degree to which a user interface feels direct is inversely proportional to the cognitive effort needed to use the interface (Hutchins *et al*., 1986). They describe directness as having two aspects. The first aspect is the *distance* between one's goals and the actions required by the system to achieve those goals. In traditional spreadsheet programming, distance is fairly small because the goals, which traditionally have to do with finance-oriented mathematics, can be accomplished using a mathematical vocabulary. For example, the goal 'what is the sum of column A' is expressed via the formula 'sum(A1 : A12)' instead of requiring recursion or a vocabulary of loops and state modification. In contrast to spreadsheet languages, Green and Petre enumerate several examples showing the unfortunate lack of this aspect of directness (termed *closeness of mapping* in their work) in commonly-used programming languages (Green and Petre, 1996). The second aspect of directness is a feeling of *direct engagement*: 'the feeling that one is directly manipulating the objects of interest'. Nardi sees direct engagement as a critical element in spreadsheet languages' usability, due in part to the freedom from low-level programming minutiae in favor of task-specific operations (Nardi, 1993). The notion of aiming for directness as a programming language design goal has in recent years begun to influence other kinds of end-user programming languages and domain-specific languages as well.

Green *et al*.'s research into how the structure of a programming language or environment's characteristics relate to cognitive issues in programming provides useful insights into the difficulties and advantages of various language or environmental devices. Two of the characteristics studied, viscosity and feedback, are of particular relevance in the realm of spreadsheet languages. *Viscosity* is programmer effort required to change a program. There is research showing that programmers iteratively create their programs, making change after change throughout the entire process (Green and Petre, 1996). If a programming environment does not allow these changes to be easily inserted, the programmer must exert considerable extra effort devoted solely to the mechanics of change. For example, in traditional programming environments, to change a program and validate the correctness of the change, a programmer must enter the change using an editor in one step, recompile the program in another step, and re-run the program (re-entering the inputs) to test

the result. The traditional requirement that programmers manually switch among several tools and modes is an example of high viscosity, and spreadsheet systems eliminate much of this viscosity through the use of a unified, relatively modeless environment that guarantees an incrementally runnable program with immediate visual feedback after each change.

The aspect of feedback most relevant to spreadsheet language design is how different liveness levels affect people's abilities to program and debug. Green and others have pointed out both positive and negative aspects of feedback but, particularly for novice programmers, more positives than negatives have been reported. (In Green's work, liveness at levels 2 and above when applied to partially-completed programs are called *progressive evaluation*.) For example, in a study comparing the comprehension differences in debugging between novice and expert programmers (Gugerty and Olson, 1986), it was shown that self-evaluating their progress frequently (via frequent executions as a program evolves) was essential for novice programmers and that, while it was not essential for experts, the experts actually use execution of partially-completed programs even more frequently while debugging than novices do. Maximizing liveness reduces the amount of effort a programmer must exert to self-evaluate an unfinished program in this fashion. A more recent study evaluating how liveness affected programmers' ability to debug in Forms/3 also reported more positive outcomes than negatives associated with liveness (Wilcox *et al*., 1997; Cook *et al*., 1997).

The fourth feature that has been investigated extensively in spreadsheet languages is *hidden dependencies*. Hidden dependencies are dependencies that are not fully visible (explicit) in the program (Green and Petre, 1996). For example, in many traditional languages, side effects are possible, so named because they are not visible in a procedure or method call; thus they are hidden in that they are not present in the programmer's communication with the procedure/method. Hidden dependencies arise in many spreadsheet languages because formula dependencies are usually partially hidden. For example if A's formula references B, which in turn references C, which in turn references D, one can examine B's formula to see cells that directly affect it (C), but not to see that D transitively affects it or which cells B itself affects (A). Hidden dependencies are linked with bug presence and debugging time per bug in programming languages in general and spreadsheet languages in particular. To solve this problem, some spreadsheet languages have incorporated devices to make hidden dependencies explicit (Hendry, 1995; Hendry and Green, 1993; Yang *et al*., 1997). Forms/3 devices aimed at this problem that are demonstrated in this paper include arrows that can be toggled on and off to show dataflow and copy dependencies, and gray shading supplemented with legends to indicate copy dependencies.

Appendix B: Turing machine simulator

Figures B1 through B4 show a basic Turing machine simulator written in Forms/3. Given Forms/3's support for recursion, its Turing completeness is not surprising. However, the implementation given here does not use recursion; rather, its func-

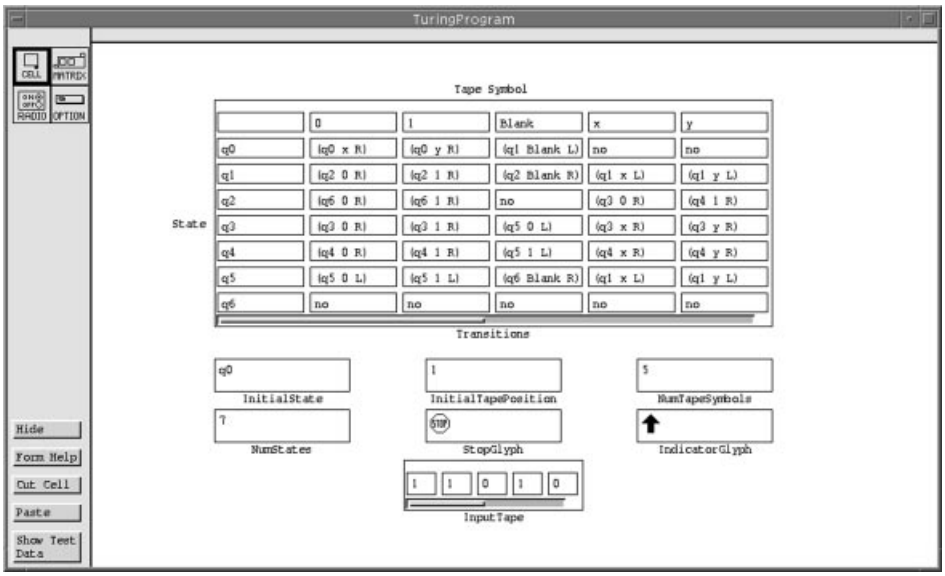


Fig. B1. Form TuringProgram allows a user to specify the Turing machine's properties. In the Transitions dynamic matrix, the first row is the index of tape symbols, and the first column is the index of states. The proper transition is found by accessing the dynamic matrix element found in the row containing the current state as its index, and in the column containing the current tape symbol. The transitions are expressed as triples, consisting of the next state, the symbol to write on the tape, and a direction to move. Every cell on this form is an 'input', that is, the user has entered its value directly via a constant formula, such as the formula '7' for cell NumStates.

tionality is achieved through spreadsheet formulas that specify groups of cells in dynamic grids over time.

Shown in figure B1 is Form TuringProgram, which allows specifying the properties of a Turing machine. This particular Turing machine performs a classic textbook example (e.g. Linz (1996)); it takes a string of binary digits as input and produces that string concatenated with itself. Form TuringCompute in figure B2 carries out the computations according to this specification, and form TuringAnimation in figure B3 provides animated output, such as that shown in figure B4.

The logic of the version of TuringProgram provided to solve this problem is as follows (state transitions are included in parentheses):

1. From the initial state (q0), traverse the original string from left to right, replacing each '0' with an 'x' and each '1' with a 'y' (when finished, transition to state q1).
2. Return to the left end of the string (and transition to state q2). Then replace the first letter with its corresponding numeral (and transition to state q3 or q4).
3. Move to the first blank to the right of the string and write the same numeral as written in step 2 (and transition to state q5).
4. If any letters remain on the tape, (transition to state q1 and) repeat from step 2. Otherwise, stop (transition to final state q6).

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

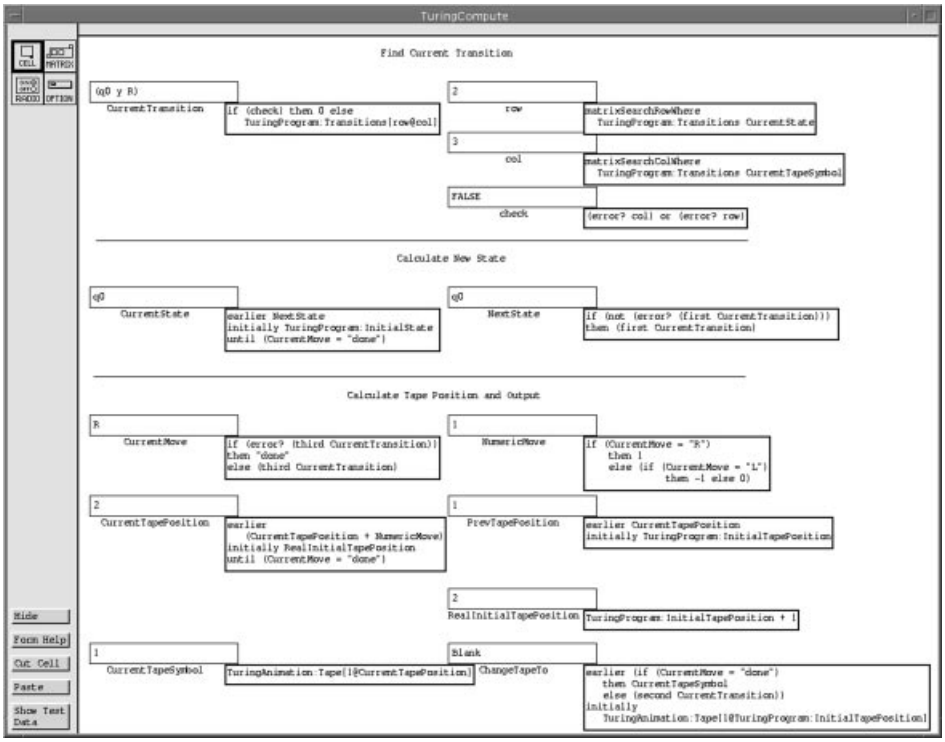


Fig. B2. TuringCompute performs the calculations specified by the Turing machine's TuringProgram (figure B1). RealInitialTapePosition is an implementation convenience, to abstract away the fact that the actual operation of the Turing machine places a 'Blank' in the first Tape position. Note the use of the else-less if expression in cell NextState: only at times at which the if test succeeds are new vt-tuples defined in NextState's temporal vector.

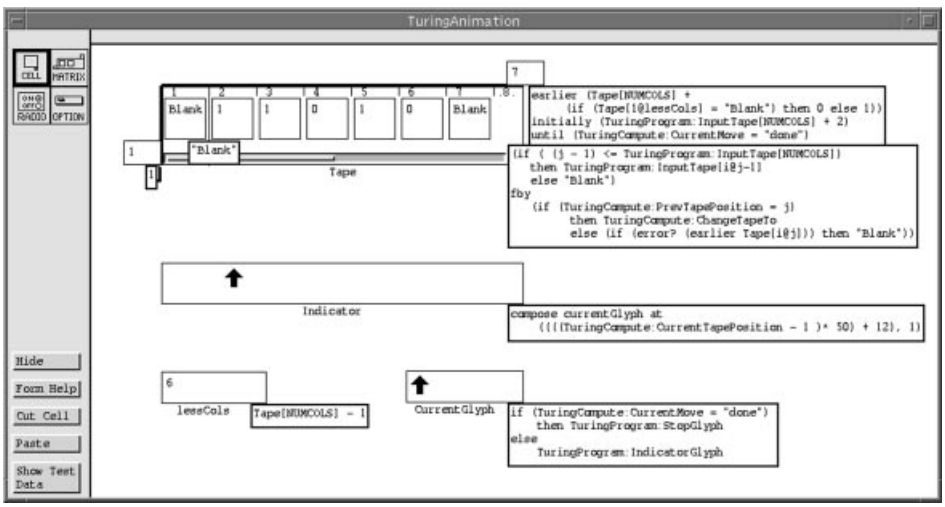


Fig. B3. TuringAnimation provides the animated output. This is the programmer's view. Cell Indicator has been moved farther down than normal to allow room for displaying the formulas. CurrentGlyph is the graphic to use in the animation.

https://doi.org/10.1017/S0956796800003828 Published online by Cambridge University Press

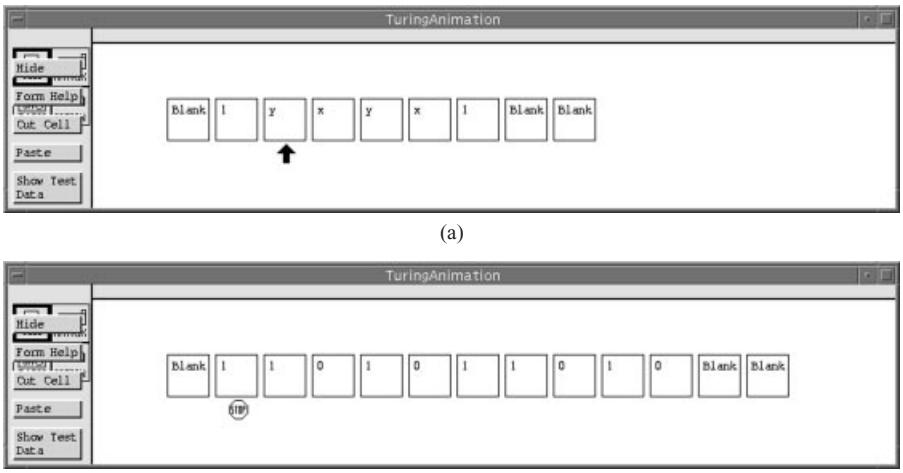


Fig. B4. (a) The animated output as seen by the user, partway through the program's execution. Only the features visible to the user are shown; cells containing implementation details have been hidden by the programmer, as have the formula tabs, borders, and labels; (b) the animation as it appears when the final state is reached.

References

Achten, P. and Plasmeijer, R. (1995) The ins and outs of clean I/O. *J. Functional Programming*, **5**(1), 81–110.

Agrawal, H., DeMillo, R. and Spafford, E. (1993) Debugging with dynamic slicing and backtracking. *Software – Practice and Experience*, **23**(6), 589–616.

Ambler, A. (1987) Forms: Expanding the visualness of sheet languages. *Workshop on Visual Languages*, Linkoping, Sweden.

Ambler, A. (1999) The Formulate visual programming language. *Dr. Dobb's Journal, August*, 21–28.

Ambler, A. and Broman, A. (1998) Formulate solution to the visual programming challenge. *J. Visual Languages and Computing*, **9**(2), 171–209.

Ambler, A. and Burnett, M. (1990) Visual forms of iteration that preserve single assignment. *J. Visual Languages and Computing*, **1**(2), 159–181.

Arya, K. (1989) Processes in a functional animation system. *Functional Programming Languages and Computer Architecture*, pp. 382–395.

Baker, H. (1991) NReversal of fortune – The thermodynamics of garbage collection. *1991 Int. Workshop on Memory Management*, St. Malo, France, 507–524.

Bharat, K. and Hudson, S. (1995) Supporting distributed, concurrent, one-way constraints in user interface applications. *ACM Symposium on User Interface Systems and Technology*, Pittsburgh, Pennsylvania, pp. 121–132.

Brayshaw, M. and Eisenstadt, M. (1991) A practical graphical tracer for Prolog. *Int. J. Man-Machine Studies*, **35**(5), 597–631.

Burnett, M., Agrawal, A. and van Zee, P. (2000) Exception handling in the spreadsheet paradigm. *IEEE Trans. Software Eng.*, October, pp. 923–942.

Burnett, M. and Ambler, A. (1994) Interactive visual data abstraction in a declarative visual programming language. *J. Visual Languages and Computing*, **5**(1), 29–60.

Burnett, M., Atwood, J. and Welch, Z. (1998) Implementing level 4 liveness in declarative visual programming languages. *1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Canada, pp. 126–133.

Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S. and van Zee, P. (1995) Scaling up visual programming languages. *Computer*, **28**(3), 45–54.

Burnett, M., Cao, N. and Atwood, J. (2000) Time in grid-oriented VPLs: Just another dimension? *IEEE Symposium on Visual Languages*, Seattle, WA, pp. 137–144.

Burnett, M. and Gottfried, H. (1998) Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Trans. Computer-Human Interaction*, **5**(1), 1–33.

Burnett, M., Sheretov, A. and Rothermel, G. (1999) Scaling up a 'What you see is what you test' methodology to testing spreadsheet grids. *1999 IEEE Symposium on Visual Languages*, Tokyo, Japan, pp. 30–37.

Carlson, P., Burnett, M. and Cadiz, J. J. (1996) A seamless integration of algorithm animation into a visual programming language. *Proc. Advanced Visual Interfaces '96*, ACM Press, Gubbio, Italy, pp. 194–202.

Carlsson, M. and Hallgren, T. (1993) FUDGETS – A graphical user interface in a lazy functional language. *ACM Conf. on Functional Programming and Computer Architecture*, pp. 321–330.

Chi, E., Riedl, J., Barry, P. and Konstan, J. (1998) Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applic*. July/August, 30–38.

Cook, C., Burnett, M. and Boom, D. (1997) A bug's eye view of immediate visual feedback in direct-manipulation programming systems. *Empirical Studies of Programmers: Seventh Workshop*, Washington, DC, ACM Press.

Cypher, A. (editor) (1993) *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.

Cypher, A. and Smith, D. (1995) KidSim: End user programming of simulations. *CHI '95: Human Factors in Computing Systems*, Denver, CO, May 7–11, pp. 27–34.

Djang, R. and Burnett, M. (1998) Similarity inheritance: A new model of inheritance for spreadsheet VPLs. *1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Canada, pp. 134–141.

Djang, R., Burnett, M. and Chen, R. (2000) Static type inference for a first-order declarative visual programming language with inheritance. *J. Visual Languages and Computing*, April, pp. 191–235.

Dix, A. (1987) Giving control back to the user. *Human-Computer Interaction – INTERACT '87*, Elsevier Science, pp. 377–382.

Du, W. and Wadge, W. (1990) A 3D spreadsheet based on intensional logic. *IEEE Software*, May, 78–89.

Elliott, C. and Hudak, P. (1997) Functional reactive animation. *ACM Int. Conf. on Functional Programming*, Amsterdam, Netherlands, June 9–11, pp. 263–273.

Finne, S. and Peyton Jones, S. (1996) Composing the user interface with Haggis. *Advanced Functional Programming: Second Interational School: Lecture Notes in Computer Science 1129*, Springer-Verlag, August 26–30, pp. 1–38.

Gottfried, H. and Burnett, M. (1997) Programming complex objects in spreadsheets: an empirical study comparing textual formula entry with direct manipulation and gestures. *Empirical Studies of Programmers: Seventh Workshop*, Washington, DC, ACM Press.

Green, T. and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, **7**(2), 131–174.

Gugerty, L. and Olson, G. (1986) Comprehension differences in debugging by skilled and

novice programmers. In: E. Soloway and S. Iyengar (editors), *Proc. Empirical Studies of Programmers* Ablex, Norwood, NJ, pp. 13–27.

Halbwachs, N. (1993) *Synchronous Programming of Reactive Systems.* Kluwer.

Hallgren, T. and Carlsson, M. (1995) Programming with fudgets. *Advanced Functional Programming: Lecture Notes in Computer Science 925*. Springer-Verlag.

Hendry, D. (1995) Display-based problems in spreadsheets: a critical incident and a design remedy. *1995 IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp. 284–290.

Hendry, D. and Green, T. (1993) CogMap: a visual description language for spreadsheets. *J. Visual Languages and Computing*, **4**(1), 35–54.

Hill, R. (1993) The rendezvous constraint maintenance system. *ACM Symposium on User Interface Software and Technology*, Atlanta, GA, pp. 225–233.

de Hoon, W., Rutten, L. and van Eekelen, M. (1995) Implementing a functional spreadsheet in CLEAN. *J. Functional Programming*, **5**(3), 383–414.

Hudak, P., Peyton Jones, S. and Wadler, P. (editors) (1992) Report on the programming language Haskell, a non-strict purely-functional programming language: Version 1.2. *ACM Sigplan Notices*, **27**(5), Ri–Rx, R1–R163.

Hudson, S. (1994) User interface specification using an enhanced spreadsheet model. *ACM Trans. Graphics*, **13**(4), 209–239.

Hughes, C. and Moshell, J. (1990) Action graphics: A spreadsheet-based language for animated simulation. In: T. Ichikawa, E. Jungert and R. Korfhage (editors), *Visual Languages and Applications.* Plenum, New York, NY, pp. 203–235.

Hughes, J. (1985) Lazy memo-functions. In: J.-P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 201*. Nancy, France, September 16–19, pp. 129–146. Springer-Verlag.

Hutchins, E., Hollan, J. and Norman, D. (1986) Direct manipulation interfaces. In: D. Norman and S. Draper (editors), *User Centered System Design: New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum, Hillsdale, NJ, pp. 87–124.

Kay, A. (1984) Computer software. *Scientific American*, **251**(3), 52–59.

Landin, P. J. (1965) A correspondence between ALGOL 60 and Church's lambda notation: Parts I and II. *Comm. ACM*, **8**(2, 3), 89–101, 158–165.

Launchbury, J. and Peyton Jones, S. (1994) Lazy functional state threads. *ACM Conf. on Programming Language Design and Implementation.*

Leopold, J. and Ambler, A. (1997) Keyboardless visual programming using voice, handwriting, and gesture. *1997 IEEE Symposium on Visual Languages*, Capri, Italy, pp. 28–35.

Levoy, M. (1994) Spreadsheet for images. *ACM Siggraph 94.* (Proceedings published as *Computer Graphics*, **28**(4), 139–146.)

Lewis, C. (1990) NoPumpG: Creating interactive graphics with spreadsheet machinery. In: E. Glinert (editor), *Visual Programming Environments: Paradigms and Systems.* IEEE Press, Los Alamitos, CA, pp. 526–546.

Lieberman, H. and Fry, C. (1995) Bridging the gulf between code and behavior in programming. *CHI '95: Human Factors in Computing Systems*, Denver, CO, pp. 480–486.

Lieberman, H. and Fry, C. (1997) ZStep 95: A reversible, animated source code stepper. In: J. Stasko, J. Domingue, M. Brown and B. Price (editors), *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA.

Linz, P. (1996) *An Introduction to Formal Languages and Automata*, (2nd edition), Heath and Co., Washington, DC.

Liu, C. and Orgun, M. (1996) Dealing with multiple granularity of time in temporal logic programming. *J. Symbolic Computation*, **22**, 699–720.

McCormick, B., DeFanti, T. and Brown, M. (editors) (1987) Visualization in scientific computing. *Computer Graphics*, **21**(6).

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*. MIT Press, Cambridge, MA.

Michie, D. (1968) 'Memo' functions and machine learning. *Nature*, **218**(5136), 19–22.

Miyashita, K., Matsuoka, S., Takahashi, S., Yonezawa, A. and Kamada, T. (1992) Declarative programming of graphical interfaces by visual examples. *ACM Symposium on User Interface Software and Technology*, Monterey, CA, pp. 107–116.

Miyashita, K., Matsuoka, S., Takahashi, S. and Yonezawa, A. (1994) Iterative generation of graphical user interfaces by multiple visual examples. *ACM Symposium on User Interface Software and Technology*, Marina del Rey, CA, pp. 85–94.

Moher, T. (1988) PROVIDE: A process visualization and debugging environment. *IEEE Trans. Software Eng.*, **14**(6).

Myers, B. (1991) Graphical techniques in a spreadsheet for specifying user interfaces. *ACM Conf. on Human Factors in Computing Systems*, New Orleans, LA, April 28–May 2, pp. 243–249.

Myers, B., Guise, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A. and Marchal, P. (1990) Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer*, **23**(11), 71–85.

Myers, B., Miller, R., McDaniel, R. and Ferrency, A. (1996) Easily adding animations to interfaces using constraints. *ACM Symposium on User Interface Software and Technology*, Seattle, WA, pp. 119–128.

Nardi, B. (1993) *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA.

Orgun, M. and Wadge, W. (1992) Theory and practice of temporal logic programming. In: L. Fariñas del Cerro and M. Penttonen (editors), *Intensional Logics for Programming*. Oxford University Press, pp. 23–50.

Pandey, R. and Burnett, M. (1993) Is it easier to write matrix manipulation programs visually or textually? An empirical study. *1993 IEEE Symposium on Visual Languages*, Bergen, Norway, pp. 344–351.

Perry, N. (1989) I/O and inter-language calling for functional languages. *Proceedings 9th International Conference of the Chilean Computer Society and 15th Latin American Conference on Informatics*, Chile.

Peyton Jones, S. L. and Wadler, P. (1993) Imperative functional programming. *1993 ACM Symposium on the Principles of Programming Languages*, Charleston, SC, pp. 71–84.

Peyton Jones, S., Gordon, A. and Finne, S. (1996) Concurrent Haskell. *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL.

Reichwein, J., Rothermel, G. and Burnett, M. (1999) Slicing spreadsheets: an integrated methodology for spreadsheet testing and debugging. *Conference on Domain Specific Languages*, Austin, Texas, pp. 25–38.

Repenning, A. and Ambach, J. (1996) Tactile programming: a unified manipulation paradigm supporting program comprehension, composition and sharing. *1996 IEEE Symposium on Visual Languages*, Boulder, CO, pp. 102–109.

Rothermel, G., Li, L. and Burnett, M. (1997) Testing strategies for form-based visual programs. *International Symposium on Software Reliability Engineering* (*ISSRE '97*), Albuquerque, NM, pp. 96–107.

Rothermel, G., Li, L., DuPuis, C. and Burnett, M. (1998) What you see is what you test: a methodology for testing form-based visual programs. *International Conference on Software Engineering* (*ICSE '98*), Kyoto, Japan, pp. 198–207.

Shneiderman, S. (1983) Direct manipulation: a step beyond programming languages. *Computer*, **16**(8), 57–69.

Silberschatz, A. and Galvin, P. (1998) *Operating System Concepts:* (5th edition), Addison-Wesley, Reading, MA.

Smedley, T., Cox, P. and Byrne, S. (1996) Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. *Advanced Visual Interfaces '96*, Gubbio, Italy, pp. 148–155.

Stoye, W. (1986) Message-based functional operating systems. *Science of Computer Programming*, **6**(3), 291–311.

Tanimoto, S. (1990) VIVA: A visual language for image processing. *J. Visual Languages and Computing*, **2**(2), 127–139.

Tolmach, A. and Appel, A. (1991) Debuggable concurrency extensions for Standard ML. *1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, pp. 120–131.

Tolmach, A. and Appel, A. (1993) A debugger for Standard ML. *J. Functional Programming*, **1**(1).

Vander Zanden, B. and Myers, B. (1995) Demonstrational and constraint-based technologies for pictorially specifying application objects and behaviors. *ACM Trans. Computer-Human Interaction*, **2**(4), 308–356.

Vander Zanden, B. and Venckus, S. (1996) An empirical study of constraint usage in graphical applications. *ACM Symposium on User Interface Software and Technology*, Seattle, WA, November 6–8 1996, pp. 137–146.

Viehstaedt, G. and Ambler, A. (1992) Visual representation and manipulation of matrices. *J. Visual Languages and Computing*, **3**(3), 273–298.

Wadge, W. and Ashcroft, E. (1985) *Lucid, the Dataflow Programming Language*. Academic Press. London.

Wadler, P. (1987) List comprehensions. In: S. L. Peyton Jones (editor), *The Implementation of Functional Programming Languages*. Prentice Hall.

Wadler, P. (1990) Linear types can change the world! In: M. Broy and C. Jones (editors) *Programming Concepts and Methods*. North Holland, Amsterdam.

Wadler, P. (1997) How to declare an imperative. *ACM Computing Surveys*, **29**(3), 240–263.

Wang, G. and Ambler, A. (1996) Solving display-based problems. *1996 IEEE Symposium on Visual Languages*, Boulder, CO, pp. 122–129.

Wilcox, E., Atwood, J., Burnett, M., Cadiz, J. J. and Cook, C. (1997) Does continuous visual feedback aid debugging in direct-manipulation programming systems? *ACM Conference on Human Factors in Computing Systems*, Atlanta, GA, pp. 258–265.

Wilde, N. and Lewis, C. (1990) Spreadsheet-based interactive graphics: from prototype to tool. *ACM Conference on Human Factors in Computing Systems*, Seattle, WA, pp. 153–159.

Wray, S. and Fairbairn, J. (1989) Non-strict languages – programming and implementation. *Computer Journal*, **32**(2), 142–151.

Yang, S., Burnett, M., DeKoven, E. and Zloof, M. (1997) Representation design benchmarks: A design-time aid for VPL navigable static representations. *J. Visual Languages and Computing*, **8**(5/6), 563–599.