A VISUAL PROGRAMMING LANGUAGE FOR VISUALIZATION OF SCIENTIFIC DATA

BY

DANIEL DAVID HILS

B.S., John Carroll University, 1985
M.S., University of Illinois, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

A VISUAL PROGRAMMING LANGUAGE FOR VISUALIZATION OF SCIENTIFIC DATA

Daniel D. Hils, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1992
R. Johnson, Advisor

Today scientists are confronted with the problem of understanding and analyzing large masses of numeric scientific data. One solution to this problem is scientific visualization: converting the numeric data into pictures more readily understood by scientists. This thesis presents DataVis, a visual programming language that is designed to be used by scientists for the visualization of scientific data. Since DataVis is a visual language, DataVis programs and functions are primarily diagrams, rather than text. The language is based on the data flow computational model. DataVis provides a library of predefined programs and functions. Scientists transform data into pictures by connecting data icons to program icons. The program icons can be customized, extended, and composed using the same "plumbing" metaphor, and new program icons can be defined using the same metaphor. Thus, DataVis is both a set of finished programs for visualizing scientific data and an easy-to-use toolset for constructing more. It is a complete programming language, so there is no limit to the kinds of visualization tools that it can construct, but its programming language is built on the same metaphor as the user interface of its applications, so it is easy for non-programmers to learn.

ACKNOWLEDGEMENTS

I thank my thesis advisor, Prof. Ralph Johnson, for being a font of creative research ideas, for his guidance throughout this research, and for his help in revising and improving this thesis.

I thank the other members of my committee, Prof. Eric Golin, Prof. Donald Hearn, Prof. Simon Kaplan, and Prof. Robert Wilhelmson,

Lastly, I thank my parents for their years of unwavering support.

PREFACE

"Another such victory over the Romans, and we are undone."

Plutarch, *Lives*, Pyrrhus.

"If you can dream—and not make dreams your master;
If you can fill the unforgiving minute
With sixty seconds' worth of distance run,"

Rudyard Kipling, "If"

"Qui audet vincit."

Anonymous

"I shall be afield before you ...have breakfasted. ...In short, I shall astonish you all."

Thomas Hardy,
Far From The Madding Crowd

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The last ten years have witnessed a tremendous growth of interest in a new class of programming languages: visual programming languages. The essential characteristic of visual programming languages is that programs written in them are composed primarily of diagrams, rather than text. Visual languages often contain text, but their main emphasis is on pictures.

1.1 Strengths Of Visual Programming Languages

Because visual programming languages are based on pictures, they have a number of advantages over traditional textual programming languages. Raeder [Raed85] has described many of these advantages. A summary of them follows.

The human mind is very visually oriented, and so people can often get information more quickly by finding graphical relationships in complex pictures than by reading text. We have random access to any part of a picture and to detailed and overall views, whereas text access is sequential. Text is a one-dimensional stream of words, but pictures are two-dimensional, allowing us to perceive information through shape, size, color, texture, direction, and distance. (Some of these characteristics can be used in one-dimensional, textual languages, but they are much more effective in two-dimensional, visual languages.) A program involves abstract ideas (such as loops, procedures, parameters, data types, and data flow); using real-world objects in pictures can make these abstract ideas easier to think about.

Glinert [Glin84] and Myers [Myer90] give two more advantages of using pictures in programs. Glinert states that pictures are easily learned, remembered, and retrieved as single units of information. This is useful because our minds can only work with a limited number of such units at a time. Myers suggests that using pictures for programs allows syntax to be de-emphasized and permits a higher level of abstraction. This may make programming easier. Pictures or graphics also aid in debugging by making it possible to present much more information about the state of the program, such as current variables or data structures, than can be presented with a solely textual program.

1.2 Weaknesses of Visual Programming Languages

As can be seen from the advantages of visual programming languages, such languages have potential for being very useful. However, visual languages have not fulfilled their potential. Most programmers do not use visual languages; visual languages have not appealed to expert programmers; nor has anyone successfully created a visual C—that is, a general-purpose visual programming language.

These failures may be due in part to several weaknesses of visual languages. First, visual languages consume a great deal of screen space. (This is called the *screen space* problem.) Second, since visual languages are a more recent development than textual languages, there are fewer tools for creating visual languages than there are for their textual counterparts. Editors, debuggers, compilers, and interpreters for visual languages often must be built from scratch, at a considerable cost in time and effort. To give one example, there is no widely accepted general-purpose editor for visual programs, no visual counterpart to emacs or vi. Third, many existing visual languages do not possess enough powerful features, such as procedural abstraction (condensing a whole section of a program down into one icon), type checking, and higher-order functions. Thus, some existing visual languages lack the power needed to deal with real application domains. Furthermore, some of these missing features are sorely needed to solve other problems besides "lack of power in the language". For example, while procedural abstraction makes a language more powerful, it *also* alleviates the first weakness of visual languages (the screen space problem) by making visual programs more compact.

It is unlikely that visual languages will replace their textual counterparts for expert programmers. However, experience with past languages has shown that visual languages are most successful: 1) when they are aimed at novice programmers, or at nonprogrammers, such as scientists; and 2) when they are aimed at a particular application domain, such as gathering data from laboratory instruments.

For example, LabVIEW [Sant90], a commercial product, is geared toward the specific application domain of collection of data from laboratory instruments, and has been successful. Also, Cantata [Rasu90] focuses on image processing and digital signal processing, and is used by a sizable community of DSP/image-processing researchers as an aid to their work. Lastly, although AVS [Upso89] is just a visual *language*, not a visual *programming* language, it is geared toward visualization of scientific data. AVS is a commercial product, and has been successful.

Visual programming languages are helpful to novices and non-expert programmers like scientists because visual languages are easier to learn than textual languages. This is because visual languages allow the direct manipulation of program objects (e.g. iteration boxes, icon boxes, links, etc.), and because they are better suited to program visualization than textual languages are. Specifically, by "program visualization" I mean that a visual language can convey more information to the user on the state of a program written in the language than a textual language can. I also refer primarily to giving the user dynamic, changing information at runtime, although static program visualization done at program-construction time by visual languages is also possible (the visual language described in this thesis does it, via coloring of links to show data types) and useful. Visual languages are more suitable for program visualization than textual languages are because a visual language can use a much broader, richer, graphic vocabulary to communicate information to the user. This vocabulary includes such elements as color, images/bitmaps, shape, line thickness, and line style.

1.3 Visual Programming For Scientific Visualization

This thesis applies visual programming languages to a particular application domain: scientific visualization. The central problem in visualization of scientific data (also called scientific visualization) [Wilh90] [McCo87] [Niel89] is that scientists today must deal with massive amounts of data. There are two sources of this data. The first source is real-world observations, from orbiting satellites, interplanetary spacecraft, ocean-monitoring instrument arrays, and medical scanners. A second source of data is the mathematical models used to simulate real-world phenomena on supercomputers. Whatever the source of the data may be, scientists must be able to comprehend and analyze the data. Understanding data is difficult when it is only available as huge masses of numbers. Thus, a focus of scientific visualization is to convert these masses of numeric data into pictures that are more readily understood by scientists.

Applying visual programming languages to the application domain of scientific visualization is an important research topic, for two reasons. First, no one has created a data flow visual programming language for scientific visualization before. This thesis is a case study for inventing such a visual programming language for a particular application domain. Second, the field of scientific visualization needs tools to help scientists visualize their data. A visual programming language can be such a tool.

1.4 Existing Visualization Systems

One of the reasons that scientific visualization is complicated is that scientific data comes in several different types, and each one can be converted into many kinds of pictures, which are also called graphic displays. (Other terms for a graphic display are *image* or *bitmap*. Throughout this thesis, the terms *graphic display*, *image*, and *bitmap* are used interchangeably.) These data types include scalars (i.e., reals and integers), vectors, tensors, fields, points and images. A field is a two- or three-dimensional grid of one of the simpler data types, such as scalars or vectors. Scalar quantities produce a scalar field; vector quantities produce a vector field. For example, quantities in nature such as temperature, density, pressure, and energy are scalar fields; quantities such as velocity, magnetic fields, and electric fields are vector fields. Other quantities, such as stress and strain, can be represented by tensors. Also, time, while not itself a data type, can add another dimension to scientific data.

There is a variety of graphic displays for scientific data. Two scientific visualization programs from the National Center for Supercomputing Applications, NCSA Image [NCSA89a] and NCSA DataScope [NCSA89b], are geared toward visualizing 2D scalar fields. They make good use of color and generate almost all the common graphic displays conceivable for a 2D scalar field. Graphic displays which can be generated from 2D scalar fields include: 2D color images, 2D XY graphs (the user draws a line across a 2D color image, and the pixel values along the line are graphed against the length of the line), 2D contour plots (see Figure 1.1), shaded data plots (like color images, but with eight gray level substituted for colors), dithered

Figure 1.1: NCSA Image contour plot [NCSA89a].

plots (like shaded data plots, but with many gray levels), and 3D plots (see Figure 1.2). These graphic displays are hard-wired into the programs, so each program can be thought of as several functions which map 2D scalar field to a graphic display.

NCSA DataScope presents a user with several windows. A numeric data window shows a spreadsheet representation of a 2D scalar field; another window shows the 2D color image generated from the field. (See Figure 1.3.) Selecting numbers in the numeric data window selects the corresponding pixel or pixels in the image window, and vice versa. A number of predefined mathematical functions can be executed on the numeric data. Users desiring new functions for mathematics or for other purposes must write their own in C.

A 3D scalar field can be depicted by isosurfaces and volume rendering. An isosurface is a 3D surface of equal value for some scalar quantity. Volume rendering creates an image which gives information about the interior of a 3D scalar field, not just about the field's outer edges. Volume rendering is performed by casting rays through a 3D scalar field to directly create an image without making an intermediate geometric representation first. Intensities are then accumulated along the ray path to give an intensity for a pixel; the process is repeated for each pixel displayed.

Another area in which scientists are interested is the visualization of fluid flow. This area often involves 2D vector fields, as well as 3D vector fields. Particle advection or flow ribbons may be used to show a 2D or 3D vector field [Shir89]. Particle advection displays the flow by

Figure 1.2: NCSA Image 3D plot [NCSA89a].

Figure 1.3: NCSA DataScope: spreadsheet and image windows [NCSA89b].

simulating the effect of the velocity field (a 2D or 3D vector field) on particles injected into the flow. The fluid may be a liquid or a gas. If the fluid being modeled is a gas, the particles mimic the use of smoke in laboratory experiments; if the fluid is a liquid, they mimic dye. Streamers may be attached to some particles in order to see their entire paths. Flow ribbons are similar to particle advection with attached streamers except that ribbons are wider than streamers and are usually shown closer up. Flow ribbons give a scientist information about twisting in a vector field, which is information that may not be provided by streamers.

Although particle advection is an important graphic display for 2D and 3D vector fields, three other related displays that portray 2D and 3D vector fields are pathlines, streamlines, and streaklines [Sale89]. A pathline is the path a particle takes as it moves through the vector field as a function of time. (Typically, the vector field itself changes with time.) A streamline is a curve that is tangent to the instantaneous (time-frozen) vector field. (That is, it is the path a particle would follow through the time-frozen field.) A streakline is the position of all particles in the vector field that have passed through a fixed point during a given time interval. Salem [Sale89] discusses these three graphic displays in connection with an application program he has developed for visualizing fluid flow.

Particle advection where the particles have streamers of infinite length attached produces a pathline. For a constant vector field, pathlines, streamlines, and streaklines are equivalent. For a time-varying field, the three are not equivalent, but a streamline can still be obtained by "freezing" the field (i.e., using the field at only one time value) and doing particle advection with streamers attached. Also, continuous injection of particles at a point gives a collection of streaklines. Salem mentions various initial placements and numbers of particles that are possible.

Although Salem's program for visualization of fluid flow provides some useful displays, his program does not constitute a programming language, does not use data flow, does not allow a user to hook components together, and provides no mechanism for a user to extend the set of graphic displays provided. Neither Image nor DataScope is based on data flow. Image and DataScope do not allow a user to connect components to produce a certain graphic display. Rather, the displays from which the user may choose are fixed for each program.

Several lessons can be drawn from the visualization systems just described. First, existing visualization systems have good features, but are inflexible. The graphic displays in systems such as NCSA Image and DataScope are hard-wired into the program. This restricts scientists to a limited set of options for visualizing data. Even more complex systems, such as the ones reviewed in Chapter 2, which give scientists more flexibility by allowing users to connect components together, do not provide the power of a full high-level programming language. Such a language is needed to add new kinds of data, new functions or modules, and new graphic displays to those already provided.

A second lesson is that there is a wide range of graphic displays (and the underlying algorithms necessary to generate the displays) in existing visualization systems. This is apparent even though this brief overview of visualization systems covered only a portion of the graphic displays desired by scientists. Because of the wide variety of displays and algorithms needed for visualization, there is a need for the capability to do programming in visualization systems.

Having argued that a programming language needs to be a part of a software system for visualization, it is worth considering why the programming language should be visual. In order to do this, it is helpful to consider the needs and wishes of scientists, the users of visualization systems.

Recall that a focus of scientific visualization is to convert large amounts of numeric data into pictures, which can be more easily understood by scientists. I just argued that programming is necessary. However, most scientists would rather study science than write programs that create basic software tools for visualization. The ultimate goal of these scientists is to gain a better understanding of nature. For them, the pictures are a means to that end. Thus, the scientists are interested in the pictures so that they may analyze their data, discover previously unknown trends or phenomena which are revealed by the pictures, and possibly modify their mathematical model of their area of study. Most scientists would prefer not to spend their time programming or becoming involved in low-level syntactic details of programming.

Because of the nature of scientific visualization and because of the needs and wishes of scientists, visual programming languages are well-suited to the application area of scientific visualization. The output of a visualization effort is pictures, so it is quite plausible to also construct the programs that do the visualization from pictures. Choosing pictures as a uniform representation for programs and their output permits a consistent user interface to be presented and makes the visualization process clearer to scientists. Also, visual programming languages are helpful when users wish to forget that they are programming.

The visualization process can be divided into two phases. The first phase is to produce data. This may be done in two ways: by running a mathematical model to simulate the area of interest to the scientist, or by gathering data (from a satellite, for example). This first phase of visualization often requires much numerical computation and does not involve graphics. This phase can be done with a visual or textual language, but it is not the focus of my thesis.

This research concentrates on the second phase of visualization. The second phase takes the large amount of numeric data generated by the simulation during the first phase, and converts numbers to pictures under the direction of the scientist. Visual programming works well here, since the work being done involves graphics and since most of the users (scientists) wish to avoid the programming details of this work.

In summary, what is needed to help scientists do visualization is a high-level visual programming language. This language can be a tool to relieve them of the burden of paying attention to low-level computing issues and syntactic details. This thesis describes such a language. This language is a visual programming language based on data flow for scientific visualization. It is both more powerful and more flexible than the software systems for visualization that were described in this chapter. Chapter 2 discusses related research in visual programming languages and in sophisticated software systems for scientific visualization. The language described in Chapter 3 not only improves on the visualization systems covered in Chapter 1, but also addresses some of the weaknesses of the advanced visualization systems discussed in Chapter 2.

Chapter 2

Related Research on Visual Programming Languages and Scientific Visualization Systems

2.1 Visual Programming Languages

Many visual programming languages have been proposed and implemented. Glinert [Glin84], Raeder [Raed85], and Myers [Myer90] survey these languages. Here I will focus on visual programming languages that are based on a data flow computational model. For a more complete survey of these languages, see [Hils92]. I use the term *data flow* to mean a computational model for a visual programming language, rather than the architecture of the machine on which the visual language runs.

I now present a brief overview of the data flow computational model, along with a survey of three visual programming languages that use it. For each language, I discuss what form or variant of the data flow model it uses, its intended application area, its features, and its strengths and weaknesses.

2.1.1 Overview of Data Flow Model

The idea of using a data flow graph to represent a program has existed for some time [Denn75]. However, the widespread use of the data flow computational model in visual programming languages is more recent [Ager82]. The central concept of the data flow model is that a program can be represented by a directed graph in which nodes represent functions and arcs represent the flow of data between functions [Davi82]. Arcs entering a node represent input data to a function; arcs exiting represent output data—that is, the function's results. Units of data which flow on the arcs are called tokens. Throughout this thesis, the terms *arc*, *arrow*, *wire*, *link*, *line*, and *arrow* are used interchangeably, as are *procedure*, *function*, *icon*, and *box*.

Data flow visual programming languages may be classified according to their use of various features. The following features are from Hils [Hils92].

2.1.1.1 Pure Data Flow Model

The *pure* data flow model is the data flow model with no added control flow constructs such as iteration, sequential-execution constructs, or CASE statements. In the pure data flow model, the order in which functions or nodes are to execute is not specified. Rather, when all of a node's inputs are available, the node fires. This corresponds to executing the function. The resulting data is then put on the output arcs, where it flows downstream to other functions. While all data flow visual programming languages are based on the pure data flow model, the following features can be used to augment the pure data flow model. (The first feature, a box-line representation, does not so much augment the model as provide one possible visual appearance for it.)

2.1.1.2 Box-Line Representation

Most data flow visual languages use boxes to represent functions and lines to represent the flow of data between functions. An advantage of choosing this box-line representation for a language is that it allows the user to easily insert viewing monitors at different spots to inspect the data at run time. An alternative is to use juxtaposition of boxes, rather than a line connecting boxes, to show that a function is being executed.

2.1.1.3 Iteration

Many data flow visual languages provide iteration. Possible iteration constructs include: cycles in the data flow graph; sequential ports [Kimu86a], which recycle a single changing value through repeated executions of an associated box; parallel ports, which split up a collection or list into its elements, execute an associated box once for each element, and gather up the results into a collection or list; control flow constructs such as FOR, WHILE, and REPEAT loops; and predefined functions that apply an operation to all elements of a list.

2.1.1.4 Procedural Abstraction

Procedural abstraction is available in some data flow visual languages—for example, in HI-VISUAL [Hira86] and in Show and Tell [Kimu86a]. Procedural abstraction means that an entire graph may be considered a procedure and may be condensed into a single node. Arcs going to and from the node may then be considered arguments to the procedure. Procedural abstraction is important for a number of reasons. First, it saves screen space, making a visual program more compact. Second, it allows the programmer to view the program at a higher level, without extraneous details. Third, procedural abstraction makes a language extensible. Fourth, it aids code reuse. Fifth, procedural abstraction allows a person who is unfamiliar with a certain program to understand the program one piece (i.e., one procedure) at a time.

2.1.1.5 Selector and Distributor

Two constructs that are part of the data flow model are the selector and the distributor (Figure 2.1). In its simplest form, a selector accepts a true or false data token to decide which of two inputs should be propagated to its output. A distributor uses a true or false token to pick an output arc to put its data on. Data flow visual languages rarely have either a selector or distributor in the exact form of Figure 2.1, but languages may have similar constructs that accomplish the same task.

2.1.1.6 Sequential-execution Construct

A difficulty with the pure data flow model is that it does not allow a user to specify that actions should be performed in a sequence: “first function A, then B, then C”. Such a sequence of actions is useful for dealing with input/output devices and for interaction with the user. Rather, in the pure data flow model, any of A, B, or C could be executed as soon as its input data was available. (Admittedly, one could draw a link from A to B and B to C to force an execution order of “first A, then B, then C”. However, such a link would be spurious, since it would not really carry any data.) Languages may introduce a sequential-execution construct

Figure 2.1: Selector and distributor functions ([Davi82]).

to allow functions to be executed in a specific order; LabVIEW [Sant90] is an example of such a language.

2.1.1.7 Type Checking

Some data flow visual languages (for example, Fabrik [Inga88]) indicate the type of data flowing over arcs, and perform type checking. Type checking is usually done at program construction time by not allowing the user to connect functions with types that do not match. This prevents run-time errors which otherwise would be caused by type mismatches.

2.1.1.8 Execution Modes

There are two possible modes of execution for a data flow program: data-driven and demand-driven. For data-driven execution, when a function node executes, it receives data from upstream nodes on its input arcs, and it then sends data to nodes downstream on its output arcs. (Upstream nodes are found by following input arcs from the node backwards; downstream nodes are found by following output arcs forward out of the node.)

With data-driven execution, a node fires as soon as its inputs are available, and all data flows downstream. A weakness of data-driven execution is that some nodes may fire needlessly, even though the final output of the program does not make use of their computations [Rasu90]. A language designer can cure this difficulty by using demand-driven execution.

For demand-driven execution, execution of a node happens when downstream nodes request data from a node's output arcs. The node may request data from upstream nodes over its input arcs; if it does so, it then waits for these nodes to send it data. It then sends data over its output arcs to downstream nodes.

With demand-driven execution, a node fires only when downstream nodes request data from it. All data still flows downstream, but requests flow upstream. These requests slow down a data flow program; thus, demand-driven execution is used less often than data-driven execution. In fact, all the visual programming languages that I have reviewed [Hils92], except Cantata [Rasu90] [Will90] and VPL [Lau91], use data-driven execution.

2.1.1.9 Level of Liveness

Tanimoto describes a four-level scale of liveness for visual languages [Tani90a]. The first, "informative" level uses a visual program representation for documentation or education only. A flowchart is at this level. The second, "significant" level uses the visual representation as the program, where the user must give a command for the program to be executed. LabVIEW [Kodo91] is at the second level. The third, "responsive" level includes systems that execute automatically whenever the user edits the program or its data, without an EXECUTE command being given. HI-VISUAL [Hira89] is a system at this level. The fourth, "live" level includes systems that continually update the display to show computation results. VIVA [Tani90b] is at the fourth level.

2.2 Survey of Three Visual Programming Languages Based On Data Flow

Over a dozen visual programming languages based on data flow have been implemented. Most of these languages are geared toward a specific application area: Hookup [Levi86] is for music, Fabrik [Inga88] [Ludo88] and InterCONS [Smit88] help construct user interfaces, VIVA [Tani90a] [Tani90b], Cantata [Rasu90] [Will90] [Rasu91], and VPL [Lau91] are for image processing; VIPEX [Haars88] [Haars90] deals with science; ConMan [Haeb88] deals with graphics; *viz* [Holt90] [Holt91], the Visual Toolset [Borg90a] [Borg90b], and PROGRAPH [Matw85] [Cox89] are general-purpose programming languages, and ESTL [Najo90] adds types and higher-order functions to Show and Tell. I will briefly survey three other data flow visual languages here: HI-VISUAL, LabVIEW, and Show and Tell.

I discuss HI-VISUAL and LabVIEW to show common features of data flow visual programming languages and to give some idea of the amount of research in visual languages.

2.2.1 HI-VISUAL

HI-VISUAL is a visual data flow language that emphasizes icons [Yosh86] [Hira86] [Hira87] [Hira88] [Hira89] [Hira90] [Hira91]. Its application domain was originally image processing, but

Figure 2.2: A complete HI-VISUAL program ([Hira86]).

now has shifted to work done in an office environment: simple accounting, word processing, report generation, data storage and retrieval.

In HI-VISUAL, icons represent functions/procedures. The flow of data is indicated by arrows. Icons represent functions, but also represent the input and output data for a procedure. Execution is immediate in the sense that as soon as a function is added to the program (and given input data, if necessary), the function executes and produces an icon representing its output data. Figure 2.2 shows a HI-VISUAL program from an early version of the language. HI-VISUAL provides a library of icons that are predefined functions.

HI-VISUAL provides procedural abstraction: one icon can represent an entire subprogram. Figure 2.3 shows the subprogram behind the QUANTIZE icon in Figure 2.2.

Although HI-VISUAL is a visual language, it allows textual labels at the bottom of icons to help identify them. HI-VISUAL provides two kinds of iteration. The first kind is a pseudo-

Figure 2.3: A lower-level HI-VISUAL function behind QUANTIZE in Figure 2.2 ([Hira86]).

FOR-loop that allows the programmer to specify that, for example, certain icons should be executed four times, but that does not allow access to the loop control variable. The second kind of iteration is a REPEAT-UNTIL loop.

HI-VISUAL has a switch statement available that behaves like a CASE statement, but controls the flow of data rather than the flow of control. Depending on the value of a boolean expression, data coming into the switch is routed to one of two sets of outgoing links. There may also be multiple incoming links to the switch. This switch is akin to the distributor mentioned earlier in section 2.1.1.

A more recent version of HI-VISUAL ([Hira88]) uses icons only for data objects, not for functions. Also, the approach to the system has become object-oriented. Since data objects are considered to "own" operations (in accordance with the object-oriented paradigm), icons representing functions are no longer shown.

Links are now used only for indicating that output data is coming from a function execution. Such a function execution happens when two data icons are superimposed. A data icon's behavior is not fixed, but depends on what icon it overlaps, or on what icon overlaps it. Each icon is now considered an object, an instance of a class. Through class membership, an icon inherits class variables, instance variables, a set of acceptable messages, and a set of transmittable messages. Acceptable messages are messages that an icon can receive and execute (that is, functions the icon can perform) and transmittable messages are messages the icon can send.

A database of rules is kept. When two icons are overlapped by the programmer, the rules determine which icon is the passive object (to be checked for acceptable messages) and which icon is the active object (to be checked for transmittable messages). If there is not an immediate match between a transmittable message and an acceptable message, a search up the inheritance hierarchy (that is, into superclasses) is done.

Figure 2.4 shows a recent HI-VISUAL program. When the programmer superimposes the calculator over the sales book at the top left, a new sales book is produced. The drawing kit can then be applied to it to produce a graph paper, and so on.

The decision to reduce the use of arrows and not to show icons for functions achieves a more compact on-screen visual representation for HI-VISUAL programs, and frees a user of HI-VISUAL from mentally tracing arrows across the screen to understand a HI-VISUAL program. However, the reduced use of arrows and the absence of icons for functions makes HI-VISUAL

Figure 2.4: A more recent HI-VISUAL program ([Hira88]).

Figure 2.5: The LabVIEW Scale Number virtual instrument. The left half of this figure contains the front panel for this instrument; it contains two input variables, or controls, and one output control. The right half is the block diagram ([Vose86]).

programs less clear than they were before. The reason is that there is no longer an explicit visual reminder of which function is being called in a certain part of the HI-VISUAL program.

HI-VISUAL has a rather broad application area (office work), which it handles well. It is a fairly powerful language, but it lacks type checking and higher-order functions. (Chapter 3 will describe higher-order functions.)

2.2.2 LabVIEW

LabVIEW is a high-level language for use in data collection from lab instruments and for use in data analysis [Kodo91] [Sant90] [Nati87] [Vose86]. It is a commercial product. Program modules (procedures) are called virtual instruments. Each instrument has two parts: a front panel on the left and a block diagram on the right (Figure 2.5). The front panel is the user interface to the procedure. It contains input and output data (numbers) for the procedure, as well as sliders, knobs, strip charts, meters, and x-y graphs. Its purpose is to graphically display input to and output from the procedure. The block diagram is the data flow visual program for the procedure.

Figure 2.6: Control flow structures in LabVIEW ([Vose86]).

LabVIEW provides a broad selection of predefined function boxes. These functions include simple arithmetic operations, trigonometric functions, string manipulation routines, statistical analyses, signal processing, matrix operations, and curve fitting. Programmers can also define their own functions.

Ports are present on boxes in LabVIEW. Ports are subareas of a box. Only one wire (link) may be connected to a port. So the number, size, and configuration of the ports has to do with the number of inputs and outputs that a programmer expects a function will need. The top left corner of the user interface panel in Figure 2.5 shows one possible configuration of ports: two input ports on the left side of the function's icon, and one output port on the right side.

LabVIEW introduces various boxes representing control flow structures into the pure data flow computational model (Figure 2.6). These boxes include a FOR loop, a WHILE loop, a CASE statement, and a sequence structure. Despite the WHILE loop's name, it is really a REPEAT...UNTIL loop: its contents are always executed at least once. The sequence structure forces sequential execution of a series of data flow diagrams; such explicit indication of "do diagram A, then B, then C" is normally lacking in data flow programs. LABVIEW's data types include boolean values and arrays of booleans, real numbers and arrays of reals, strings and arrays of strings, and structures (similar to structures in C).

LabVIEW is one of the more powerful visual data flow languages. Its power is due to the fact that it provides iteration and procedural abstraction, and to its large library of predefined functions. It handles its application domain (collection and processing of data from scientific lab instruments) well. However, it is specialized for laboratory instrument control, and it is not designed for scientific data visualization. (That is, it has library functions to collect data from a certain standardized bus that many lab instruments have. it does not provide any graphic displays for scalar displays for scalar fields or vector fields. Nor does it handle 3D data sets.) Scientific visualization, in contrast to lab-instrument data collection and analysis, deals with a wide variety of input data types (e.g., 2D scalar field, 3D vector field, etc.) and with a large range of output graphic displays (e.g., 2D color images, arrow displays, particle advection). A language for scientific visualization needs the ability to handle all of these input data types and output displays, but LabVIEW lacks such an ability. Further, the power provided by higher-order functions is useful for a language for scientific visualization, but LabVIEW does not have higher-order functions. (Chapter 3 will describe higher-order functions.)

2.2.3 Show and Tell

Show and Tell is a general-purpose programming language [Kimu86a] [Kimu86b] [Kimu86c] [Kimu86d]. Its intended users are schoolchildren. Show and Tell has particularly influenced the design of the language for visualization that will be described in Chapter 3.

As in most visual data flow languages, boxes represent functions/procedures and data flows over links between boxes. However, in Show and Tell boxes may also be constants, variables, containers of inconsistency, and iterators. Links carry integers and real numbers. A Show and Tell variable is simply an empty box capable of holding a number. A constant is a variable that has a value at the start of execution, and that may not be changed.

Inconsistency is a key concept in Show and Tell. Inconsistency is a way to represent control flow in a data flow language. Each box in a Show and Tell program is either consistent or inconsistent. Data flow may not continue through an inconsistent box. Boxes may become inconsistent in two ways: two different values flow into a variable box, or an open box containing a predicate (like '<') is evaluated to be false. Closed (solid) boxes contain inconsistency, while open (dotted) boxes allow inconsistency to propagate out to the next larger containing box. Cross-hatching (parallel diagonal lines on top of a box) indicates inconsistency.

Figure 2.7 shows how inconsistency affects data flow on links. Part (a) of Figure 2.7 shows how two, trying to flow over a link into a variable box that already contains three, causes the box surrounding the link to become inconsistent. This, in turn, prevents other data (i.e. one) from flowing through the inconsistent box. In part (b), an additional, closed box surrounding the link from two to three contains the inconsistency inside itself. Thus the one may flow

Figure 2.7: Show and Tell: the effect of inconsistency on data flow ([Kimu86a]).

Figure 2.8: Bounded sequential iteration in Show and Tell ([Kimu86a]).

unimpeded over the link which passes through the outermost box. Part (c) replaces the added closed box of part (b) with an open box. The open box, unlike its closed counterpart in part (b), does not contain the inconsistency. Rather, the inconsistency propagates outside the open box, and is contained by the outermost box, which is closed. Since the outermost box has become inconsistent, the one may not flow through it.

Show and Tell provides iteration through iteration boxes. (Figure 2.8 shows an iteration box. In Figure 2.8, the iteration box is the large, double-walled box. Specifically, Figure 2.8 is a program to sum the integers from six down to one.) An iteration box stops iterating when it becomes inconsistent. In Figure 2.8 this will happen when the variable in the iteration is no longer greater than zero.

Two kinds of ports can be attached to iteration boxes: sequential ports and parallel ports. Show and Tell's ports are not for input/output with all boxes, but only for iteration boxes.

Sequential ports are used for recycling a single data value from one iteration to the next. The value may be changed during any of the iterations. A sequential port is analogous to the variable X in the Pascal code:

```
FOR I := 1 TO 10 DO
   X := X + I;
```

Sequential ports are always used in pairs with one on each side of the iteration box. In Figure 2.8 two pairs of sequential ports are used. The top pair helps give a value to the counter variable, and the bottom pair maintains the sum variable. (The counter variable starts at six

Figure 2.9: Parallel iteration in Show and Tell ([Kimu86a]).

and is zero when the program terminates; the sum variable's initial value is zero and its final value is 21.)

Parallel ports are used for transferring a whole series or collection of data values between boxes. A parallel port is analogous to the array INFO in the Pascal code:

```
FOR I := 1 TO 10 DO
   X := X + INFO[I];
```

where INFO is a one-dimensional array. Figure 2.9 shows two iteration boxes connected by parallel ports. When the top box becomes inconsistent, iteration in both boxes ceases. While the iteration is occurring, if an inconsistent lower box is created, it is discarded.

Not only does the language described in Chapter 3 add features beyond those just mentioned (i.e. beyond inconsistency, iteration boxes, sequential ports, and parallel ports) to Show and Tell; it fundamentally reorients the language by focusing it on the application domain of scientific visualization, and on the user community of scientists and engineers (rather than schoolchildren).

Show and Tell is a powerful and elegant language. It has iteration, procedural abstraction, recursion, and file and database operations. It is as powerful as any other data flow language I have seen, and its programs are usually quite readable. It is the best visual data flow programming language I have reviewed.

Although Show and Tell has procedural abstraction, it does not possess type checking or higher-order functions. Type checking is desirable for catching type errors at program-construction time, well before a program is run. Type checking also provides a basis for type inference and type propagation, which can provide useful information about what type of data each link will carry, what types of input data various functions will demand, and what types of output data functions will produce. Higher-order functions (i.e., functions which take other functions as arguments) add power and flexibility to a language. The extra power comes from the fact that allowing one function to take another function as an argument leads to a more visually compact representation for programs than would otherwise be possible. The extra flexibility that higher-order functions give a language is due to a new possibility presented by such functions. Higher-order functions make it possible to radically alter the effect of a program by simply changing the first-order function taken as an argument by a higher-order function.

The language for visualization described in Chapter 3 adds type checking and and higher-order functions to Show and Tell, as well as implementing inhibitory arrows. Such arrows were suggested by Kimura. Inhibitory arrows allow the transmission of inconsistency, rather than data, and are useful in making IF...THEN statements in Show and Tell more concise.

In the data flow visual languages I have referenced or surveyed, the data flow computational model has been used for everything from music to user interfaces to scientific visualization. Since the focus of the data flow model is data, the data flow model seems to work best for visual languages whose application area focuses on data. One such application area is scientific visualization. Thus, data flow seems an apt computational model for a visual language whose application domain is scientific visualization.

2.3 Scientific Visualization Systems

Scientific Visualization is an active area of research. The more sophisticated software systems created by this research are briefly summarized here to give an idea of what work has been done in creating software visualization systems, and to make clear the limitations of existing systems that I am trying to overcome with the language that is explained in Chapter 3.

AVS, apE, and the Iris Explorer are fairly extensive software systems for doing scientific visualization. AVS [Upso89] is the Application Visualization System, a commercial product of the AVS Institute (although originally created by Stardent). apE [OSU89] is the Animation Production Environment, originally produced at Ohio State University, and later handed off to a private company. It is similar to AVS, but has less functionality. Iris Explorer is a commercial product from Silicon Graphics. As of March 1992, it is available for sale, but has only been available for a month or two. From the little information available about Explorer, it shares the characteristics and weaknesses of AVS. Because AVS's features are a superset of apE's features and because Explorer is relatively unknown, yet similar to AVS, the discussion will focus on AVS.

AVS allows a user to connect various code modules together. Modules are represented by boxes with data flowing on links between them. Modules do such things as read numeric data in, transform one set of numeric data to another (for example, by interpolation and scaling), transform numeric data into geometry (for example, generate a polygonal isosurface from a scalar field), transform geometric data into images (rendering), and transform 3D numeric data into images (volume rendering).

AVS provides the following pictures (also called graphic displays or visualization techniques): 1) for 3D scalar fields, volume rendering. Volume rendering consists of casting rays through a 3D data set to directly create an image without making an intermediate geometric representation first. Intensities are then accumulated along the ray path to give an intensity for a pixel; the process is repeated for each pixel displayed. 2) For 3D scalar fields, isosurfaces can be shown. An isosurface is a surface of equal value in the scalar field. A connected net of polygons is used to approximate the isosurface. 3) For 3D vector fields, particle advection is available.

Although AVS has a powerful set of modules, AVS is not a complete programming language. Scientists who wish to customize available modules must write their own modules in a conventional programming language. AVS is not a *programming* language, because scientists cannot stay within AVS and define modules visually. Also, AVS is visual and based on data flow only at the top level; below that, one must write modules in textual C or Fortran. This means that a scientist wishing to change one of the AVS-provided predefined library modules must make two large, awkward mental leaps. The first leap is from a graphical user interface (boxes and lines) to a textual interface (C or Fortran code), and the second leap is from a data flow computational model (in which one thinks of data flowing from filter function to filter function) to a control flow model (in which one focuses on "which chunk of code gets executed next?").

The result of these two mental leaps is that AVS is difficult for its end-users (scientists) to modify. Indeed, AVS users have publicly complained about this difficulty [Trei90] [Palm92].

Visual data flow languages for visualization, however, need not be difficult for their end users to modify. DataVis has been designed to make customization much easier than it is in present visualization systems, AVS included.

In summary, there are several software systems for doing scientific visualization. They suffer from a number of weaknesses. Some systems, such as NCSA Image and DataScope (discussed in Chapter 1), have the available graphic displays hard-wired into the program. This restricts scientists to a limited set of options for visualizing data. Even systems like AVS and apE, which give scientists more flexibility by allowing users to connect components together, do not provide the power of a full high-level programming language unless you go outside AVS and use Fortran or C. Such a language is desirable so that scientists can add new kinds of data, new functions or modules, and new graphic displays to those already provided.

Finally, although some systems (AVS, apE, and Explorer) are based on data flow and offer a visual interface, they do so only at the top level. At lower levels, data flow is not present and textual code is used. Such languages do not offer a consistent computational model or interface across all their levels. This lack of consistency between the top level and lower levels is a weakness in AVS, apE, and Explorer. The top level will be easy for scientists to learn to use, since it is visual and is based on data flow. But scientists will inevitably want to customize existing top-level visualization functions and will want to make their own new top-level functions. In order to do this, they will need to use lower-level functions. In AVS, apE, and Explorer, they will encounter a barrier between the top level and lower levels. The lower levels will be in textual code and will not be based on data flow. Thus, lower-level functions will be harder for scientists to modify and understand.

Chapter 3 describes a language which remedies these deficiencies of present visualization systems. The language carries visual programs and data flow from the top level to a lower level than AVS, apE, or Explorer do. This makes it easier for scientists to change and understand functions at all levels in the language.

Chapter 3

DataVis: A Language for Visualization of Scientific Data

This chapter describes DataVis (data visualization), a visual programming language based on data flow for scientific visualization. Section 3.1 gives a general description of the language, while sections 3.2 and 3.6 describe DataVis's extensions to Show and Tell. Sections 3.2–3.10 present a bottom-up, implementor's view of the components of DataVis. In contrast, Chapter 4 will provide a top-down, end-user view of DataVis, which shows how a scientist might use DataVis.

DataVis is based on Show and Tell (which was discussed in Section 2.1), but does not include Show and Tell's file or database operations.

3.1 Introductory Description of DataVis

DataVis is intended for scientists who wish to visualize their data. Most scientists would rather study science than program to create basic software tools for visualization. In some cases, DataVis allows scientists to do visualization without programming. In other cases, DataVis relieves scientists of the necessity to memorize low-level syntactic details present in conventional textual languages.

DataVis provides a library of predefined visual programs that can be hooked up to icons representing data. If this library includes a program to produce the graphic display the scientist is interested in for the kind of data the scientist has, he or she can simply connect a data icon

Figure 3.1: An example call to the DataVis library program **vectsToArrows**.

to the program icon. It is not necessary for the scientist to do any programming. For example, an atmospheric scientist might wish to view wind velocity in a 2D horizontal plane at an elevation of 300 meters by using an arrow plot. Figure 3.1 shows the call to the DataVis **vectsToArrows** program. Hooking up the icon for the velocity data (that is a 2D vector field) to the vectsToArrows icon will cause the arrow plot to be generated and displayed. The 100 @ -700 in Figure 3.1 is a parameter to **vectsToArrows** that is set by the user. 100 @ -700 is a point, and indicates that the arrow plot should be displayed in a window 100 pixels to the right and 700 pixels above the window holding the **vectsToArrows** icon box. An arrow plot may be seen in Figure 3.2, overlaid with a contour plot. (The contour plot would be generated from a scalar field by a second DataVis program, not by **vectsToArrows**.) (This example is presented in more detail in Chapter 4.) As a result of the ease of this visualization effort, the scientist can spend time on science rather than on programming.

If the scientist wants a new graphic display that is not generated by the DataVis library programs, or wants to visualize a new kind of data that the library programs do not handle, DataVis can be extended in two ways. The main way is to hook together some predefined DataVis functions (also called components). These functions are written in the same DataVis visual language as the predefined programs. The only difference between DataVis programs and DataVis functions is that the functions are pieces of the programs. The functions will do things like iterate over a 2D data field, interpolate data, and display a color image on the screen. That is, the functions are also less complete than the programs, since a program will handle the complete job of changing data into a graphic display, whereas a function will only do a part of that job. Although the functions are less complete and less powerful than the programs, the functions are still high-level enough to free the scientist from dealing with low-level syntactic details.

Figure 3.2: Arrow field graphic display, overlaid with a contour plot.

For example, Figure 3.1 shows the call to the complete DataVis **vectsToArrows** program. This program does all the work of calculating the arrows generating the graphic display (a color image–in this case an arrow plot), and displaying the image on the screen. In contrast, Figure 3.3 shows the DataVis function **displayAt:** (as part of **displayMergedImages**). **DisplayAt:** is only a piece of a program. It simply takes a color image and displays it on the screen at an offset from the window containing the top-level DataVis program.

The fact that DataVis may be extended by hooking predefined components together points out an important reason to base a language for visualization on the data flow computational model. Data flow is a natural metaphor for the visualization process because visualization involves the transformation of numeric data into pictures. Data flow, when used with a visual language like DataVis, allows the scientist to watch this transformation: a flow occurs from the numeric input data, through various functions (each of which transforms the data in some way), to the pictures or graphic displays that are the end result of the visualization process.

Since no extension language has a complete set of predefined library programs, a minor way that a scientist may extend DataVis is by programming in a lower-level textual language

Figure 3.3: The DataVis **displayAt:** function (at right), as part of the **displayMergedImages** function.

(Smalltalk). However, this will usually be unnecessary because both the top level of DataVis and all but the bottom level will be visual. For example, major programming in Smalltalk would not be necessary to extend DataVis to handle three-dimensional fields of data, but would be needed to let DataVis handle using a CD-ROM player as a source of data.

DataVis is like AVS in that both are visual at the top level. However, the visual part of DataVis extends farther downward to lower levels than the visual part of AVS does.

The following sections of this chapter explain the salient language features of DataVis. Section 3.2 deals with the box-line visual representation used by DataVis.

3.2 Box-line Representation, Unidirectional or Bidirectional Flow of Data, and Execution Mode

DataVis uses the standard visual data flow language representation: boxes are functions and lines carry data between them. This representation is more concise than that used in an early version of HI-VISUAL (see Figure 2.2), since DataVis does not make a box to represent every intermediate data object created during execution of a complex program (whereas HI-VISUAL's early version did). However, if the user wishes to view such intermediate data, DataVis makes it easy to insert a variable box (to show a single scalar number) or some viewing-monitor function (for more complex data objects—2D scalar fields, for example).

Figure 3.4: Bounded sequential iteration in Show and Tell [Kimu86a].

For example, Figure 3.4 could be modified by adding a number box to the right of '+' but still inside the iteration box.

Section 3.4 will more fully explain Figure 3.4. The modified program may be seen in Figure 3.5. (Since DataVis is in the same family of languages as Show and Tell, Figure 3.5 shows what can be done in DataVis, not solely what is possible in Show and Tell.)

For now, suffice it to say that Figure 3.4 sums the integers from six down to one. The sums run through the '+' box. The new number box (to the right of the '+') in Figure 3.5 will show all the intermediate, partial sums as the iteration occurs, rather than just the final sum of 21 after the iteration has stopped.

DataVis, like all other visual data flow languages except Fabrik [Inga88], has unidirectional flow of data on its links. Unidirectional flow is easier to implement than the bidirectional flow espoused by Fabrik. In addition, DataVis's unidirectional flow of data avoids the problem of finding an inverse for functions like max and min, or loops.

Bidirectional data flow does not seem to be very useful for scientific visualization. Since unidirectional data flow is easy to implement and to think about, I decided to use unidirectional data flow in DataVis.

The one place where bidirectional data flow would be useful is to let the user point at a final graphic-display image and see what initial scientific data resulted in it. This is an interesting and useful research problem, but I did not address it.

Like all other data flow visual languages other than Cantata [Rasu90] [Will90] [Rasu91] and VPL [Lau91], DataVis uses data-driven execution of programs. (Recall section 2.1.1.8 on

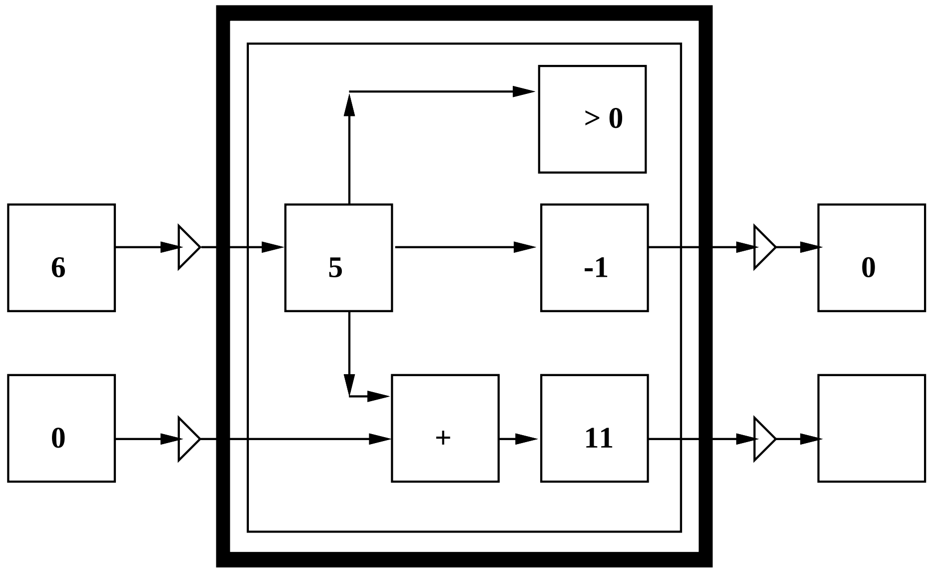

Figure 3.5: Modified version of Figure 3.4: additional number box showing partial sum of 11, during execution, after five has been added.

data-driven and demand-driven execution of programs.) This choice makes possible a simpler implementation than demand-driven execution would require.

On the other hand, demand-driven execution saves unnecessary computation, since a function does not fire (i.e., execute) until its output data is requested by a downstream function. This is particularly useful for languages with application domains that include compute-intensive functions. Image processing is one such domain. Since Cantata and VPL deal with image processing, they provide demand-driven execution. (Cantata also provides data-driven execution, at the user's option.)

Demand-driven execution could be added to DataVis. (And certainly, scientific visualization is a domain that can benefit from demand-driven execution, since scientific visualization that often has compute-intensive functions.) Although the primary benefit of demand-driven execution is improved performance (achieved through not executing compute-intensive functions unless a downstream function demands their output data), the current slowness of the DataVis prototype implementation is due to other factors (mentioned in Chapter 6). These other factors—bottlenecks, actually—would have to be removed before demand-driven execution could improve the performance of DataVis. Because demand-driven execution would not immediately make DataVis faster, and because data-driven execution is simpler to implement, I chose to use data-driven execution in DataVis.

3.3 Inconsistency

Inconsistency is a key concept in both Show and Tell and DataVis, and is found only in these two languages, and in ESTL/CUBE [Najo90] [Najo91]. Inconsistency is a way to represent control flow in a language based on data flow. Each box in a program is either consistent or inconsistent. Data flow may not continue through an inconsistent box. Boxes may become inconsistent in three ways: two different values flow into a variable box, or an open box containing a predicate (like '<') is evaluated to be false, or a field or collection feeding a parallel port on an iteration box is exhausted. (I will discuss fields, collections, ports, and iteration boxes in the next section.) Closed (solid) boxes contain inconsistency, while open (dotted) boxes allow inconsistency to propagate out to the next larger containing box. Cross-hatching (parallel diagonal lines on top of a box) indicates inconsistency. Figure 3.6 shows how inconsistency affects data flow on links.

One way that DataVis uses inconsistency is as a valve to turn on and off the flow of data. Show and Tell also uses inconsistency in this manner. Often this is done by shutting off data flow over a link that crosses a large box which just became inconsistent, perhaps because a predicate such as '>0' was evaluated to be false (i.e., inconsistent).

Another way that DataVis uses inconsistency is to terminate loops. (Loops were explained in section 2.2 and will be discussed again in section 3.4.) An iteration box stops executing its contents repeatedly only when it has become inconsistent.

One more use for inconsistency in DataVis is to return a value from a function. For example, imagine that in Figure 3.4 we replace the open (dotted-outline) predicate '>0' with an open user-defined icon box **foo**. **Foo** would return a value of consistent or inconsistent. The return value would be indicated by **foo**'s turning cross-hatched if it were returning 'inconsistent'. If this happened, since **foo** was an open box, its inconsistency would spread outside of itself to affect (and stop) the whole iteration box.

Although DataVis also uses inconsistency for its IF...THEN...ELSE statement, it augments inconsistency with inhibitory arrows, a construct suggested by Kimura to solve some problems

Figure 3.6: Show and Tell: effect of inconsistency on data flow ([Kimu86a]).

Figure 3.7: Example of a DataVis IF...THEN statement...ELSE statement:
. IF $6 > 9$
. . THEN direct flow of data to box A
. . ELSE direct flow of data to box B.

with expressing IF...THEN statements [Kimu86a]. Whereas other arrows in DataVis propagate data and type information, inhibitory arrows propagate inconsistency. If the source box for an inhibitory arrow is consistent, the destination box will be inconsistent. If the source box is inconsistent, the arrow will have no effect.

Figure 3.7 shows a DataVis IF...THEN...ELSE construct containing an inhibitory arrow. (Inhibitory arrows have white arrowheads and dashed tails.) If $6 < 9$, then the large upper box will be consistent, the inhibitory arrow will make the bottom box inconsistent, and data will flow to box A. If $6 \geq 9$, then the reverse will occur and data will flow to box B. To the right of A and B is **join**, a DataVis function that accepts whichever one of its two inputs is present, and produces that input as its output.

Note that the DataVis IF...THEN...ELSE construct is skewed more toward data flow than control flow: the condition's truth or falsity determines not which chunk of code is executed, but which way the data flows.

3.4 Iteration

DataVis, like Show and Tell, provides iteration through iteration boxes. (Figure 3.4 shows an iteration box. Specifically, Figure 3.4 is a program to sum the integers from six down to one.) DataVis repeatedly executes the contents of an iteration box until the box becomes inconsistent, whereupon the box stops iterating. In Figure 3.4 this will happen when the variable box inside the iteration box is no longer greater than zero.

Two kinds of ports can be attached to iteration boxes: sequential ports and parallel ports. Ports regulate the flow of data objects into and out of iteration boxes.

Sequential ports recycle a single data value from one iteration to the next. The value may be changed during any of the iterations. A sequential port is analogous to the variable X in the Pascal code:

```
FOR I := 1 TO 10 DO
   X := X + I;
```

Sequential ports are always used in pairs with one on each side of the iteration box. (The two sequential ports in a pair can be identified visually by the fact that they are always horizontally level with each other.) Thus Figure 3.4 shows two pairs of sequential ports. The top pair helps to give a value to the counter variable, and the bottom pair maintains the sum variable. Specifically, after each execution of the contents of the iteration box, the value of the right sequential port (in each pair of ports) will be copied ("recycled") back to its corresponding left sequential port.

Parallel ports transfer a whole series or collection of data values between boxes. A parallel port is analogous to the array INFO in the Pascal code:

```
FOR I := 1 TO 10 DO
   X := X + INFO[I];
```

where INFO is a one-dimensional array. Figure 3.8 shows two iteration boxes connected by parallel ports. When the top box becomes inconsistent, iteration in both boxes ceases. While the iteration is occurring, inconsistent lower boxes are discarded.

While a pair of sequential ports recycles a single (possibly changing) value between iterations, parallel ports, in contrast, either construct or disaggregate a group of data elements.

Figure 3.8: Parallel iteration in Show and Tell ([Kimu86a]). Part a) (Syntax) is at left; part b) (Semantics) is at right.

Figure 3.9: The DataVis **vectsToArrows** program.

The group will either be a field of elements or a collection of elements. (A field has multiple rows; a collection is all one large row.) Two parallel ports can be seen in Figure 3.8, and Figure 3.9 shows two more parallel ports. Figure 3.8 executes the α function repeatedly, sending α's outputs to β. Function α continues to execute over and over until something within α causes its iteration box to become inconsistent. Then α's execution will cease, as will that of β.

Finally, Figure 3.9 is the **vectsToArrows** program and will be explained more fully in Chapter 4. Here, however, the object of interest is the iteration box with its input parallel port and its output parallel port. The input parallel port will eventually cause the iteration box to become inconsistent when the port has disaggregated and exhausted all the individual vectors in the **VectorField**. This inconsistency will then make the iteration stop, and the new

Figure 3.10: The DataVis **vectorToArrow** function.

field (which the output parallel port had been forming while the iteration continued) will be transmitted downstream to **displayMergedImages**.

3.5 Procedural Abstraction

A DataVis procedure box is the definition of a function/procedure and an icon box is the call to that function/procedure. The phrases *procedure box* and *function definition* will be used interchangeably, as will *icon box*, *call to function*, and *call to procedure*.

A procedure box displays the name of the procedure in its upper left corner. All functions have a textual name. They may also have an optional graphical, iconic name. The iconic name is a small bitmap that the user creates with a bitmap pixel editor. Figure 3.10 is the **vectorToArrow** function. (The graphical portion of its name area is simulated here with black lines; in the prototype, however, the graphical portion is a color bitmap.)

Base boxes are input and output parameters for functions. Base boxes may be recognized by their heavy outline. Those for input are generally located toward the left of the procedure box, and those for output, toward the right. As Figure 3.10 makes clear, the **vectorToArrow** procedure box has one input base box and one output base box.

An icon box is a call to a procedure. The icon box has the same combination of graphical and textual names as does its corresponding procedure box. Thus in Figure 3.9, the textual name "vectorToArrow" and the graphical name (arrows) above the textual name are the same as the name area in the top left corner of the **vectorToArrow** procedure box in Figure 3.10.

Arrows to (from) an icon box correspond to input (output) base boxes in the matching procedure box. For example, the incoming arrow on the left of the **vectorToArrow** icon box in Figure 3.9 transmits its data to the "Vector" input base box at the left of Figure 3.10. Conversely, the "Rotated Image" output base box at the right of Figure 3.10 sends its data to the arrow exiting the right side of the **vectorToArrow** icon box. (See Figure 3.9.)

3.6 Type Checking, Type Inference and Propagation, Color of Links, and Program Animation

Data is typed. The types of data which DataVis allows include: an individual scalar number (real or integer), an individual vector, a 2D scalar field, a 2D vector field, a point, an image (a 2D color bitmap), a 2D field of points or images, and a collection (of one of the previous types that is not a scalar or vector). The distinction between the last two types (field and collection) is that a collection lacks the internal row-by-row structure which a field possesses.

DataVis does type checking at program-construction time by not allowing a user to connect boxes with types that do not match. DataVis displays a pop-up error message if the user tries to connect two boxes of different types.

Although figures in this thesis portray links between boxes as thin black lines (for simplicity and because color figures are difficult to produce and to include in a thesis), the software prototype of DataVis gives links color.[1] Color of links conveys two kinds of information in DataVis. First, links are colored to indicate the type of data they carry. The following correspondences between data type and link color exist: scalar ⇒ light blue, vector ⇒ light orange, scalar field ⇒ light purple, vector field ⇒ light brown, point ⇒lightGreen, field of point or image ⇒ lightGreen, collection ⇒ lightGreen, and image ⇒ lightCyan. Links are colored white if the data type which they are to carry is unknown.

This first kind of link coloring occurs before run time—either when the user enters data into certain boxes in a program or at program construction time.

[1] It is possible to get color photocopies of the original color figures in this thesis. See the appendix for further information.

Figure 3.11: DataVis program before data entry: empty boxes and white links.

Figure 3.12: DataVis program of Figure 3.11 after data entry: type inference, type propagation, and link coloring.

Figure 3.11 is an example of link coloring that happens when the user enters data into boxes. For this example only, assume that DataVis distinguishes the data types "real scalar" and "integer scalar", and colors links orange for reals and blue for integers. When the user first constructs this program, all boxes are empty and all links are white, since their types are unknown.

DataVis does type inference: as far as possible, it tries to determine what the type of a link or box should be, based on available type information. Further, DataVis performs type propagation: determining the type of one link or box often gives information about the types of other links and boxes. As the type of each link is identified, the link is colored appropriately. Often, entry of data of a certain type into a box on one side of a program will cause a wave of color to sweep through links, to the other side of the program, as DataVis infers and propagates types.

This "wave of color" occurs in Figure 3.12. As part of type inference, DataVis keeps information for each function on the acceptable types of that function's input parameters and output parameters. (For Figure 3.12, assume the acceptable type combinations for the $\star$ function are those shown in Table 3.1.)

input 1	*input 2*	*output*
integer	integer	integer
integer	real	real
real	integer	real
real	real	real

Table 3.1: Acceptable type combinations for $\star$ function in Figure 24

Figure 3.13: DataVis type propagation and type inference due to program construction by adding links.

When the user enters two into the lower left box, its outgoing link turns blue to show that the type just propagated to it is integer. No further links are colored, because there is insufficient type information to uniquely determine the type of $\star$'s first (i.e. upper) input link and the type of its output link. However, when the user enters 2.5 in the upper left box, DataVis propagates that real type to $\star$'s first input link and turns the link orange. DataVis then infers that $\star$'s output type is real and propagates that real type to $\star$'s output link, thus turning that link orange.

Note that all this type inference and propagation happens before the program is ever run. Type information is at a higher liveness level (three on Tanimoto's scale) in DataVis than regular data (two on Tanimoto's scale). This is the case because as soon as something is done that gives one function or link more type information (e.g. entering a scalar into a box, or adding a link between two boxes, where at least one box's type is known), that type information is immediately propagated throughout the program. The propagation occurs without any user intervention. In contrast, regular data is only transmitted when the user gives the **start** command to begin running the program. Making regular data be at liveness level three would cause DataVis to attempt execution of incomplete programs while they were still being constructed. I believe that confusion would ensue.

Another example of the first kind of link coloring can be seen in Figure 3.13. In this example, type inference and propagation happen not as a result of entering data into boxes, but from program construction—specifically, adding links. Assume that the three boxes containing one, one, and five had those values entered before outgoing links were added to those boxes. Then adding outgoing links to boxes holding one (on the right) and five (at the top left) would turn just those links blue. No other links would change from their initial white color, because no other links' types can be inferred yet. But as soon as the user adds an outgoing link to the "one" box (on the left side of Figure 3.13), a wave of color spreads through all the white links in the program.

Links to the left of the output parallel port (on the right side of the iteration box Box1) turn blue to indicate that their type is scalar. However, links between lines A and B turn purple: their type is scalar field. The parallel port on the iteration box Box1 takes the individual scalar data objects coming out of that box and composes them into a scalar field. Finally, links to the right of the input parallel port (on the left side of the iteration box Box6) turn light blue. Their type is scalar because the parallel port on the iteration box Box6 disaggregates the scalar field into its individual scalars.

The color of links conveys not only type information at program construction time, but also program animation information at run time. This second kind of link coloring can be turned on or off by the user within each layout. (A layout is a window containing one DataVis program or procedure.) When program animation is turned on in a layout, each link in that layout temporarily redisplays itself in a darker version of its normal color when the link is transmitting data across itself. When the link is done transmitting that data object, it returns to its former lighter color. For example, a scalar link temporarily "flashes" to a dark blue, while a vector link flashes to dark orange.

Another program animation feature that is helpful in watching a program execute is that numbers dynamically appear first in boxes upstream of an icon box, then in the procedure box's input base boxes, then in the output base boxes, and finally in boxes downstream from the icon box. Together with link flashing, this dynamic display of data in boxes has the potential to be useful for teaching someone how certain DataVis code works, or for figuring out how someone else's DataVis code works.

Conveying the information given by flashing links and dynamically displayed data would be difficult at best in a textual language such as C or Fortran, but fits quite naturally into a data flow visual language like DataVis. Also, giving that information in a textual language would require considerable programming effort on the part of the user. In contrast, the DataVis user gets these features for free!

Records are a possible method for allowing a user to define new types of data. Data objects of these types could then flow over links in a DataVis program.

Figure 3.14: DataVis stop mark on a '+' box.

3.7 Stop Marks

Stop marks in DataVis are akin to a similar feature in Show and Tell. A stop mark is a breakpoint, and is represented visually by a red 'X' on the top left corner of the box that has been marked. Figure 3.14 displays a stop mark on a '+' box. The stop mark halts execution of the program just before the + begins its work. The user will be put into the standard Smalltalk debugger. (DataVis is implemented in Smalltalk.) Stop marks are a useful tool for debugging DataVis programs, since the debugger allows inspection and modification of variables. However, it would be nice to have a more visual debugger than the Smalltalk debugger.

3.8 Pop-up Comment Boxes

DataVis provides pop-up comment boxes for procedure boxes, icon boxes, input and output parameters, variables, simple operators (e.g. +) and predicates (e.g. $>$), and for layouts (windows containing a DataVis program or function). These comment boxes are useful for documentation. They give information on the box's purpose, name, current type, set of allowable types, number of inputs and outputs, and, if appropriate, the particular instance that this box represents.

All this information can be very useful to a scientist looking at the icon box for a predefined DataVis library function or program. The comment box will tell him or her what the program or function does, what types of input data it expects, and what type of output data it produces. The idea is to give the scientist some clue about which other functions or programs this function/program may be connected to. For example (see Figure 3.14), since +'s allowable input types include "scalar", it may be hooked up to the output of any other function whose output type is "scalar".

The reason that it is useful to describe functions' types by textual comments, in addition to coloring links, is that links only show up after two functions (i.e., icon boxes) have been connected. But as part of type checking, DataVis will not allow the user to create a link by connecting two functions unless their types match. Thus, textual comment boxes containing information on types can help prevent the scientist from being frustrated by only being able to discover a function's type by trial and error: repeatedly trying to link a function to various functions. Eventually some function's type would match the original function's type, but the scientist would just have had a frustrating, user-hostile experience with DataVis. Although coloring links is not a good way to discover a function's input and output types *before* the function is part of a DataVis program, coloring links by types is still a useful way to convey type information to a user about an *existing* program, for purposes of debugging one's own program or understanding someone else's already-written program.

3.9 Higher-order Functions

DataVis provides higher-order functions, which can take other functions as arguments. Higher-order functions add power and flexibility to a language. Higher-order functions are a recent development in visual programming languages. Many earlier visual programming languages do not have them. Of all the data flow visual languages that I surveyed [Hils92], only *viz* [Holt90], CUBE ([Najo91]; also called ESTL [Najo90]), and DataVis provide higher-order functions. More recently, the data flow visual programming language VPL [Lau91] provides them. Both HI-VISUAL and LabVIEW lack higher-order functions.

There are several ways to represent higher-order functions in a visual programming language. One way is to allow functions to flow over links; that is, to allow functions to be treated as data objects. If this is done, an "Apply" node can then be used to apply the function to its inputs, and to produce as output the function's result.

Alternatively and more clearly, a language can present higher-order functions via function slots. A function slot is a box inside the icon for a higher-order function. The higher-order function (or one of the functions it calls) will eventually apply the first-order function to some data. When a language uses function slots, it does not allow higher-order function slots to flow on links, but only to be transmitted between function slots.

DataVis uses function slots, rather than functions flowing on links, to implement higher-order functions. A DataVis higher-order function may have only one function slot. The reason that only first-order functions may be placed in a function slot is that only a first-order function's icon box is small enough to visually fit into the function slot. A higher-order function's icon box must itself contain a function slot, which makes that icon box too large to fit into a function slot.

FieldIterator, shown in Figure 3.3, is a DataVis higher-order function. The function slot is the box at the top of the **fieldIterator** icon box. The function slot holds a first-order function—**display:x:y:onto:**. **DisplayMergedImages** takes a field or collection of small individual one-arrow bitmaps, and displays them onto a large bitmap containing all the arrows. The large bitmap is the arrow plot that is the desired graphic display. The arrow plot is then displayed on the screen.

In DataVis either the higher-order function or one of the functions it calls will eventually apply the first-order function in its function slot to certain data. In this case, we see in Figure 3.3 that **graphicTransform** itself applies **display:x:y:onto:** to certain data. Essentially, **display:x:y:onto:** is told to display a small one-arrow bitmap, at a certain x and y index, onto a larger bitmap which will become the arrow plot. (Figures 3.9 and 3.3 show just two parts of the DataVis **vectsToArrows** library program, which will be explained more fully in Chapter 4.)

Higher-order functions substantially increase the power and flexibility of a language (such as DataVis) that has them. They allow more general functions to be written, functions that are less bound to specific details associated with their use in one particular program. Such general higher-order functions can be adapted to a variety of situations simply by plugging a new first-order function into their function slot.

As an example of the flexibility and generality that higher-order functions give a language, consider the **fieldIterator** higher-order function in Figure 3.3. Although the **vectsToArrows** program just discussed puts just one particular function in the function slot of **fieldIterator**, any other function can be used. For example, a DataVis program **scalsToImage** that produces a 2D color image from a 2D scalar field might call **displayMergedPixels**—a function quite similar to **displayMergedImages** (Figure 3.3)— by feeding it a 2D scalar field where the scalars are color-table indices. Thus each scalar would be an index to the value of a colored pixel. (See Figure 3.16.) Instead of putting **display:x:y:onto:** in **fieldIterator**'s function slot, one could put the function **pixelOn:x:y:in:**. As before, the function parameter (**pixelOn:x:y:in:**)

Figure 3.15: Generality of higher-order functions: call to **passThruFieldIterator** higher-order function as part of **displayMergedPixels** function, which is part of DataVis **scalsToImage** program.

would be applied to each field element (here, a color-table index, not a one-arrow bitmap), at the current x and y indices, and with a graphic display (a large bitmap). **PixelOn:x:y:in:** would set the pixel in the graphic display at location (x,y) to the value in the color table at the index given by the field element. Also, **displayMergedPixels** calls **passThruFieldIterator**, which is similar to **fieldIterator**. However, **passThruFieldIterator** passes a palette (i.e., a color table) through itself—unchanged—and out to **restorePalette**.

Other uses for **fieldIterator** can be imagined. The beauty of a higher-order function like **fieldIterator** is that it can iterate over a field of anything, whether the field contains scalars, vectors, small arrow bitmaps for an arrow plot, or pixels in an image. Other iterators can be created, such as an **imageIterator**. Its function slot would hold a first-order function that would be a local image-processing function—for example, a function to average the values of the four nearest neighbors of a pixel. **ImageIterator** would then apply this function to every pixel of an image, producing a new, processed image as its result.

Note that although a first-order function may be used as an input argument to a higher-order function by placing the first-order function in the function slot of the higher-order function, DataVis does not allow a first-order function to be an output argument of a higher-order function. That is, a DataVis higher-order function may not produce a first-order function (or *any* function) as output. Nor does DataVis allow first-order functions to be stored in variables.

This means that DataVis does not implement higher-order functions to the full generality of a language like Scheme. DataVis higher-order functions are more like Ada generics, which means that they could be implemented by macro expansion. In fact, the current implementation of DataVis is more like Scheme than Ada. Higher-order functions could probably be made more efficient by taking advantage of the restricted way they are used.

3.10 Representing Scientific Data

DataVis does not provide the ability to graphically represent scientific data sets at a detailed level—e.g. individual scalars or vectors in a field. Rather, DataVis represents the entire data set as a single icon, called a data box. For example, WindVelocity in Figure 4.1 is a data box representing a 2D vector-field scientific data set.

It might be useful to give DataVis the capability to let the scientist graphically access individual elements of a data set—e.g. pixels in an image, scalars or vectors in a field, etc. Such access could take two forms. First, the access could be direct manipulation to help accomplish programming-by-example—i.e., the system would watch the user select a particular vector and would generalize on some basis from that selection to a program to process the whole field. The second possibility is that such graphical access to one vector in the field could somehow be used to help specify the visual program. For example, the user could select a row and column in the vector field that would represent the ith row and jth column of the field in some algorithm that operated on the field. The row and column that were selected would be reverse-highlighted, and would—forever reverse-highlighted—form a part of the visual representation of the DataVis program that embodied the algorithm that processed the field.

The only data flow language that I am aware of that provides graphical access to pieces of a data set is MPL [Yeun88]. MPL's application domain is matrices. MPL allows users to select one row or column of a graphical representation of a matrix, or the whole matrix, and to make the graphical representation of the matrix or matrix fragment be part of the program. So MPL provides the second kind of graphical access to a data set that was mentioned above.

I chose not to include graphical access to portions of data sets in DataVis for two reasons. First, I associated such graphical access with its first form—i.e., programming by example. I do not believe that programming by example is effective, because the user repeatedly has to tell the system *on what basis* to generalize from the example provided by the user. Second, I was more interested in visual representations of the computation—i.e., the program and its data flow—than I was in visual access to portions of initial data sets. After all, the entire DataVis language is geared toward full-fledged visual displays of processed data sets. I felt that was a more important problem than becoming enmeshed in the details of crude visualizations for portions of unprocessed data sets, which I believe is a relatively unimportant problem.

This chapter has given a bottom-up, implementor's view of the components of DataVis. Chapter 4, however, will provide a top-down, end-user view of DataVis. Chapter 4 will give some examples in order to show how a scientist uses DataVis.

Chapter 4

A Scientist's View Of DataVis

Chapter 3 presented a bottom-up, implementor's view of DataVis. This chapter views DataVis from a different perspective, giving a top-down, end-user presentation of the language.

One of the main design goals of DataVis was that it be easy to learn. Towards that aim, it was designed in three layers, with the first layer being simple and intuitive and each of the other layers being a simple extension of the previous one. This should reduce the learning curve of DataVis by making it possible for people to learn it a piece at a time.

The first, *connection* layer of DataVis consists of the simple point-and-click connection of first-order functions. This layer makes available to the user a large library of predefined DataVis first-order functions. This layer is all that someone needs to start using DataVis, and is equivalent to the visual part of AVS or apE.

Once someone has learned to use DataVis, it is easy to learn the next level, which is to define new functions. Any part of a DataVis program can be turned into a user-defined function by wrapping it in a procedure box and replacing its inputs and outputs with base boxes. This makes it easy to add to the function library. The set of features that are used to define new functions is called the *definition* layer.

The third layer of DataVis consists of those features that are used primarily to design library functions, and are not likely to be used in a top-level program. This includes inconsistency and using iteration boxes in their full generality. Because this set of features is not needed for typical DataVis programs, it is called the *expert programming* layer.

This chapter gives five examples to show how a scientist can use DataVis. The examples are divided into three groups. Each group of examples corresponds to one particular layer of use of DataVis. The examples progress from the easiest, *connection* layer through the medium-complexity *definition* layer to the advanced *expert programming* layer.

The first two examples involve the *connection* layer of DataVis. Using this level is little more than calling an existing program, and does not require any programming. This kind of use of DataVis should be typical of a scientist who is just starting to learn about DataVis, and does not require much knowledge of the system. This is about the same layer as that provided by AVS, apE, and Explorer, and is above the layer of NCSA Image and NCSA DataScope. It is expected that many scientists who use DataVis will not need or wish to go beyond this layer.

The second two examples show how a scientist can modify or extend an existing DataVis library program by using predefined functions and the other features present in DataVis's *definition* layer. DataVis's *definition* layer involves calling higher-order functions (including predefined library functions), defining first-order functions, and using iteration boxes with only parallel ports. For instance, one predefined function, **fieldIterator**, iterates across the elements of a 2D field, applying a first-order function to each field element. Another DataVis library function, **displayAt:**, displays an image (i.e., bitmap) on the screen. The *definition* layer includes things that can only be programmed in C or Fortran in other visualization systems. The second two examples show how to modify a DataVis program without getting into the lowest layers of the language. Just as it is thought that many scientists will not move beyond the layer of use of DataVis given by the first example, in the same way it is anticipated that many users who do work at DataVis's *definition* layer will not delve deeper into DataVis.

The third group of examples shows how a scientist can create a new function or program using the full range of DataVis's constructs. This is the *expert programming* layer of the system, which involves using inconsistency, defining higher-order functions, and using iteration boxes with sequential ports or global inputs, not just parallel ports. Chapter 3 has discussed these elements of the system. Probably only a small portion of all the users of DataVis would use this layer of the language. However, it is a simple extension of the previous layer, so those who are motivated will not need to learn many new concepts.

Note that it is not *necessary* to learn the *expert programming* layer in order to implement new visualization algorithms (i.e., algorithms not embodied in DataVis library functions)—for

Figure 4.1: An example call to the DataVis library program **vectsToArrows**.

example, the *marching cubes* algorithm to create an isosurface from a 3D scalar field. At least some new algorithms can be programmed using just the *definition* level. However, if the user wishes to do so, the *expert programming* level may be used to implement new visualization algorithms.

4.1 Two Point-And-Click Examples: How A Scientist Can Use DataVis Programs

This section presents two examples that demonstrate the easiest, *connection* layer of DataVis. The first example shows how a scientist runs a predefined DataVis program to produce an arrow-plot image from a vector-field data set. The second example concerns how a scientist runs a program to produce a contour-plot display from a scalar field, and how the scientist can merge this display with the earlier arrow-plot display.

4.1.1 Example One: Producing An Arrow-Plot Image

A scientist wishing to run a predefined DataVis program first opens a DataVis window. To construct the call to **vectsToArrows** visible in Figure 4.1, the scientist gives the command `add procedure` (by selecting a command from a pop-up menu) and chooses **vectsToArrows** from a list of predefined DataVis programs and functions. The **vectsToArrows** icon box appears. The scientist gives the `add data from library` command and selects WindVelocity from a list of available data sets, whereupon the WindVelocity data box appears. **vectsToArrows** will produce an arrow plot from the WindVelocity data box, which represents a 2D vector field. This vector field might be the velocity of the wind in a horizontal plane at an altitude of 1,000 meters in a thunderstorm. The arrow plot consists of many small arrows, where each arrow represents one vector. A given arrow's orientation is dictated by its vector's direction, while the arrow's length is proportional to its vector's magnitude.

Although the user recalls that **vectsToArrows** needed a vector field as an input, he or she can't remember if it needs any other inputs. So the scientist gives the `functionInfo` command. In response, a pop-up comment box containing information on **vectsToArrows** appears. This comment box indicates that **vectsToArrows** needs two inputs: a vector field and a point. The point tells DataVis where to display the arrow-plot image made by **vectsToArrows**. The image will be displayed (in a separate window from the program) 10 pixels below and to the right of the top left corner of the current window.

So, with his or her knowledge about **vectsToArrows**'s required inputs refreshed, the scientist adds a number box and enters the point 10@10 into it. Finally, the `start` command runs the program. **VectsToArrows** runs, produces an arrow-plot image, and asks the user what scaling is preferred. Scaling of 1.0 produces a normal-size image, while a scale factor greater than 1.0 makes an enlarged arrow-plot image. After the user gives a scale factor, DataVis opens a new window containing the arrow-plot image.

The user can do this whole process easily. Calling **vectsToArrows** is easy because the user only has to click five or six times with the mouse to construct and run the call to **vectsToArrows** in Figure 4.1. The user does not have to do any programming.

4.1.2 Example Two: Producing A Contour-Plot Image

This example shows how a scientist would produce a contour-plot image from the same vector-field data set used in the first example. Figure 4.2 displays the call to **scalsToContour**. **ScalsToContour** is a predefined DataVis library program to produce a contour plot from a scalar field. Since a scalar field, not a vector field, is the basis for generating a contour plot, Figure 4.2 uses **vectsToScals** to convert a vector field into a scalar field consisting of the magnitude of each vector in the vector field.

The **scalsToContour** function takes one piece of data (a scalar field) and four parameters: low and high contours, a contour increment, and an offset point. Figure 4.2 shows what happens when a scientist opens a DataVis window, gives the `add data` command, and selects Wind Velocity from a list of available data sets. This causes the Wind Velocity data box to appear. The scientist then adds four number boxes and enters the parameter values visible in Figure

Figure 4.2: An example call to the DataVis library program **scalsToContour**.

4.2 (10.0, 60.0, 5.0, and 10@10). By doing this, the scientist indicates that contour lines should be drawn for values of the scalar field ranging from a low value of 10.0 to a high of 60.0 by increments of 5.0. (In this case, the scalar field contains numbers which represent the magnitude of a vector field—wind velocity—at various points. So the contour lines will be drawn for a range of wind speeds, from 10 km/hr up to 60 km/hr.) Also, the fourth parameter means that after **scalsToContour** has produced a contour-plot image, it should open a window containing the image on-screen at an offset from the current window. (Here, the offset is 10 pixels below and to the right of the current window.)

To get **scalsToContour** to appear, the scientist gives the `add procedure` command and chooses **scalsToContour** from a list of DataVis predefined programs and functions. Next, the scientist gives the command `start`. In response, the **scalsToContour** program runs and asks the user the desired scaling for the window containing the contour-plot image. The scientist enters a scale factor and the system displays a window akin to that displayed in Figure 4.3.

Figure 4.3 shows two merged or overlaid images—one for an arrow plot and one for a contour plot. The call to **scalsToContour** in Figure 4.2 will produce an image similar to just the contour-plot portion of Figure 4.3. However, Figure 4.4 shows how the scientist can merge the arrow-plot image from the first example of this section with the contour-plot image produced by **scalsToContour**.

Figure 4.4 shows two example calls—one to **vectsToArrowsProduce**, one to **scalsToContourProduce**—to generate a combined arrow plot and contour plot, similar to the plot of Figure 4.3, from a single vector-field data set. **VectsToArrowsProduce** and **scalsToCon-**

Figure 4.3: Arrow-field image, overlaid with a contour-plot image.

Figure 4.4: A call to produce a merged arrow-plot and contour-plot image from a vector-field data set.

Figure 4.5: The DataVis **vectsToScals** function.

tourProduce are variants of **vectsToArrows** and **scalsToContour**, which produce an image as a data-object output, instead of themselves displaying it on the screen. **Merge** overlays the two plots (i.e., images), creating a single image which **displayAt:** puts up on the screen.

Figure 4.5 presents the definition of the **vectsToScals** function, which converts the Wind Velocity vector-field data set into a scalar-field data object. **VectsToScals** accomplishes this by returning a scalar field composed of the magnitude of each vector in the vector field.

Next, this example will describe portions of **scalsToContour** in greater detail. Note, however, that the scientist does not need to know any of this detail to run **vectsToContour** or to merge its image with the image generated by **vectsToArrows**. The commands just discussed are all that are needed in order to run **vectsToArrows**.

4.1.2.1 The Implementation of ScalsToContour

The implementation of **scalsToContour** is visible in Figure 4.6. **ScalsToContour** feeds the low and high contour values and the contour increment to **contoursInterval. ContoursInterval** produces an interval or sequence of contour values. (Section 4.3.1 will show how the scientist could create and define **contoursInterval.**) For example, for the input parameters (10.0, 60.0, and 5.0) in the call to **scalsToContour** (recall Figure 4.2), which calls **contoursInterval**, **contoursInterval** produces the interval (10.0, 15.0, 20.0, ..., 55.0, 60.0).

Figure 4.6: The DataVis **scalsToContour** program.

ContoursInterval sends its interval of contour values to **processCells**. **ProcessCells** treats the scalar field as if it were composed of numerous small cells. (Any four adjacent scalars in a square pattern in the scalar field may be considered a cell.) **ProcessCells** uses the cells in the scalar field to generate a contour-plot image. **ProcessCells** could produce no output data and could itself open a window to display the contour plot on the screen. However, in the interest of program modularity, **processCells** does not itself display the contour plot. Rather, **processCells** produces the contour-plot image as an output and sends it downstream to **displayContours**.

Note that in the icon box for **processCells**, the dashed box on the right indicates that **processCells** produces an image as a data object, rather than itself displaying the image on the screen. The solid box on the right of the icon box for **displayContours** means that **displayContours** takes the image data object and displays it on the screen. These meanings (a dashed box is an image data object that has not yet been put on-screen, and a solid box is an image that has been displayed on the screen) also hold true for the icon boxes of other DataVis library programs and functions.

DisplayContours is very simple, and is shown in Figure 4.7. **DisplayContours** calls **displayAt:**, a predefined DataVis library function that accepts an image and displays it on-

Figure 4.7: The DataVis **displayContours** function.

screen by opening a window containing the image and displaying the window on the screen, at the specified offset from the top left corner of the current window.

Recall that **processCells** accepts a scalar field, an interval of contour values (where one contour line is to be drawn for each contour value), and an offset point, and produces a contour-plot image. Figure 4.8 reveals the details of **processCells**.

Before explaining Figure 4.8, this example will discuss the algorithm that **processCells** uses to produce a contour plot. **ProcessCells** uses a technique called cell processing, which involves dividing the scalar field into a mesh of square cells. This technique processes the mesh cell by cell, checking each cell to see if each contour level crosses that cell. A contour level is wind speed = 25.0 km/hr, for example. If a contour level crosses the cell, the level may produce one or two contour lines. The algorithm plots these contour lines and processes the next cell.

The cell processing algorithm that **processCells** uses is:

Cell processing

Iterate over all cells:

1) For each cell:

  1a) Find minimum and maximum values over the set of four cell vertices.

  1b) For each contour level:

    1bi) If this contour level is in the range min...max,

      use vertex classification on this cell—i.e., how many

Figure 4.8: The DataVis **processCells** function.

contour lines intersect this cell and where their endpoints are.

1bii) Plot these contour lines.

Most of the **processCells** function in Figure 4.8 is inside the double-walled iteration box. At the top left of the function, the scalar field feeds into the iteration box both through a parallel port and as a global input. (The global input is just below the parallel port.) The parallel port splits up the scalar field into its constituent scalars and passes one element at a time into the iteration box. The field will automatically provide its elements in row-major order, from the top down. DataVis executes the interior of the iteration box once for each scalar in the field. Once all the field elements are exhausted, the iteration box becomes inconsistent, and the iteration stops.

Just below the parallel port is a global input carrying the scalar field. The ContourLevels (an interval or sequence) and Offset (a point) also enter the iteration box through global inputs. Each of these global inputs passes its value into the iteration box each time the interior of the iteration box executes.

At the middle and toward the bottom of the iteration box are three pairs of sequential ports. (For DataVis, the term *pair of sequential ports* means one sequential port on the left side of the iteration box and one port on the right side, where the two ports are horizontally aligned with each other.) For the moment, focus just on the bottom two pairs of sequential ports in Figure 4.8. Before the first iteration, the two sequential ports on the left side of the iteration box receive their initial value (one, for each of the two ports) from their number boxes. After the first and subsequent iterations the two matching sequential ports on the right side of the iteration box transfer their values to the matching sequential ports on the left. After the final iteration, the iteration box becomes inconsistent and the two sequential ports on the right send their values downstream (in this case, to two number boxes).

The bottom half of the iteration box contains a call to the **iteratorDriver** function. **DisplayContours** iterates over the scalar field (in this case, wind speed) that is its first input (see the input base box at the very top left of Figure 4.8). Iterating over the elements of a field often requires knowing the location of each element. **IteratorDriver** provides these locations.

At the top of the iteration box is a call to **getCell**. **GetCell** takes as input a point (xIndex, yIndex). This point comes from the arrow that exits from the top of the **iteratorDriver** icon box. The point is the location of an element of the scalar field. The other two inputs to **getCell** are the field element at (xIndex, yIndex) (through the incoming parallel port) and the entire field (via a global input which is just below the parallel port). Cell processing involves viewing the scalar field as a mesh of square cells. The cell-processing algorithm takes the scalars in the field to be vertices of the cells, so that every four scalars that are adjacent in a square pattern can be made into a cell (a 2x2 array). Thus **getCell** produces a cell of four scalars, a cell whose top left vertex is the scalar at position (xIndex, yIndex) in the field. **GetCell**'s graphic name (at the top of its icon box) indicates that it transforms part of a scalar field into a cell. **GetCell**, together with the incoming parallel port and the field, is equivalent to the "Iterate over all cells" portion of the cell processing algorithm. This example will not show the details of **getCell**.

The call to **processContours**, on the top right side of the iteration box, implements step 1b) of the cell processing algorithm. This call takes five arguments: 1) a cell; 2) an interval of contour levels; 3) a point (x,y), containing the current x and y indices from **iteratorDriver**; 4) an offset point; and 5) an image. When **processCells** first executes, the **newImage** function (to the

Figure 4.9: The DataVis **processContours** function.

left of the image, halfway down) creates a new, blank image (i.e., a bitmap). **ProcessCells** iterates over the cells. For each cell, **processContours** determines if any contour levels cross the current cell, and if they do, plots the appropriate contour lines on the image. The final version of the image is the completed contour plot.

This example will now show some of the details of **processContours**. Figure 4.9 is a detailed view of **processContours**. There are five input base boxes at the left. The interval of contour levels goes through an incoming parallel port into the iteration box, thus ensuring that the contents of the iteration box will be executed once for each contour level.

For a given cell, as Figure 4.9 shows, **rangeCheck** compares each contour level to the cell's vertex minimum and maximum in order to see whether that contour crosses this cell or not.

MinMax generates one of the inputs to **rangeCheck**. **MinMax** accepts a cell from **getCell** and returns the minimum and maximum values of the cell's four vertices. **MinMax** implements step 1a of the cell processing algorithm. This example will not explore the details of **minMax**.

Note that **rangeCheck** produces no output. Rather, since it is an open box, it "returns" a value of consistent or inconsistent. If it returns (i.e. becomes) inconsistent, meaning that a particular contour does not cross the cell, inconsistency will propagate outward from the open **rangeCheck** icon box to the entire large, closed complex box that surrounds **rangeCheck**. This complex box includes **generateLines**, which actually plots a contour line on the portion of the large, blank image corresponding to the cell, when appropriate. Thus the **generateLines** icon box will become inconsistent, and will not execute (for this particular cell). This is the desired behavior of **generateLines** for this situation.

However, assuming that the contour level is in the desired range, **rangeCheck** will remain consistent, and **classifyVerts** will use vertex classification to indicate how the contour line(s) crosses the cell, based on whether each of the four vertices of the cell is greater or less than the contour level. **ClassifyVerts** then passes this information to **generateLines**. This example will not investigate the details of **rangeCheck**.

GenerateLines takes four arguments: 1) the vertex classification from **classifyVerts**; 2) and 3) the (x,y) location within the scalar field of the top left vertex of the current cell; and 4) the image, a partly-completed contour plot. Based on the vertex classification, **generateLines** decides if any contour lines intersect the current cell. If some lines do, **generateLines** plots them on the image, completing steps 1bi and 1bii of the cell processing algorithm.

Note that **generateLines** sends the newly-modified image to two places. First, it sends the image to **join** and thence to the outgoing sequential port (at the right), so the image can be recycled to the incoming sequential port (at the left) before the iteration box executes its own contents again to process the next contour level. (*Incoming* sequential or parallel ports are those to the left side of an iteration box; *outgoing* sequential or parallel ports are those on the right side of an iteration box.) Second, its sends the image to **displayAt:**, which will display the newly-modified, partly-finished arrow plot on the screen. **GenerateLines** does this so the scientist can see each intermediate version of the contour plot as soon as **generateLines** plots the contour line(s) crossing a new cell. If only the call of Figure 4.6 to **displayContours** (which just calls **displayAt:**—see Figure 4.7) were used, the scientist would not see any intermediate versions of the arrow plot, but only the final arrow plot. This final arrow plot would only be shown when the call of Figure 4.6 to the **displayContours** occurred—i.e., only after all the computation of **processCells** had finished.

Also, at the bottom of the iteration box in Figure 4.9 is another complex box, connected to the upper, larger complex box by an inhibitory arrow. An inhibitory arrow has a hollow head and a dashed tail, and works as follows. If the upper box is consistent, the inhibitory arrow causes the lower box to become inconsistent. This is the situation in which the current contour level passes the range check, and therefore the incoming sequential port directs the partly-done contour-plot image to **generateLines**. The image cannot pass through the lower complex box because that box has become inconsistent.

On the other hand, if the upper complex box is inconsistent, the inhibitory arrow has no effect on the lower complex box, which thus remains consistent. Thus the image flowing downstream from the incoming sequential port cannot pass through the upper box (and **generateLines**), but can pass through the lower box. This is appropriate, since if the contour level failed the range check, **rangeCheck** and the entire upper box will become inconsistent, and **generateLines** should not (and will not) be allowed to modify the current image. Rather, that image will flow unchanged through the lower complex box and then to **join**.

Join produces as outputs whichever one of its inputs is present (i.e., non-nil). **Join** joins together two branches of data flow. Here, the two branches go through the upper and lower complex boxes, respectively.

4.2 Two Examples With Predefined DataVis Functions: How A Scientist Can Modify DataVis Programs

This section presents two examples to show ways that a scientist could modify or extend a DataVis predefined library program. The first example is a smaller change, while the second is a more radical modification. Both examples are at DataVis's *definition* layer. That is, the examples require the scientist to define a first-order function (either by moving around predefined DataVis library functions within an existing program or by substituting one function for another), to use higher-order functions, or to use iteration boxes that have parallel ports. However, the examples do not make the user build an entirely new DataVis function from

Figure 4.10: The DataVis **vectsToArrows** program.

the ground up, or use the most hard-to-use DataVis constructs, such as inconsistency. Both examples deal with changing the DataVis **vectsToArrows** program. Before I give the two examples, I will discuss **vectsToArrows** in detail to clarify the examples.

4.2.1 A Detailed Look At vectsToArrows

DataVis will provide functions that generate typical plots, but new functions are often needed. It will probably be common for scientists to create a new function by modifying an existing function or program. Of course, they first must understand the existing function or program. This section describes **vectsToArrows**, a program that converts a vector field into an arrow plot. Later sections will show how to change **vectsToArrows**.

Figure 4.10 is the **vectsToArrows** program. There are two input base boxes (i.e., parameters) at the left: the vector field, and the offset point for placing the arrow-display window. Each vector is fed to **vectorToArrow**, a function that transforms the vector into a small arrow image. The arrow's direction is that of the vector and the arrow's length is proportional to the magnitude of the vector.

The parallel port on the right side of the iteration box collects the individual images and produces a field of images when the iteration is done. The **displayMergedImages** function then takes the field of images and converts it into a single large image. The resulting arrow-plot image looks like Figure 4.3, but without the overlay of the contour plot.

Figure 4.11: The DataVis **vectorToArrow** function.

For **vectsToArrows** as shown in Figure 4.10, program animation can give the scientist information about the input parameters (vectorField and Offset) and about the links. When **vectsToArrows** is run, the offset point appears in Offset so the user can verify that it is correct. Further, a standard vector-field icon appears in the VectorField input base box, indicating that that parameter has a value. Also, each link flashes to a darker color and back as it transmits data across itself. (Section 3.6 described flashing of links.) Flashing of links is particularly useful in Figure 4.10 because the scientist can watch the input and output links for **vectorToArrow**. Since **vectorToArrow** is repeatedly executed inside a loop, the speed with which its links flash shows how quickly the whole **vectsToArrows** program is running.

Figure 4.11 shows a detailed view of **vectorToArrow**. **VectorToArrow** scales a field of four points representing a canonical arrow by the vector's magnitude, rotates the points by the angle representing the vector's direction, and draws the points onto a small image as an arrow.

As it did in Figure 4.2, program animation can give the scientist useful information in Figure 4.11. **VectorToArrow** is called inside a loop. Each time it is called, **vectorToArrow** displays the particular vector being converted to an arrow in the Vector input base box, calculates the small arrow image, and displays a standard image icon in the **RotatedImage** output base box. Before **vectsToArrows** calls **vectorToArrow** again, **vectsToArrows** visibly clears these two base boxes of their contents, to indicate that it will call the function without leaving any old parameter values hanging around from the previous call.

Figure 4.12: The DataVis **displayMergedImages** function.

Recall once more the implementation of **vectsToArrows** (Figure 4.10): the purpose of the **displayMergedImages** function is to take a field of small images (here, arrows), to write them all into a single large image, and to display the large image on the screen. Figure 4.12 displays the definition of **displayMergedImages**.

DisplayMergedImages calls **fieldIterator**. **FieldIterator** iterates over all the elements in a field in row-major fashion, applying the function in its function slot to each field element. As a side effect, each time **fieldIterator** applies the function to a field element, it modifies an accumulator. The accumulator could be, for example, a large blank image onto which arrows are plotted, or a sum variable for adding up the values of a scalar field. When **fieldIterator** has iterated over the field elements, it produces the modified accumulator as output.

FieldIterator is a higher-order function and possesses a function slot. Here, **display:-x:y:onto:** is the first-order function sitting in the function slot. **FieldIterator** will apply **display:x:y:onto:** to a field element—in this case, a small arrow image. **Display:x:y:onto:** writes or "displays" a field element (which it expects to be a small image) onto a larger image at a certain x index and y index. (The name **display:x:y:onto:** is the same as that of the Smalltalk function that it calls.) Essentially, **display:x:y:onto:** does much of the work of creating a single large arrow-plot image from all the individual small arrow images. (**NewImage** creates the initially-blank large image to be 800 x 800 pixels and passes it to **displayMergedImages**.)

As in the functions previously described, program animation flashes links and displays values in base boxes. But now program animation has an added feature: before **fieldIterator** is called, the icon for the **display:x:y:on:** first-order function is displayed in the function slot of **fieldIterator**'s icon box.

This concludes the detailed description of **vectsToArrows**. The next section shows how **vectsToArrows** might be customized at DataVis's *definition* layer.

4.2.2 Example One: Modifying VectsToArrows For Quicker Display

VectsToArrows, as described in section 4.1, takes a long time because the current implementation of DataVis is so slow. Although it is probably possible to make DataVis several orders of magnitude faster than it currently is, there would still be data sets and programs that would take hours to run. The following example gives one approach to ameliorating the problem of slow execution for large data sets and complex programs.

Clearly, the user would prefer not to be forced to wait a long time before seeing some output. (In **vectsToArrows** as described in section 4.2.1, the arrow plot image is not displayed on-screen until the program has almost finished running.) It would be better to display intermediate results as soon as **vectsToArrows** generates them. Specifically, it would be nice to see the large arrow-plot image displayed on-screen as soon as each new small arrow image is written onto the large image. This modification to the **vectsToArrows** of section 4.1 would be particularly useful in revealing early incorrect results to the user before the program has completely finished running. If the program produces an early wrong result, it will be apparent in the incomplete arrow-plot image on-screen, and the scientist using DataVis can kill the program and fix the bug without wasting a lot of time.

In order to display each arrow image (as part of the large arrow-plot image, of course) as **fieldIterator** writes the arrow image onto the large image, it is useful to modify **display-MergedImages** (see Figure 4.13) by putting a new first-order function into **fieldIterator**'s function slot. The user has replaced the **display:x:y:onto:** that was in **fieldIterator**'s function slot (in Figure 4.12) with **plotAndPaint** in Figure 4.13. Whereas **display:x:y:onto:** only plotted the individual arrow images onto the large arrow-plot display, **plotAndPaint** (see Figure 4.14) not only plots each arrow onto the large image, but also displays or "paints" each temporary large image onto the screen. That is, after the user has put **plotAndPaint** into

Figure 4.13: Modified version of **displayMergedImages** for displaying each arrow plot as soon as it is written onto the large arrow-plot image.

Figure 4.14: The DataVis **plotAndPaint** function for modified write-and-immediately-display version of **vectsToArrows**.

fieldIterator's function slot, he or she will see the large arrow image change as soon as each arrow is plotted onto it.

An issue now arises for the user to decide. Should the call to (i.e., icon box for) **displayAt:** in **displayMergedImages** (Figure 4.12) be deleted or not? If the the arrow-plot image generated by **vectsToArrows** should be kept around after **vectsToArrows** finishes running, then the call to **displayAt:** is useful and should be retained. (Images dumped directly to the screen by **plotAndPaint** cannot be saved.) On the other hand, if the arrow-plot image does not need to be saved, then the call to **displayAt:** may be deleted. For purposes of this example, I will assume that the user wants to save the arrow plot and therefore will not delete the call to **displayAt:**. (Section 6.3 mentions *viewing monitors*, which could be a better way to display images at run-time, and which would not require changing the DataVis program by adding a function like **displayAt:**.)

The changes made to **vectsToArrows** to get the write-and-immediately-display version are simple conceptually, but involve several commands. Most of the commands are needed to deal with housekeeping chores necessary to modify an existing procedure that has extant calls.

The user must remove **display:x:y:onto:** from **fieldIterator**'s function slot and replace it with **plotAndPaint**. This change to the original **displayMergedImages** of Figure 4.12 will produce the modified version of **displayMergedImages** shown in Figure 4.13. The user will first give the `clear function slot` command to remove **display:x:y:onto:** from the function slot, followed by the command `add proc to func slot` to put **plotAndPaint** into the slot.

4.2.3 Example Two: Extending VectsToArrows To Create ScalsToImage

The previous example showed how to make a fairly minor modification to the DataVis **vectsToArrows** program. This example is a more radical extension of **vectsToArrows** (that is, an extension of the original version in sections 4.1.1 and 4.2.1, not an extension of the version produced by the previous example). In fact, the extension is radical enough to justify changing the name of the modified top-level program from **vectsToArrows** to **scalsToImage**.

This example extends **vectsToArrows** to handle a scalar field instead of a vector field, and to produce a color image instead of a black-and-white arrow plot. The new program, **scalsToImage**, produces the color image by coloring pixels based on the magnitude of scalars in the scalar field.

Figure 4.15: Extending **vectsToArrows** of Sections 4.1.1 and 4.2.1 to become **scalsToImage**.

ScalsToImage looks similar to **vectsToArrows**, so I will not show it here. **ScalsToImage** has two main differences from **vectsToArrows** (recall Figure 4.10). First, **scalsToImage** does not contain an iteration box and does not call **vectorToArrow**. Second, **scalsToImage** calls **displayMergedPixels**, not **displayMergedImages**.

DisplayMergedPixels is shown in Figure 4.15. **DisplayMergedPixels** is a modified version of **displayMergedImages** (Figure 4.12). This modified version gives a different first-order function to **fieldIterator** than the original **displayMergedImages** (Figure 4.12) did. While **displayMergedImages** put **display:x:y:onto:** into **fieldIterator**'s function slot, **displayMergedPixels** places **pixelOn:x:y:in:** there. This small change has a major impact on extending the behavior of **vectsToArrows**.

Another difference between **vectsToArrows** and **scalsToImage** is that the **displayMergedPixels** input base box ScalarField in Figure 4.15 accepts a scalar field of color-table indices, not a vector field (recall Figure 4.10). Each element in this scalar field is an index to a value of a colored pixel.

There are two more differences in **scalsToImage**. First, somewhere there must be DataVis code to convert the original scalar-field data—different from the field of color-table indices given to one of **scalsToImage**'s input base boxes—into the field of color-table indices. I will not discuss the particulars of that conversion code since they are not pertinent to this example.

Figure 4.16: An alternate implementation of **displayMergedImages**, using a complex box to sequentialize execution, instead of bogus links.

Second, **scalsToImage** uses (i.e., calls) **passThruFieldIterator**, a modified version of **fieldIterator**. **PassThruFieldIterator** is just like **fieldIterator**, but it passes its first input (a palette—a color table) through itself, unchanged, to become its first output.

The purpose of using **passThruFieldIterator**, instead of **fieldIterator**, is just to sequentialize **preparePalette**, **fieldIterator**, and **restorePalette**. That is, I first want to prepare the hardware palette (i.e., color table) for the execution of **passThruFieldIterator** by loading a rainbow (i.e., ROY G. BIV) palette into the hardware palette, then to execute **fieldIterator**, and then to restore the palette to its original value. Thus, in Figure 4.15, I have introduced two extra links (from **preparePalette** to **passThroughFieldIterator**, and from **passThrough-FieldIterator** to **restorePalette**), to ensure that these three functions execute in the proper order. These two links are "extra" in the sense that the links do not represent any data dependencies. Specifically, **passThruFieldIterator** does not need data from **preparePalette**, and **restorePalette** does not need data from **passThruFieldIterator**. Rather, **restorePalette** needs data from **preparePalette**, so these two functions should be directly linked.

An alternate implementation of **displayMergedImages**, which directly links **preparePalette** to **restorePalette**, is shown in Figure 4.16. This alternate implementation achieves sequential execution of **preparePalette**, **passThruFieldIterator**, and **restorePalette** by putting the complex box Box1 around **fieldIterator**, and by running the new direct link from **preparePalette** to **restorePalette** across Box1. In this way, **restorePalette** will not execute until **fieldIterator** has executed, since the incoming link to **restorePalette** crosses Box1 will not be able to transmit data, if Box1 has become inconsistent.

In either implementation, **FieldIterator** will have the first-order function **pixelOn:x:y:in:** in its function slot. **FieldIterator** will apply **pixelOn:x:y:in:** to each field element, at the current x and y indices, and with a large, blank image which will become the desired color image. **PixelOn:x:y:in:** will set the pixel in the large image at location (x,y) to the value in the color table at the index given by the field element.

To extend the original **displayMergedImages** of Figure 4.12, that was called by **vectsToArrows** (Figure 4.10), and to make **displayMergedImages** into **displayMergedPixels** (Figure 4.15), that can be called by **scalsToImage**, give the `rename` command for the first (i.e. topmost) base box to change its name to ScalarField. In the procedure box for the original **displayMergedImages** (see Figure 4.12), give the `clear function slot` command for **fieldIterator**, followed by `delete` to remove **fieldIterator**, and then succeeded by `add procedure` to add **passThruFieldIterator**, and finally followed by the `add proc to func slot` command to put **pixelOn:x:y:in:** into **passThruFieldIterator**'s function slot. (**PixelOn:x:y:in:** is a predefined DataVis library function.) Next, give the **add procedure** command twice to add **preparePalette** and **restorePalette**, and add the links to connect them to **passThruFieldIterator**. Next, give the `define procedure` command once in the **displayMergedImages** procedure box to define it as a procedure. Lastly, use the `add procedure` command to put a call to **displayMergedImages** into the procedure box for **scalsToImage** wherever the call is needed. Although the series of commands just described may seem complex, they are not, and making the changes just described is fairly easy.

4.3 Using The Expert-Programming Level

DataVis's most difficult features or language constructs include newly-defined higher-order functions, iteration boxes used with sequential ports or global inputs, and inconsistency. The preceding examples have shown how to use higher-order functions and iteration boxes with parallel ports, as well as how to define first-order functions. Therefore this section will give two examples showing how a scientist can create a new DataVis function "from scratch"—that is, using DataVis's most difficult constructs. The examples will cover the constructs other than newly-defined higher-order functions, since the commands needed to define a higher-order function are very similar to those to define a first-order function. The first example deals with iteration boxes that use sequential ports; the second, with inconsistency. Both examples are at DataVis's *expert programming* layer.

4.3.1 Example One: Using Iteration Boxes

Suppose the scientist decides to create the DataVis **contoursInterval** function for making an interval (i.e., sequence) of increasing integers from a lower bound through an upper bound. **ScalsToContour**, the example program of section 4.1.2, called **contoursInterval** (in Figure 4.6) to create an interval or sequence of integers that could be given as an input to **processCells**, in order to create the contour plot. (The user can also think of **contoursInterval** as a function which creates a scalar field containing the desired integers.) The user may decide to make the lower bound, upper bound, and increment be parameters to the function.

To begin, open a DataVis window and add a procedure box to it. Then give the `name procedure` command. This command lets the user give a textual name and permits, but does not require a graphical bitmap name as well. Next, add four base boxes: three for input, one for output. Clearly, iteration is required, so add an iteration box. The results so far may be seen in Figure 4.17.

To create an interval or sequence of integer scalars, add a pair of sequential ports, hooked up in a circuit of links including the DataVis '+' primitive. To show what integer will be added to the sequence at any time, add an empty number box to the left side of this circuit. To give the circuit a starting value, align the sequential ports to be horizontal with the Lower Bound base box when the ports are first added. (See Figure 4.18.) Also, add a parallel port and the

Figure 4.17: Partially-constructed DataVis **contoursInterval** function.

Figure 4.18: Creating a new DataVis function to make an interval for a contour plot: **contoursInterval**.

link to it in order to agglomerate all the individual integer scalars into an interval (i.e., a scalar field).

Next, add an open box containing the predefined predicate '<=' and hook it up to Upper Bound and the empty number box. This open '<=' box provides a termination condition for the loop: when the current number in the sequence exceeds the (upper) limit, the '<=' box will become inconsistent. In turn, so will the iteration box, and iteration will stop.

Figure 4.18 shows the results of giving the `define procedure` command. The call to **contoursInterval** in figure 4.6 can be created by giving `add procedure`, by adding a few number

Figure 4.19: Call to DataVis **contoursInterval** function.

boxes, and by entering ten and one into two of them. After the call in Figure 4.19 is run, Box3 will contain the interval `(1 2 3 4 5 6 7 8 9 10)`.

4.3.2 A Detailed Explanation Of VectsToArrows

Before I present the second example, which deals with the implementation of **vectsToArrows**, I will explain the implementation of **vectsToArrows** in some detail.

Recall that the original **vectsToArrows** of Figure 4.10 calls **displayMergedImages** (see Figure 4.12), passing a field of small arrow images and an offset point to **displayMergedImages**. Next, **displayMergedImages** calls **fieldIterator**, which iterates over the field elements (in this case, arrow images) and produces a modified accumulator (here, a large arrow-plot image). (I will explain the idea of the accumulator more in a moment, when I discuss **fieldIterator**'s procedure box.) **DisplayMergedImages** then hands this arrow-plot image and the offset point to **displayAt:**, which displays the arrow plot on the screen at the offset from the current window.

As **fieldIterator**'s icon box shows (see Figure 4.12), **fieldIterator** is a DataVis higher-order function and possesses a function slot. In this situation, **display:x:y:onto:** is the first-order function sitting in the function slot. **FieldIterator** or one of its callees will apply **display:x:y:onto:** to each field element. **Display:x:y:onto:** plots a small image onto a larger image at a certain x index and y index.

The procedure box for **fieldIterator**, shown in Figure 4.20, contains an apply box (the box labeled "FunctionToApply"). The apply box can be thought of as a function slot. This slot corresponds to the slot in **fieldIterator**'s icon. When **displayMergedImages** calls **fieldIterator**, the **display:x:y:onto:** function is transmitted into **fieldIterator**'s apply box. **Display:x:y:onto:** can be thought of as a function parameter to **fieldIterator**, albeit one that

Figure 4.20: The DataVis **fieldIterator** function.

is not passed via the usual mechanism of arrows coming in to **fieldIterator**'s icon box and of matching input base boxes in **fieldIterator**'s procedure box.

Figure 4.20 shows that **fieldIterator** takes two input parameters (in addition to its function slot): a field, and a large, unmodified accumulator. **fieldIterator** iterates over the field in row-major, top-to-bottom order. At each field element, it applies the function in its function slot to the field element. Each application of this first-order function modifies the accumulator in some fashion. After the function has been applied at all field elements, **fieldIterator** outputs the resulting modified accumulator.

FieldIterator's second input and its output are called accumulators, rather than images, to reflect the idea that **fieldIterator** may be called with any accumulator, not just a large image. (For example, the accumulator could be a variable that would sum up the values in a scalar field.) In this case, the accumulator is a large, initially-blank image. Every time **fieldIterator** applies **display:x:y:onto:** to a small arrow image, **display:x:y:onto:** plots the arrow image onto the large accumulator image. Eventually, the accumulator becomes the desired arrow-plot image. In addition to all these other things that happen at runtime when **displayMergedImages** calls **fieldIterator**, program animation displays **display:x:y:onto:**'s icon in the function slot of **fieldIterator**'s icon box.

The most important part of **fieldIterator** is inside the double-walled iteration box that is visible in Figure 4.19. Links enter the iteration box in Figure 4.19 through a parallel port and through sequential ports.

At the top left of the function, the field of images feeds into the iteration box through a parallel port. The field of images can be thought of as a 2D array of images. A parallel port disaggregates a field into its elements—here, arrow images—and passes one element into the iteration box at the start of each iteration. The number of iterations is the number of elements passed into the parallel port. The contents of the iteration box be executed once for each field element as the elements pass through the parallel port, one at a time.

At the top center of the iteration box, FunctionToApply is applied once for each element of the field, with the element, its X and Y coordinates, and the old value of the accumulator, and with the new value of the accumulator as the output.

The bottom half of the iteration box contains a call to the **iteratorDriver** function. Iterating over the elements of a field often requires knowing the location of each element. **IteratorDriver** provides these locations.

Thus, the locations produced by **iteratorDriver** flow to the apply box (labeled "FunctionToApply"). There the locations tell the **display:x:y:onto:** function where to display each small arrow image on the large image. The first-order **display:x:y:onto:** function is transmitted from the function slot in **fieldIterator**'s icon box (recall Figure 4.12) into the function slot in the apply box (labeled FunctionToApply) at the top of the iteration box. **Display:x:y:onto:** is applied to four pieces of data: a field element, the current x and y indices, and the large, formerly-blank image. **Display:x:y:onto:** writes or "displays" the field element (i.e., a small arrow image) onto the large image with the arrow image's top left corner at pixel (xIndex, yIndex). The large image is cycled through the uppermost pair of sequential ports. When the iteration is done, the output base box ChangedAccumulator holds the modified, completed large arrow-plot image.

The **filter** function, at the top left of Figure 4.20, filters out special #endOfRow symbols from the field of images. Each field with two or more rows has such symbols. They indicate the end of a row in the field. **Filter** gets rid of them so that **iteratorDriver** will not use up a legitimate (xIndex, yIndex) pair on each of them, only to find that #endOfRow is not displayable on the large image.

Figure 4.21: The DataVis **iteratorDriver** function.

4.3.3 Example Two: Using Inconsistency

This example shows how to create the **iteratorDriver** function, which was called both in **fieldIterator** (see Figure 4.20) as part of the **vectsToArrows** program of Sections 4.1.1 and 4.2.1, and in the previous example to this subsection (i.e., in the example in section 4.3.2). Figure 4.21 displays **iteratorDriver**. To create **iteratorDriver**, first add and rename the five base boxes: three for input, two for output. (They need to be renamed because they will initially be given default names such as Box1, Box2, etc.) Then consider the two most important data paths through **iteratorDriver**. The first major data path goes from the CurrentXIndex input base box to the NextXIndex output base box, and produces the next x index. The second major data path goes from CurrentYIndex to NextYIndex, and generates the next y index.

Important input parameters to the task of generating the next (x,y) indices include the width and height of each field element, and the number of columns in the field. The predefined DataVis function **elementSize** gives the size of any field element as a point: (width, height). (All elements of a certain field must have the same width and height.) The predefined **x** and **y** functions extract the point's x and y coordinates for use as width and height of a field element. The predefined **cols** function gives the number of columns in the field.

Figure 4.22: The DataVis **resetAtEnd** function with inconsistency but without inhibitory arrows.

Supposing that the field elements (small arrow images) are 40 pixels x 40 pixels and there are 10 columns in the field, the desired sequence of indices, expressed as points (x,y), is: 0@0, 40@0, 80@0, ..., 360@0, 0@40, 40@40, ..., 360@40, ..., 0@360, ..., 360@360. (In Smalltalk on the Tektronix 4319, the origin of display coordinates is at the top left of the screen and y coordinates increase as one moves downward.) Two characteristics of this sequence are evident: when the end of a "scan line" (i.e. row) is reached, x should be reset to zero; and if the new x index is zero, y should be incremented by the height of a field element.

The DataVis functions **resetAtEnd** and **addWhenZero** implement these two characteristics of the sequence of (x,y) indices to be generated. These two functions are similar; both require inconsistency.

ResetAtEnd takes three inputs: the old x index, incremented by the width of a field element; the width of a field element; and the number of columns in the field. Since **resetAtEnd** produces the next x index, it must decide whether to pass through the incremented x which it is given, or to reset x to zero because it has reached the end of a scan line (i.e., row). **ResetAtEnd** is at the end of a row if the incremented x $>$ element width $*$ (number of columns - 1). For example, in the sequence mentioned earlier, **resetAtEnd** has reached the end of row when the incremented x $> (40 * 9) = 360$.

Figure 4.22 shows one way to use inconsistency to implement **resetAtEnd**. If x $<=$ width $*$ (columns - 1) then the '>' box will become inconsistent and so will Box12, thus allowing the

Figure 4.23: The DataVis **resetAtEnd** function with inconsistency and inhibitory arrows.

incremented x index to pass through to **join**. If the opposite is true, the ‘<=’ box and Box11 will become inconsistent, and zero will flow through to **join**. **Join** is a predefined DataVis function that does not require that both of its inputs be present before it fires (i.e. executes). In fact, **join**'s output is whichever one of its two inputs is present.

Figure 4.22 gets the job done, but is awkward and ungainly, particularly for computing something that would be a simple IF statement in most textual languages. Inhibitory arrows can improve and clarify Figure 4.22's implementation of **resetAtEnd**. Inhibitory arrows were discussed in section 3.2. If the source box for an inhibitory arrow is consistent, the destination box will be inconsistent. If the source box is consistent, the arrow will have no effect. So inhibitory arrows propagate inconsistency, whereas other arrows (i.e., links) in DataVis propagate data and type information.

Figure 4.23 shows a more concise, streamlined implementation of **resetAtEnd**, using inhibitory arrows. Only one inhibitory arrow is needed. It goes from Box11 to Box12. If x <= width ∗ (columns - 1) then Box11 will be consistent, the inhibitory arrow will make Box12 become inconsistent, and the x index will flow through to **join**. In the other situation, Box11 will be inconsistent, Box12 will be consistent, and zero will flow through to **join**.

This chapter has shown how a scientist would use predefined DataVis programs, would modify or extend DataVis programs using predefined functions, and would create new DataVis functions and programs using the full panoply of basic DataVis constructs (higher-order functions, inconsistency, and iteration boxes with sequential ports or global inputs). The examples show that DataVis is easy to learn, use, and modify.

Chapter 5

Semantics of DataVis

This chapter describes the semantics of DataVis. The first section describes how to represent a DataVis program; the second describes the meaning of executing a DataVis program without loops, inconsistency, or higher-order functions; the third describes inconsistency; the fourth describes loops (i.e., iteration); and the last describes higher-order functions. Names of functions and domains will be capitalized, while lowercase letters will mean an element of a function's domain.

5.1 Representing DataVis Programs

DataVis can be described by three primitive domains and a number of complex ones. The three primitive domains are Boxes, Links, and Values.

Values describes the values that flow along the links of a program as the program executes, so it will include scalars, vectors, points, images (i.e. bitmaps), and fields. This chapter ignores the structure of elements of Values and treats it as a primitive domain out of which others are built.

The State of a DataVis program is an assignment of values to links, i.e. State: Links $\rightarrow$ Values. Usually a state only assigns values to some of the links in a program. Executing a program means assigning values to more and more of the links. Typically, the value on a link is never changed once it is assigned, although I will show later how inconsistency can disrupt this trend.

Figure 5.1: Bounded sequential iteration in Show and Tell [Kimu86a].

Another complex domain is Behavior: ValueSequence $\rightarrow$ ValueSequence, which represents the behavior of each box. A behavior is a function from the values of the box's input ports to the values of the box's output ports. A program will assign a behavior to each of its boxes. Many of the behaviors are primitive, such as the behavior of arithmetic operations, and I will ignore them. However, I will later show how to calculate the behaviors of a user-defined procedure and of an iteration box, which are more interesting.

The final domain is Program. A program p has many components. p.boxes are the boxes mentioned explicitly by the program. p.boxbehavior is a function from a box to its behavior. p.connected is a function from a box and a port to the link connected to that box at that port. p.inputports is a function from a box to its set of input ports. p.outputports is a function from a box to a sequence of its output ports. p.inbaseboxes is a sequence of input base boxes in p that represents the interior of a procedure box, and p.outbaseboxes is a sequence of output base boxes for p. Base boxes act as the input and output parameters of user-defined procedures.

p.connected may deal with an input or output port or with a sequential or parallel port, since p.connected gives the link connected to a non-iteration box at a certain input or output port, or to an iteration box at a sequential or parallel port. An input or output port is different from the sequential ports (the two small pairs of small triangles in Figure 5.1) or the parallel ports (the two small rectangles on either side of the iteration box in Figure 5.2) which were discussed in Section 2.2.3 (Show and Tell). An input or output port is invisible, and is not associated with iteration or an iteration box. Instead, such a port provides an area of a box

Figure 5.2: The DataVis **vectsToArrows** procedure box.

Figure 5.3: Ports for attaching links to or from a box: two input ports and one output port.

to which a link may be attached. Boxes are divided into two halves, with input ports on the left half and output ports on the right half. Figure 5.3 shows a '$*$' box with two input ports and one output port. Only one link may be attached to each input port, while multiple links (all carrying the same value, of course) may be attached to the '$*$' function's sole output port. (DataVis also allows a function to produce multiple, different outputs. Such a case requires multiple output ports.)

For example, the program in Figure 5.4 can be represented in terms of the three sets and three functions belonging to p—an element of the Program domain—as follows. p.boxes = {Box1 Box2 Box3 Box4 Box5}. [1] It does not include Counter or Test, because they are inside the iteration box Box3 and are considered part of its program. Box3 will have another program

[1] A side note: the program in Figure 5.4 generates a field or sequence of five ones in Box4. Box4 displays the standard icon for a number box holding a scalar field. Also, Link2 is a global input to the iteration box. Finally, the pair of sequential ports, along with the boxes Box1, Counter, and Box3, counts from six down to zero. The Counter box is the DataVis *decrement* primitive.

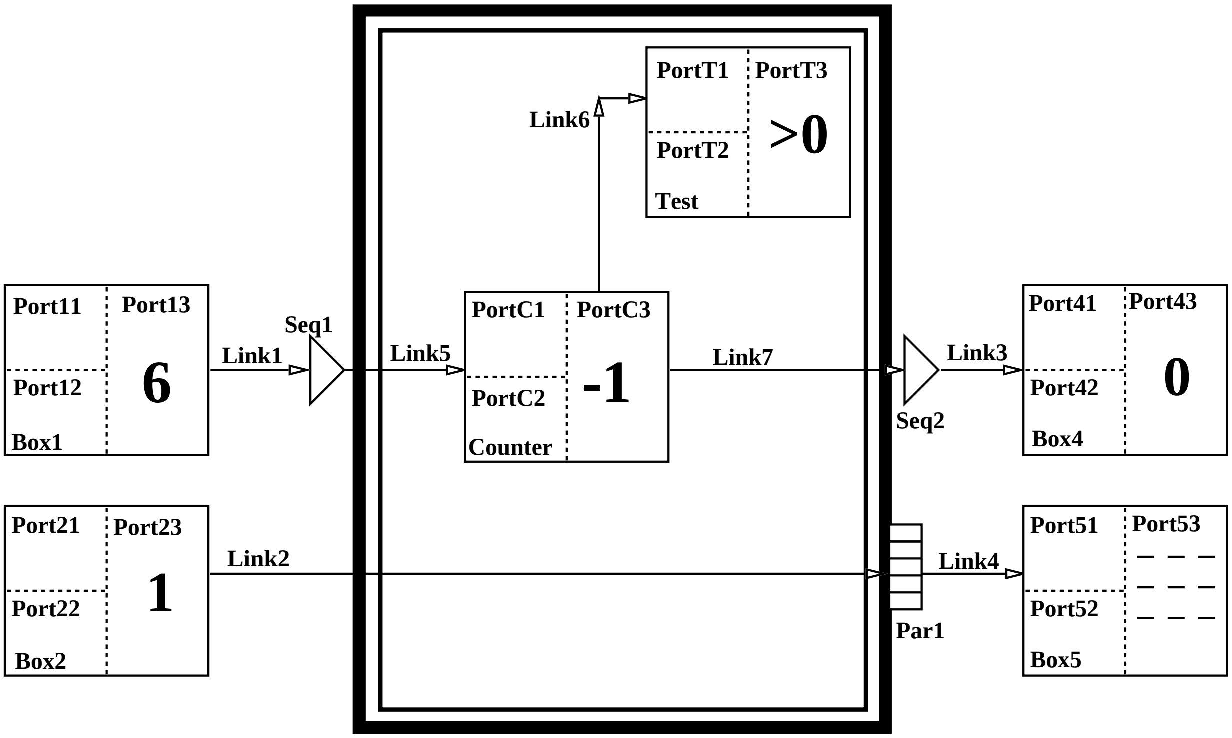

Figure 5.4: Bounded sequential and parallel iteration, with input/output ports and all objects labeled.

p′ that represents its interior. That is, a typical DataVis program consists of a hierarchy of elements of the Program domain. One element of the Program domain represents the overall DataVis program. Every DataVis box that contains another box is represented by its own element of the Program domain. For example, a complex box (Figure 5.10 shows two large complex boxes, connected by a dashed arrow), iteration box (Figure 5.1 shows one), or procedure box (Figure 5.2 displays one) each has its own element of the Program domain.

p.connected = {(Box1, Port13, Link1), (Box3, Seq1, Link1), (Box2, Port23, Link2), (Box3, Seq2, Link3), (Box4, Port42, Link3), (Box3, Par1, Link4), (Box5, Port51, Link4)}. Since Box1, Box2, Box4, and Box5 are just number boxes, the behavior produced by p.boxbehavior for them is a constant behavior. Thus I will not investigate those behaviors farther. For the iteration box, p.boxbehavior will produce a behavior that takes the current state, executes the iteration box until it becomes inconsistent, and adds the values of the two sequential ports on the right to the original state. (Section 5.4 will explain in greater detail how an iteration box is executed.)

Finally, the functions p.inputports and p.outputports return a sequence of input and output ports, respectively, of a box in p.boxes. Thus we have p.inputports = {(Box1, (Port11, Port12)), (Box2, (Port21, Port22)), (Box3, ()), (Box4, (Port41, Port42)), (Box5, (Port51, Port52))} and p.outputports = {(Box1, (Port13)), (Box2, (Port23)), (Box3, ()), (Box4, (Port43)), (Box5, (Port53))}. DataVis functions can produce two or more distinct output values, although all the primitives in Figure 5.4 (such as -1) produce just one output value. DataVis functions returning two or more distinct values will have two or more output ports. This makes the fact that p.outputports produces a *sequence* of output ports more meaningful. I will further discuss why p.inputports and p.outputports return a *sequence* of ports in the part of Section 5.2 that deals with parameter passing.

Note that Figure 5.4 labels links, input/output ports, and sequential and parallel ports and makes input/output ports visible. Regular DataVis programs, however, have no labels on links and ports, and contain only invisible input/output ports.

5.2 Executing DataVis Programs

To execute a DataVis program (specifically, a DataVis procedure box—a function definition), we need a program and an initial value sequence. The initial value sequence provides a value for all number boxes that act as inputs. Number boxes hold a single data object of one of the types mentioned in Section 3.6 (Type Checking): for example, a vector field, or a single scalar number (either integer or real). In Figure 5.4, the two number boxes at the left, containing six and one, act as inputs to the rest of the program. These two boxes must have their values entered by the user before program execution begins.

Since the initial sequence means all number boxes acting as inputs have a value, it follows that the links exiting those number boxes also have values before execution begins.

An important property of the connected function is that a DataVis program is an acyclic graph. (Unless otherwise noted, all statements in this section about executing DataVis programs also hold true for DataVis's procedure boxes.) Iteration boxes do not introduce any cycles into the graph because they are considered single nodes in the graph. Since programs are acyclic, the boxes in a DataVis program (that is, the nodes in the acyclic graph) can be topologically sorted. In fact, the prototype of DataVis executes programs by doing a topological sort on all boxes in the program, except for number boxes acting as inputs. (The user enters a value for each such box, and thus the program does not have to calculate a value for these number boxes.)

DataVis topologically sorts boxes in the program to determine the order in which boxes will execute. Execution of a box corresponds to the firing of a node in the pure data flow model (discussed in Section 2.1.1.1). Specifically, the box checks that all of its input links have data on them, it executes using that data, and it places results on its output link or links. It is important that boxes execute in the proper order. For example, Figure 5.5 displays the **vectorToArrow** function. In **vectorToArrow**, the output of **scaleAndTranslate:** is the input of **rotate:**, so **scaleAndTranslate:** must execute before **rotate:**, and similarly **rotate:** must execute before **drawArrow**. Thus, if **scaleAndTranslate:**, **rotate:**, and **drawArrow** do not execute in the order just mentioned, the function will not produce the correct result.

The topological sort begins by finding all boxes in **vectorToArrow** with no output links. The only such box is **RotatedImage**. It is added to a list of sorted boxes. Its input links are marked. The sort then finds all boxes with no unmarked output links (**drawArrow**), and adds them to the list. This algorithm repeats until all boxes are accounted for, and results in a list of sorted boxes: **RotatedImage**, **drawArrow**, **rotate:**, **direction**, **scaleAndTranslate:**,

Figure 5.5: The DataVis **vectorToArrow** function.

magnitude:, **merge**. To execute **vectorToArrow**, DataVis makes one pass in reverse order over the list of sorted boxes, telling each box to execute itself as the pass moves along.

To recapitulate, a property of Connected is that a DataVis program may be viewed as an acyclic graph. Therefore, the boxes in the program (nodes in the graph) can be topologically sorted. This implies that DataVis can calculate a set of input links that must have values before program execution begins. Among the items that are necessary to execute a program is an input sequence. This input sequence is a value sequence that is defined exactly for the set of input links just mentioned.

A valueSequence is either a sequence of values or *failure*. That is, if the icon box for the DataVis function being executed becomes inconsistent, the function has failed, and it does not return a sequence of values. Rather, its valueSequence is *failure*. Failure of an icon box has two effects: first, the icon box sends no result values downstream on its output links; second, if the icon box is open (i.e. has a dotted outline), the icon box's inconsistency will propagate outward and thus will affect other parts of the program. Section 5.3 will discuss inconsistency.

Executing a DataVis program means executing each box in the program, in topological order. Executing a box modifies the state by placing values on links. Eventually every box will have executed, resulting in the end state.

There are two ways to get a start state. A top-level program has a start state that assigns no values to links. That is, it is the empty function from links to values. All number boxes in top-level programs are constants, and the user must enter a value into number boxes (i.e., edit the program) before executing the program. However, a program for a user-defined function has base boxes (i.e., parameters) that act as its inputs and outputs. Therefore, the start state of a program for a user-defined function must map output links of the input base boxes to the corresponding input values. Also, the output values of a user-defined function must be extracted from its ending state so that they can be placed on the links of the program that called it.

Execution is defined by two functions. Execute defines the execution of one procedure box (i.e., a function definition). Execute takes a program and a sequence of starting (i.e., input) values to the procedure, and produces an ending (i.e., output) value sequence that is the result of executing the procedure with those starting values as input. ExecuteList takes a list of boxes that are inside a procedure box, a start state, and a program, and executes each box in turn on the state, producing an end state.

Execute and ExecuteList and their major subsidiary functions use "start" and "end" in the names of objects (i.e. states and sequences; for example, "startState" or "endSequence"), and use "Initial" and "Final" in the names of functions that produce such objects (e.g. "InitialState" or "FinalSequence"). The purpose of these names is to distinguish objects from the functions that make them.

ExecuteList's definition is:

ExecuteList: BoxList $\times$ State $\times$ Program $\rightarrow$ State

```
λboxList, startState, p.
   If boxList = ∅ then startState
   else {ExecuteList(tail(boxList),
                     FinalState(
                        startState, p, head(boxList),
                        (p.boxbehavior(head(boxList))(InitialSequence(
                                                      startState, p,
                                                      head(boxList),
                                                      p.inputports(head(boxList))))),
                     p)
        }
```

ExecuteList computes the state that results from executing the first box in the boxList and then calls itself recursively on the rest of the boxes in the boxList.

InitialSequence returns the sequence of input values for the first box, p.boxbehavior returns the sequence of output values that the first box produces, and FinalState returns the state resulting from placing these values on the output links of the first box. InitialSequence computes the sequence of input values by picking from startState only the values that are input links to the box. The definition of InitialSequence is:

InitialSequence: State $\times$ Program $\times$ Box $\times$ PortSequence $\rightarrow$ ValueSequence

```
λstartState, p, box, inputPorts.
   {if inputPorts = nil then nil
    else cons(startState(p.connected(head(inputPorts), box)),
              InitialSequence(startState, p, box, tail(inputPorts)))
    }
```

ExecuteList uses p.boxbehavior to get the behavior of the current box. ExecuteList then applies the box's behavior to the start sequence for the box, producing an end sequence. That is, ExecuteList executes the box.

ExecuteList calls FinalState to convert the end sequence into endState. FinalState copies the values in the end sequence to the box's output links, and returns the resulting end state. It iterates over the output ports with a helper function, FS. FinalState assumes that the order of the end sequence is the same as the order of the sequence of output ports. Its definition is:

FinalState: State $\times$ Program $\times$ Box $\times$ ValueSequence $\rightarrow$ State

```
λstartState, p, box, endSequence.
   {FS(startState, p, box, p.outputports(box), endSequence)}
```

FS: State $\times$ Program $\times$ Box $\times$ PortSequence $\times$ ValueSequence $\rightarrow$ State

```
λstartState, p, box, outputPorts, endSequence.
   {λlink. if outputPorts = nil then startState(link)
           else if p.connected(head(outputPorts), box) = link
              then head(endSequence)
           else FS(startState, p, box, tail(outputPorts),
              tail(endSequence)) (link)}
```

The p.boxBehavior function gives the behavior of every box in the program p. If a box b calls a primitive procedure then p.boxbehavior(b) returns a primitive behavior. However, if b is an icon box for a first-order user-defined function, an iteration box, or an icon box for a higher-order function, then p.boxbehavior(b) must be defined in terms of the structure of b. If b is an icon box that calls a first-order user-defined function then its behavior depends on the program for the interior of the function's procedure box. Thus p.boxbehavior(b)= UserDefinedFunction(p.internalProgramFor(b)).

InternalProgramFor is a new component of a program. This new function, given an iteration box or an icon box, returns the Program domain-element that represents the interior of the iteration box or the interior of the procedure box corresponding to the icon box. The new function's type is InternalProgramFor: Box $\rightarrow$ Program.

UserDefinedFunction produces a behavior: a function that accepts a sequence of input values, creates a start state, calls ExecuteList to run the user-defined function—thus producing an end state—and returns endSequence, a sequence of output values that was built from endState.

The definition of UserDefinedFunction is:

UserDefinedFunction: Program $\rightarrow$ Behavior

```
λp.
   {λinputSequence.
       {Execute(p, inputSequence)
        }
    }
```

This definition means that a user-defined function depends on the program p—the program for the interior of the procedure box. Among other components, p contains p.boxes, which is the set of boxes inside the procedure box. For example, **vectorToArrow** in Figure 5.6 contains the set of boxes **new**, **magnitude**, **direction**, **merge**, **scaleAndTranslate:**, **rotate:**, and **RotatedBitmap**.

Execute's definition is:

Execute: Program $\times$ ValueSequence $\rightarrow$ ValueSequence

```
λp, startSequence.
   {FinalSequence(p,
                 ExecuteList(TopSort(p.boxes, p.connected),
                             InitialState(p, startSequence, p.inbaseboxes),
                             p),
                 p.outbaseboxes)
    }.
```

The Execute function describes how to execute a DataVis procedure box (i.e. the definition of a DataVis function). Execution needs information about the program that represents the interior of the procedure box, and about the starting value sequence (that is, the sequence of values of those number boxes that have starting values). During execution, DataVis makes one pass over the boxes in topological order, applying the behavior of each box to the current state to obtain a new state.

InitialState is a function that converts a starting sequence of input values given to the DataVis procedure box into a state that maps the links that come out of input base boxes to the input parameters for these base boxes.

The definition of InitialState is:

InitialState: Program $\times$ ValueSequence $\times$ BoxSequence $\rightarrow$ State

```
λp, startSequence, inputBaseBoxes.
   {λlink. if inputBaseBoxes = nil then ∅
     else if (∃port ∈ p.outputPorts(head(inputBaseBoxes))
              and p.connected(box, port) = link)
        then head(startSequence)
        else InitialState(p, tail(startSequence),
                tail(inputBaseBoxes)) (link)
    }
```

Execute uses ExecuteList to calculate the effect of the function on the start state. As part of this execution, ExecuteList computes the end state. FinalSequence converts the end state into an end sequence by extracting the endSequence from values of incoming links to the procedure box's output base boxes.

The definition of FinalSequence is:

FinalSequence: Program $\times$ State $\times$ BoxSequence $\rightarrow$ ValueSequence

```
λp, state, outputBaseBoxes.
   {if outputBaseBoxes = nil then nil
    else cons(state(p.connected(head(outputBaseBoxes), port)
                 where port ∈ p.inputports(head(outputBaseBoxes))),
              FinalSequence(p, state, tail(outputBaseBoxes)))
    }
```

It is important that a behavior for a user-defined function or, indeed, for any DataVis box, have the same number of inputs and outputs, respectively, as its box. So for any *legal program* p, p.boxbehavior(x) must assume as many inputs as p.inputports(x) has, and as many outputs as p.outputports(x) has. In the next section, we will see that p must also possess the *no inconsistent input boxes* property.

It is important to note that Execute bypasses the base boxes (i.e. parameters)—both input and output—of a procedure box. The reason for noting this is to justify the fact that Behavior is a function from a value sequence to a value sequence, even though input (output) base boxes have no incoming (outgoing) links. There are three ways in which Execute bypasses a procedure box's base boxes. First, Execute calls ExecuteList with Execute's p.boxes. Execute's p.boxes does not include input base boxes (although it does include output base boxes). Thus, input base boxes are not in ExecuteList's boxList, and it ignores them.

Second, InitialState builds a start state that assigns values to outgoing links of input base boxes. In this way InitialState bypasses the input base boxes.

Third, FinalSequence creates an ending sequence from endState by copying the values of incoming links of output base boxes to the sequence. So FinalSequence bypasses the output base boxes.

Page 93 discusses the semantic function Execute, which describes the meaning of executing a DataVis function (i.e., a procedure box), rather than the meaning of executing an entire DataVis program. The meaning of an overall DataVis program (as opposed to just a procedure box) is given by the OverallExecute semantic function.

OverallExecute's definition is:

OverallExecute: Program → State

```
λp.
   {ExecuteList(TopSort(p.boxes, p.connected), ∅, p)
    }
```

OverallExecute executes a DataVis program by topologically sorting the boxes in the Program, and by executing each box in turn, starting with the empty state but having each box use the resulting end state of its predecessor.

The above definition of OverallExecute is quite different from the definition of Execute on page 93. Whereas Execute uses value sequences for its inputs and outputs, OverallExecute works with states. Why does this difference exist? It is appropriate to consider sequences of input and output values for Execute, since the input and output base boxes are obviously the inputs and outputs of the function.

However, a DataVis program does not have input or output base boxes. Thus, it is not clear which set of number boxes in a DataVis program would be the analog to input or output sequences of parameters in a procedure box.

Also, when considering a DataVis program's possible analogs to a procedure's output sequences, it is hard to precisely identify the program's outputs. Admittedly, we can identify an ending set of number boxes in a DataVis program that receive their values after most other boxes. (This set includes exactly those boxes with no outgoing links.) However, it is debatable whether only this ending set of boxes constitutes the outputs of a DataVis program. Intermediate number boxes that are not in the ending set may have been given new values during execution of the program. So in some sense, such intermediate boxes are also outputs of the program.

For these reasons, OverallExecute returns a state, not a value sequence. This state includes the values of all boxes in the program (not just of ending boxes): beginning—those given values by the user before execution starts—boxes, intermediate boxes, and ending boxes. The state includes values of boxes by including the values of outgoing links from those boxes.

Figure 5.6: Situation where *no inconsistent input boxes* property does not hold.

5.3 Inconsistency

Inconsistency requires more information about a DataVis program than is currently represented. First, it requires information about how links cross boxes, so that the flow of data over a link can be cut when a box that it crosses becomes inconsistent. It also needs information about how boxes are nested to control the propagation of inconsistency. Finally, it requires the set of boxes that are inconsistent.

The best way to represent this information is to add the functions Exits, Container, and Open to the program representation (as new components) and to add the set of inconsistent boxes to the state. The type of the Exits function is: Exits: Link $\times$ Box $\rightarrow$ Boolean. Exits(links, box) returns true if a link exits a box, and false otherwise. Section 3.3 discussed how open and closed boxes propagate inconsistency differently. Open: Box $\rightarrow$ Boolean returns true if the box is open and false otherwise.

DataVis allows one box to be contained in another box. Multiple levels of this nesting of boxes are common. For example, in Figure 5.2 the container of **vectorToArrow** is the double-walled iteration box **Box2**, and the container of **Box2** is the large **vectsToArrows** procedure box. This "container" relationship is expressed by the Container function. Container's type is: Container: Program $\times$ Box $\rightarrow$ Box.

All legal DataVis programs must possess the *no inconsistent input boxes* property. The situation in which this property is relevant is shown in Figure 5.6. Informally put, the *no inconsistent input boxes* property means that if box2 becomes inconsistent or causes boxes downstream to become inconsistent, this may not affect its own input boxes. For example, suppose box2 produces a negative scalar (a single number) as a result. This negative number flows downstream to box3. Box3 is the predefined predicate '>0'. Since box3 evaluates to false, it becomes inconsistent; since it is an open (i.e. dotted) box, the inconsistency spreads out to cover all of box4. This includes box1, which is one of box2's input boxes.

Figure 5.6 showed a situation where the *no inconsistent input boxes* property does not hold. The property is stated more formally as follows:

$\forall$ p $\in$ Program, box1 $\in$ p.boxes, box2 $\in$ p.boxes, and box3 $\in$ p.boxes:

(box1< box2 $\wedge$ box2 < box3 $\wedge$ p.container(box1) = p.container(box3))

$\Rightarrow$ p.container(box1) $\in$ Container(box1) $\in$ Containers(box2)

where Containers(box) is the set of all boxes that contain box. For example, in Figure 5.2, Containers(**vectorToArrow**) = {**Box2**, **vectsToArrows**}.

The '<' relation means that the first box topologically precedes the second. The conclusion of the property (i.e., the "then" part) states that box2 must be contained in the container of box3, rather than standing off to one side by itself, as in Figure 5.6. All legal DataVis programs must have the *no inconsistent input boxes* property.

The reason for insisting that all DataVis programs have the *no inconsistent input boxes* property is to insure that each box in a DataVis program only has to be evaluated once. If a box is allowed to have an inconsistent input box, then it would be necessary to reevaluate a box whenever one of its input boxes changed. If the box were not reevaluated when its inputs changed, it is possible that the DataVis program containing the box would be nondeterministic, because the final answer that a program would give would depend on the order in which its boxes were evaluated. It is desirable to rule out this possibility, especially since any DataVis program which violates the *no inconsistent input boxes* property is probably nonsensical anyway. At least, I considered possible DataVis programs, and I could not think of any useful DataVis programs that would violate the *no inconsistent input boxes* property.

Since the group of inconsistent boxes changes as the program executes, it should be part of the state as follows. Up until now a state was just a function from Links to Values. The InconsistentBoxes set is modeled by extending states to be functions on Boxes, as well. State's type becomes State: (Links $\rightarrow$ Values) $\cup$ (Box $\rightarrow$ Boolean). If b is a box, then state(b) is true if and only if b is inconsistent.

Figure 5.7: Show and Tell: effect of inconsistency on data flow [Kimu86a].

```
MakeInconsistent: Box × State × Program → State
  λbox, state, p.
    {state1 ← RemoveAll(λb. if b = box then true else state(b),
                        {lnk | p.exits(lnk, box) = true}).
     if p.open(box)
       then MakeInconsistent(p.container(box), state1, p)
       else state1
     }
```

Figure 5.8: MakeInconsistent function.

Inconsistency permeates an open box (one with a dotted outline) and continues to spread outward until it encounters a closed box (one with a solid outline). In contrast, inconsistency is confined within a closed box. Figure 5.7 demonstrates open and closed boxes. In the top half of the figure, two's flowing into three causes boxA to become inconsistent, but since boxA is closed, the inconsistency is confined within boxA. In the bottom half, boxB also becomes inconsistent, but it is open, so its container (boxC) also turns inconsistent.

The MakeInconsistent function describes the meaning of making a box inconsistent and propagating the inconsistency outward to the first closed box within which the original box is contained. Its definition is shown in Figure 5.8.

MakeInconsistent makes box become inconsistent (by passing to RemoveAll a state function that sets state(box) to true) and removes box's outgoing links from the current state, to indicate that no values may be transmitted on links exiting box. If box is open, its container is also made inconsistent. Finally, if box is closed, MakeInconsistent returns the modified state1.

RemoveAll removes a set of links from a state. The definition of RemoveAll is:

RemoveAll: State $\times$ Links $\rightarrow$ State

```
λstate, links. {if links = ∅
                    then state
                    else RemoveAll(Remove(state, head(links)),tail(links))
               }
```

RemoveAll calls Remove, whose definition is:

Remove: State $\times$ Link $\rightarrow$ State

```
λstate, link1. {λlk. if lk = link1 then ⊥ else state}
```

Remove takes link1 out of the current state by returning $\perp$ if it is given link1 as input. $\perp$ (*bottom*) means that no values will be transmitted on link1. If Remove is given a different link than link1, it simply returns the value given to it.

An open icon box may "return" a value of inconsistent or consistent by turning cross-hatched or not. If the icon box is open (i.e. dotted) the inconsistency will spread outside the icon box to the rest of the program. This method of returning a value from the icon box is separate from the traditional mechanism of outgoing arrows from the icon box.

Primitives can be open, as well as icon boxes. In Figure 5.1, '>0' is an example of such an open primitive. There, '>0' becomes false (and thus inconsistent) only when the number being cycled through the top pair of sequential ports (small triangles) is decremented to 0. When '>0' becomes inconsistent, since it is an open primitive, the inconsistency spreads outward to affect the entire iteration box. The iteration box stops its repetition because it has become inconsistent.

Another point worth noting about DataVis primitives is that some primitives will only execute if all their input data objects are present, whereas other primitives are more lenient. **join** (see Figure 3.7) is an example of the second class of primitive.

Inconsistency requires redefining the function FinalState, which ExecuteList calls. ExecuteList itself is unchanged, and still applies the behavior of any DataVis box, but will now also handle inconsistency appropriately if the box happens to be an open icon box that has just become inconsistent.

If an icon box is consistent, then applying its behavior yields a sequence of output values. If it is inconsistent, then applying its behavior yields a value of *failure*. If it is inconsistent and open then the inconsistency must be propagated outward, but if it is closed, then *failure* does not propagate.

Instead of changing ExecuteList to check whether a box is inconsistent, we will change FinalState, the semantic function that converts the output sequence into endState. FinalState does this whether outputSequence is a sequence of output values or the single value *failure*. If the output is a sequence of values, FinalState copies the values to the box's output links. If the output is *failure* and the box is open then the new version of FinalState causes the container of the box to become inconsistent. Otherwise, the state doesn't change.

The definition of the new version of FinalState is:

FinalState: State $\times$ Program $\times$ Box $\times$ ValueSequence $\rightarrow$ State

```
λinitialState, p, box, outputSequence.
   {if outputSequence ≠ failure
     then FS(initialState, p, box, p.outputports(box), outputSequence)
     else if (outputSequence = failure and p.open(box))
        then MakeInconsistent(iconBox, initialState, p)
        else initialState
    }
```

5.4 Iteration Boxes/Loops

To describe the meaning of executing an iteration box, it is first necessary to briefly review how global inputs and sequential and parallel ports function together with an iteration box.

Figure 5.9 displays an iteration box with a pair of sequential ports and an outgoing parallel port. (A note about terminology: when I deal with sequential and parallel ports, I use the terms "*incoming* and *outgoing* sequential or parallel port". An incoming sequential or parallel port is located near the left side of an iteration box, and provides input data to the boxes inside the iteration box. An outgoing sequential or parallel port is located near the right side of an iteration box, and accepts output/result data from boxes inside the iteration box. In contrast,

Figure 5.9: Example of global inputs, sequential ports, and outgoing parallel port together with an iteration box.

Figure 5.10: Example of incoming parallel port together with an iteration box.

the terms *input port* and *output port* refer to invisible ports which are not sequential or parallel ports, and which are part of any box—not just an iteration box.)

In Figure 5.9, the 5 is a global input that controls the number of iterations, and that is sent into the iteration box every time the iteration box's contents are executed. The 1 is sent through the left sequential port, is incremented, and goes to the right sequential port, which recycles the 2 back to the left sequential port. This goes on until the number reaches 6. At that point it is greater than 5. This causes the '$\geq$' box, and thus the whole iteration box, to become inconsistent. The right sequential port sends its value of 6 to the empty number box Box4.

Whereas the pair of sequential ports dealt only with one (albeit changing) value, the outgoing parallel port accumulates the values one...five into a field, which the parallel port then sends to the empty number box Box5.

One more example will help explain how an incoming parallel port works. In Figure 5.10, Box1 contains a scalar field consisting of the values one...four. This field flows to the incoming parallel port, which disaggregates the field into its constituent integer elements. The incoming parallel port then feeds the integers into the iteration box one at a time, executing the contents of the iteration box once for each integer. After the last execution, the scalar field is exhausted, so it causes the iteration box to become inconsistent, thus halting the iteration. All the while, as the parallel port has been performing its job, the pair of sequential ports has recycled the sum variable. Box2 initialized this variable to zero; after the iteration is finished, the sum's final value is 10. Box3 will hold the 10 after the program has executed.

The BehaviorForIterationBox function produces a behavior for an iteration box—a function that accepts a sequence of input values, describes the execution of the iteration box, and returns the resulting sequence of output values. Thus, if b is an iteration box in program p then p.boxbehavior(b) = BehaviorForIterationBox(p.internalProgramFor(b)). The input sequence given to Iterate includes three sub-sequences of values: seqPortValues (initially, these are input values to the incoming sequential ports), parInputs (the values of incoming parallel ports), and globalInputs (the values of global inputs—i.e., links that cross into the iteration box without passing through an incoming sequential or parallel port).

For each of the three subsequences, there is an accessing function that, given an input sequence for an iteration box, returns the appropriate subsequence. Each of the three accessing functions is a component of a Program domain-element that represents the DataVis program for the interior of (i.e., surrounding) the iteration box. The three accessing functions and their types are: ExtractSeqPortValuesOf: ValueSequence → ValueSequence; ExtractParInputsOf: ValueSequence → ValueSequence; and ExtractGlobalInputsOf: ValueSequence → ValueSequence.

The definition of BehaviorForIterationBox is:

BehaviorForIterationBox: Program → Behavior

```
λp, iterationBox.
   {λinputSequence.
      {Iterate(p.extractSeqPortValuesOf(inputSequence),
               p.extractParInputsOf(inputSequence),
               ∅,
               p.extractGlobalInputsOf(inputSequence),
               p)
       }
    }
```

Iterate: ValueSequence $\times$ ValueSequence $\times$ ValueSequence $\times$ ValueSequence $\times$ Program $\rightarrow$ ValueSequence

```
λseqPortValues, parInputs, parOutputs, globalInputs, p.
   {if any element of parInputs is empty
       then   % i.e., the iteration box is now inconsistent, due to
              % exhaustion of a field feeding into a parallel input port
          append(seqPortValues, parOutputs)
       else   % iterate again
          {sequence1 ← Execute(p, append(seqPortValues,
                           GroupHead(parInputs), globalInputs)).
           if sequence = failure
              then % i.e., execution of iteration box just performed
                   % has made it inconsistent
                 append(seqPortValues, parOutputs)
              else % start over
                 Iterate(p.extractSeqPortValuesOf(sequence1),
                         GroupTail(parInputs),
                         GroupCons(parOutputs,
                            p.extractParOutputsOf(sequence1))
                         p, globalInputs)
           }
    }
```

Figure 5.11: Iterate function.

Among the data items that BehaviorForIterationBox passes to Iterate are seqPortValues. This is a sequence of the values held by the pairs of sequential ports. Each pair holds one value. Since each pair of sequential ports recycles a single value, it is unnecessary to separately consider values for outgoing sequential ports and incoming sequential ports.

ParInputs is a sequence of input data values for incoming parallel ports. These values will be one field or collection per incoming parallel port. (A field is two-dimensional, with rows as well as columns, whereas a collection is a one-dimensional list of elements. The program given as an input to BehaviorForIterationBox (named p) represents the interior of the iteration box.

The Iterate function (see Figure 5.11) describes the meaning of executing an iteration box. Iterate first executes the boxes inside the iteration box—that is, the body of the loop. This execution produces a new sequence of output values: sequence1. Execute may have caused the iteration box to become inconsistent in one of two ways. First, the exhaustion of a field feeding into an incoming parallel port may have caused the iteration box to become inconsistent.

Second, an open inconsistent box inside the iteration box may have caused the iteration box to become inconsistent. (This second possibility is the reason for the call to IterationBoxForProgram. IterationBoxForProgram takes a Program domain-element representing the interior of an iteration box and returns the iteration box. The type of this function is: IterationBoxForProgram: Program $\rightarrow$ Box.)

If Execute has caused the iteration box to become inconsistent, then the iteration is finished. So Iterate returns a single large value sequence consisting of a sequence of the new values on the outgoing sequential ports, appended to a sequence of the new values of the outgoing parallel ports. This is equivalent to putting the results of the iteration box onto its outgoing sequential and parallel ports.

If, on the other hand, the iteration box is still consistent at this point, Iterate executes it again by calling itself recursively with the new seqPortValues from the new value sequence (sequence1), a new sequence of parInputs, a new sequence of parOutputs, and the same program (p) and values of global inputs (globalInputs) as before. The new sequential port values, which p.ExtractSeqPortValuesOf returns, are those generated for the outgoing and incoming sequential ports by the previous iteration. The recursive call to Iterate uses `GroupTail` of the old parInputs to indicate that the first element of each field or collection flowing into an incoming parallel port has been used. (Since seqPortValues is the values of all parallel ports—there may be several parallel ports on one iteration box—and since each such value of a single parallel port is itself a sequence of values, `GroupTail` means a function that takes the `tail` of each sequence that is the parOutput value for a certain parallel port. `GroupCons` has a similar meaning, which is "`Cons` an element of a sequence (here, parOutputs) onto the corresponding sequence that is a value for an outgoing parallel port".) Likewise, the recursive call uses `GroupCons` to add new outgoing port values, generated by the execution of the iteration box and accessed by p.ExtractParOutputsOf, to the old list parOutputs of values held by outgoing parallel ports to show that the outgoing parallel ports are accumulating individual elements (output values) into new fields or collections.

Eventually the iteration box will become inconsistent. Either an open predicate or function inside the iteration box will turn inconsistent, or some field or collection feeding an incoming parallel port will exhaust itself, thus causing the iteration box to become inconsistent. Either way, iteration will cease, and the last sequence of new values will be put on outgoing ports (both

```
HiOrdUserDefinedFunction: Program → HiOrdBehavior
   λp.
      {λbehavior.
         {UserDefinedFunction(ProgramWithBehavior(p,
             λbox. if box = p.functionSlot then behavior
                                 else p.boxbehavior(box)))
          }
       }
```

Figure 5.12: Meaning of a higher-order user-defined function.

sequential and parallel) so that the results of the iteration box will be available to whatever portion of the overall DataVis program lies downstream from it.

5.5 Higher-Order Functions

The function slot used with higher-order functions is a special kind of box, since it has no particular Program domain-element defined for it. Rather, the slot uses the program associated with whatever box (i.e., first-order function) the user places into it. I will use this fact in the semantic function that describes the meaning of a higher-order function.

For a given function, both the icon box (function call) and the procedure box (function definition) have a function slot. Although at run time the contents of the icon box's function slot will be transmitted to the function slot of the procedure box, the two function slots are distinct.

The function slot of a procedure box with program p is p.functionSlot. The function slot of an icon box b inside program p is p.slot(b). Thus, both functionSlot and slot are new components of programs.

The type of the HiOrdBehavior function is: HiOrdBehavior: Behavior → Behavior. This type indicates that a higher-order function's behavior is to use the behavior of the first-order function placed into its function slot to form a new behavior for the combination of the higher-order function and the first-order function in its function slot.

In the same way that UserDefinedFunction (recall UserDefinedFunction's definition in section 5.2) gave the meaning of a user-defined function, HiOrdUserDefinedFunction (see Figure 5.12) tells the meaning of a higher-order function.

HiOrdUserDefinedFunction takes as input an element of the Program domain that represents the interior of the procedure box for the higher-order function. HiOrdUserDefinedFunction then produces a HiOrdBehavior, which is a function that takes a behavior for the first-order function in the slot and produces a behavior for the higher-order function, *with* the first-order function in its function slot. The HiOrdUserDefinedFunction gives a program (the result of calling ProgramWithBehavior) to UserDefinedFunction.

Assume that p is a program and b is a boxbehavior. ProgramWithBehavior(p, b) produces a program just like program p, except that the boxbehavior function is b. (ProgramWithBehavior's type is Program $\times$ Behavior $\rightarrow$ Program.) The new boxbehavior is just like the old one except that the behavior of the slot (i.e., the first-order function *in* the slot) is behavior.

UserDefinedFunction returns a behavior. Thus, HiOrdUserDefinedFunction has completed the definition of the function which it is required to return—a function which, given a behavior, produces another behavior.

When a higher-order function is called, either the user has already placed a first-order function into its function slot, or the function that calls the higher-order function transmits the first-order function from the caller's function slot to the function slot of the called higher-order function. The called higher-order function then takes in input data from its incoming links, applies the first-order function that is in its function slot to some or all of the input data, and puts the resulting output data on its outgoing links.

What happens when a hiordbehavior is used? This is equivalent to asking, "What happens when a higher-order user-defined function is run?". Recall that p.boxbehavior defines the behavior of the boxes in p. If b is a higher-order function, then p.boxbehavior(b) = HiOrdUserDefinedFunction(p.internalprogramfor(b)) (p.boxbehavior(p.slot(b))). The behavior returned by p.boxbehavior(b), where b is a higher-order function, corresponds to running the higher-order function—i.e., taking in input data, applying some or all of the input data to the first-order function, and returning the resulting output data. The behavior is applied to a starting value sequence; it describes the execution of the higher-order function with the first-order function in the function slot, and returns an ending value sequence.

This chapter has provided a precise specification of the DataVis language. The development of this specification uncovered a few small problems in the implementation of DataVis. The specification was created after most of the implementation had been done. It might have been useful to make the specification earlier in the implementation process, as an aid in discovering flaws in the implementation. Chapter 6 gives conclusions of this thesis.

Chapter 6

Conclusions

This chapter presents conclusions of the thesis. These include: significant features of this research; knowledge gained about the design, implementation, and use of a data flow visual programming language for scientific visualization (as exemplified by DataVis); approaches to scaling up DataVis; and, more generally, topics for future research in the areas of visual programming and software systems for visualization of scientific data.

DataVis is an important addition to the pantheon of existing visual programming languages. DataVis is the only visual programming language that is geared toward the application domain of scientific visualization. Also, visual programming languages are a new area of computer science. Because visual programming is such a new field, there are not many visual programming languages that work well. Thus, the advent of a new visual programming language on the scene is interesting. Also, I encountered (and handled) some unexpected problems as DataVis was implemented. (Section 6.2.2 will discuss these problems and their solutions.) For both of these reasons, the creation of a visual programming language for scientific visualization has been a worthwhile research effort.

6.1 Significant Features of This Research

This research has a number of significant features:

1) DataVis embodies three ideas which, although they are not new individually, have never been combined before. The ideas are a data flow computational model, a visual programming language, and an application domain of scientific visualization. There are no existing data flow

visual programming languages for scientific visualization. (AVS, apE, and Iris Explorer are not programming languages, just libraries of components that may be hooked together. It could be claimed that LabVIEW is a visual language for visualization. However, LabVIEW is specialized for lab-instrument data collection and data analysis, and is not particularly well-suited for visualization of scientific data. VIPEX might be considered a visual language for visualization, but its application domain covers only image processing, which is a narrow subfield of the broader area of scientific visualization. That is, VIPEX does not work on scalars, vectors, or fields, just images.)

DataVis shows that it is possible to make a visual programming language for scientific visualization. Further, such a language can be more flexible and powerful than existing software systems for doing scientific visualization. Most of these systems, like NCSA Image or NCSA DataScope, limit the scientist by hard-wiring in only certain kinds of scientific data and certain graphic displays. DataVis provides more flexibility than such systems: scientists can connect predefined DataVis functions together to handle new data or new graphic displays.

Admittedly, visualization systems like AVS, apE, and Iris Explorer also provide predefined functions that may be hooked together. However, should the functions of AVS, apE, or Explorer prove inadequate to the scientist's visualization task, the scientist must shift levels by leaving the world of visual icons and data flow, and resort to writing textual code.

DataVis is more powerful than AVS, apE, or Explorer because it provides a complete programming language that is already integrated with DataVis's predefined functions and that can be used to extend the visualization system. This programming language is visual and based on data flow, so the scientist need not leave the world of pictures and data transformations to which he or she has become accustomed. The results of DataVis's greater flexibility and power are that: a) a scientist using DataVis should be able to accomplish spend more time on visualization and less on learning the system; b) DataVis should be easier to modify and extend than existing visualization systems; and c) DataVis should be easier to learn than existing visualization systems.

2) The addition of higher-order functions helps DataVis alleviate the *screen space* problem and the *control flow* problem. The screen space problem is that even rather simple visual programs greedily consume available screen space. Higher-order functions (together with procedural abstraction) lead to a more concise, compact representation for the program and

so help DataVis to handle the screen space problem better than other visual languages do. Higher-order functions make it possible to build DataVis functions such as **imageIterator** and **fieldIterator**, which are more compact than the full iteration boxes for the iteration that **imageIterator** and **fieldIterator** perform. As well as helping with the screen space problem, higher-order functions also make DataVis more flexible because the effect of a program can be radically altered by placing a different function in a function slot, as shown by the example of section 4.2.3 which extended **vectsToArrows** to make it into **scalsToImage**.

In addition, higher-order functions help solve the control flow problem. The control flow problem is that it is often necessary to introduce control flow constructs (loops, CASE statements, sequential-execution constructs, etc.) into a data flow visual language. However, control flow constructs can be awkward and hard to think about when inserted into a data flow language. Higher-order functions attack the control flow problem because such functions package up control flow features of a language that is primarily based on data flow. For example, the details of an iteration box, which is partially based on control flow, are rather ugly and are somewhat hard to understand. However, DataVis's higher-order functions allow the language designer to present an easy-to-understand abstraction of an iteration box, such as **fieldIterator** or **imageIterator**. A scientist can iterate over a field or image and apply a function to each field element or pixel by using **fieldIterator** or **imageIterator**, respectively, and does not have to deal with the details of iteration boxes, sequential ports, parallel ports, and global inputs.

3) DataVis, as a visual language, shows that an extensible system for visualization need not make the user view it solely or even primarily through one-dimensional textual strings, unlike AVS, apE, Iris, and other visualization programs written in C or Fortran. Rather, like all visual programs, DataVis programs are two-dimensional and graphical. Also, DataVis uses color, shape, and thickness (e.g. the different outlines of various kinds of boxes), fill patterns (e.g. the cross-hatching that indicates a box is inconsistent), and graphics (e.g. the graphical name portion of a procedure's icon box) to convey information to the user. These visual elements (color, shape, and so on) give a visual language a broader, richer communications medium for helping and informing the user than textual languages such as C or Fortran have.

4) Visual languages make certain aspects of a program much easier to see than in a comparable C or Fortran program. (However, the screen space problem partially counteracts this

benefit of a visual language.) While many visual languages have made this claim, few have fulfilled it. Two reasons are evident. First, many existing visual programming languages were toy systems and lacked sufficient power to attack real problems. Specifically, they lacked procedural abstraction, as well as higher-order functions and type checking. For example, Hookup [Levi86] and ConMan [Haeb88] lack procedural abstraction; Hookup, Fabrik [Inga88], InterCONS [Smit88], HI-VISUAL [Yosh86], VIVA [Tani90a], Cantata [Rasu90], VIPEX [Haars88], LabVIEW [Nati87], ConMan, the Visual Toolset [Borg90a], PROGRAPH [Matw85], and Show and Tell [Kimu86a] lack higher-order functions; finally, Hookup, InterCONS, HI-VISUAL, VIVA, VIPEX, *viz* [Holt90], the Visual Toolset, PROGRAPH, and Show and Tell lack type checking. On the other hand, DataVis has procedures, higher-order functions, and type checking.

Second, a number of existing visual languages did not fully exploit the richness of graphics for communicating information. In contrast, DataVis has iteration, procedures, higher-order functions, and type checking. In contrast, DataVis shows how to fulfill visual languages' claim of presenting some aspects of a program more clearly. Another name for such visual presentation of aspects of a program is *program visualization*. For example, statically (i.e. at program-construction time) DataVis displays the graphical, or "iconic" name portion of icon boxes and colors links by type. (Figure 6.1 identifies the graphical name portion and the textual name portion of an icon box.) Both of these features have no analog in textual languages. In addition, DataVis, being a data flow visual language, makes the flow of data in a program clearly visible, and quickly shows whether or not changes to one part of a program will affect another part.

Dynamically (i.e. at run time), DataVis flashes links and alternately displays parameters in base boxes and visually clears out the base boxes. Again, textual languages must resort to calls to a graphics library to give the user any comparable kind of aid. Also, one piece of the knowledge gained by this research is that flashing links and displaying parameters dynamically, as well as coloring links by types statically, are useful program-visualization techniques in a data flow visual programming language.

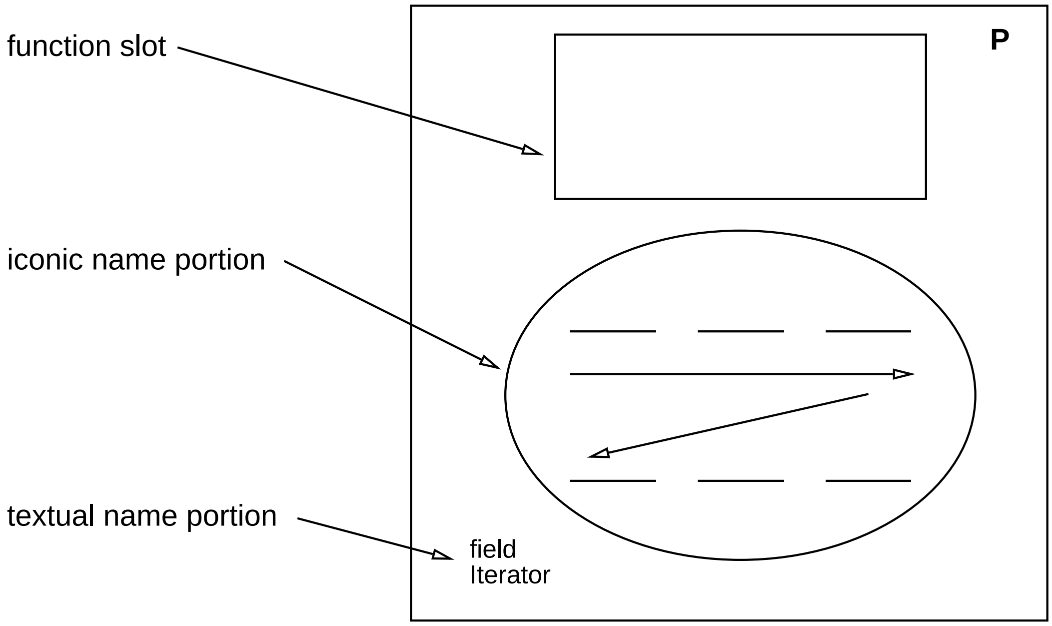

Figure 6.1: Various parts of a blown-up DataVis icon box.

6.2 Knowledge Gained About the Design, Implementation, and Use of a Data Flow Visual Language for Visualization

6.2.1 Design

An assumption behind the design of DataVis is that a software system for visualization should be customizable and extensible, and should have the full power of a programming language. Users will always want to customize library programs, no matter how large and varied the library of predefined programs and functions. Since current visualization systems are hard to change, and it is important that they be easy to customize, it is not surprising that users (in this case of AVS) have complained that their visualization systems are hard to modify [Trei90] [Palm92]. Even outside the realm of scientific visualization, users are constantly wanting to customize existing programs. For example, most MIS departments in corporations have a huge backlog of requests for modifications to existing software [Luca88] [Weth91]. Visualization is such a rapidly changing field that it is unreasonable to expect a fixed set of routines to satisfy most scientists. Most scientists will often wish to extend or modify the existing functionality provided by a system. A good visualization system should make it easy for the users to customize existing library functions. DataVis does make it fairly easy to modify and extend the library functions provided by the system. (However, DataVis's IF statements are difficult to understand and modify. See Section 6.4.) The reader can see this ease of modification and extension by looking at the examples of Chapter 4.

Another assumption underlying DataVis is that a customizable system requires a full programming language. DataVis is a full programming language. Two reasonable questions to ask are, "Need a visualization system be a full programming language?" and "Does DataVis do better at visualization, by being a full language?".

The answer to the first question is, "If the user wants full modification/extension capabilities *in the system*, Yes." A user cannot make the kind of modifications possible in DataVis's second, *definition* level in AVS or apE without leaving those systems and going to C or Fortran, which are entirely different languages from AVS and apE.

The answer to the second question is "No. DataVis will not produce graphic displays that couldn't have been done in C or Fortran." However, DataVis lets users build the same kind of displays that could be built in C or Fortran without ever learning C or Fortran. Also, DataVis should be easier to learn, for two reasons. (Because a user must learn about a visualization system before he or she can modify it, these are also reasons that DataVis is easier to modify—at least, easier than AVS and Iris Explorer, if not easier than C or Fortran.)

First, the simplest, *connection* level of DataVis is easy to learn because it contains few difficult concepts, because it is presented to the user in a visual/graphic manner, and because it is based on the data flow computational model. This model is an intuitive model for visualization of scientific data.

Second, the fact that DataVis is a complete language helps eliminate big jumps between what is needed to used various parts of it. The gaps between the three levels of DataVis are small. Thus, if a scientist knows how to use a certain level, he or she should not have much trouble learning how to use the next, more-advanced level to modify his or her DataVis functions for visualization.

In contrast, existing visualization systems have a large gap between the top visual level, and the next-lower textual level. DataVis, in contrast, makes this gap much smaller, by integrating its language with its predefined library programs. When a scientist wishes to customize a canned program in AVS, apE, or Explorer, for example, he or she must switch from visual data flow to textual C or Fortran. This is a large mental gap for the user to jump, and, in fact, requires the hapless AVS user to switch gears mentally twice: once for a new computational model (control flow, not data flow), and a second time for a new user interface (textual, not visual). In contrast, all levels of a DataVis program use the same interface and the same model.

An additional assumption behind DataVis's design is that visual programming languages (DataVis, for example) and the application domain of scientific visualization are a good match. Two reasons bear this assumption out. First, the users of software systems for visualization, such as DataVis, would be scientists, who are often nonprogrammers or at least non-expert programmers. Such a user community (that is, a group of non-expert programmers) is well-suited to use a visual language. In fact, a number of the visual languages already developed have successfully catered to such non-expert users. Second, languages for visualization, including DataVis, can get away with a small number of data types—five to seven. (This is also the largest number of colors a language designer should simultaneously display on the screen, according to user-interface researchers [Durr82] [Rowe88]). Since data flow visual languages often represent data types with color or a fill-area pattern (e.g. different line styles for arcs carrying different types), this is fortunate.

Another assumption underlying DataVis's design is the minimal language-design philosophy: that a minimal set of language features is best. For example, DataVis, following the lead of Show and Tell, provides only two major kinds of iteration (iteration boxes with sequential ports, and iteration boxes with parallel ports), rather than providing FOR, WHILE, and REPEAT loops. I have considered in detail a number of algorithms for generating various scientific-visualization images, most of which are not in the thesis. DataVis's two main kinds of iteration have been sufficient to implement these algorithms. The advantage for DataVis in having a minimum number of iteration structures is that a user of the system does not have to learn or comprehend three or four major kinds of iteration. So the above assumption about a minimal feature set has held true for iteration.

Another feature that has successfully been omitted from DataVis is recursion, which is an essential part of most modern languages. Show and Tell has recursion, but I did not implement it in DataVis, and never needed it. This is probably because the standard data structures used in visualization, such as fields and vectors, don't need recursion but are better handled with iteration. An application domain that used trees and other recursive data structures would cry out for recursion, but trees are rarely used in scientific visualization. One exception is octrees, which are recursive data structures used extensively in medical imaging [Same89] [Same90], so it may be eventually necessary to add recursion to DataVis.

This leads to the more general issue of what features (also called design alternatives or language constructs) are necessary in the design of a data flow visual programming language to make the language powerful. A data flow language may be considered *powerful* if it provides the necessary programming constructs and predefined functions so that a user of the language can tackle large, complex problems in the language's application domain. Note that many data flow visual programming languages do not possess the features that I will soon claim are necessary for a language to be powerful. Also note that my definition of *powerful* deals with the practical limitations of what people can do, not just with Turing-completeness. So, although DataVis is no more Turing-complete than Fortran or a Turing machine, it is higher-level and easier for a person to use than either Fortran or a Turing machine.

The following features are *necessary* for a data flow visual programming language to be powerful: iteration, procedural abstraction, and a large and diverse library of predefined functions for use as building blocks in constructing new programs. Procedural abstraction is important to hide unnecessary details and to save screen space. An extensive library of predefined functions which are defined properly can reduce the time needed to create a program, and can help avoid recreating an existing function. Strictly speaking, a large function library is a property of the language implementation, not the language design. A language possessing a small function library may be well-designed, and all that is needed is for a programmer to implement many algorithms as programs in the language for a year or two. In this way a large function library would be built up. Nevertheless, the function library has just as big an impact on the power of a system as any of the language features does.

Although the language features just mentioned are *necessary* to make a language be powerful, whether they are *sufficient* to make a data flow visual language powerful is an open question. I *believe* that that these features are sufficient to make DataVis powerful. However, due to time constraints I have not tested DataVis on enough data sets and graphic displays to *prove* that these features will make DataVis powerful.

Higher-order functions are also desirable for making languages more powerful. (Wherever "language" is used in this section, assume it means "data flow visual programming language".) The trade-off involved in adding higher-order functions to a language is that they add to the complexity of the language, making it harder to implement and presumably, harder for a new user to learn. Type checking does not add to the power of a language so much as it helps catch type errors early, and so makes the language easier to read and use.

6.2.2 Implementation

As is often the case when one implements a new language, unexpected problems arose while implementing DataVis. One is worth mentioning.

This problem is a special case of the *persistent objects* problem, which means keeping objects around between executions of a DataVis program. I decided early on that DataVis should be able to store the current version of each icon box and procedure box on disk. Storing objects such as icons and procedures in memory was insufficient, because if the Smalltalk "image"—the file containing the current representation of all objects in memory—were lost or damaged (which has occurred), all previously-written DataVis programs (which presumably include some icons and procedures) would be lost.

Not only was the safety of disk storage for icon boxes and procedure boxes desirable; speed in retrieving them from disk was also desirable. For example, `add procedure`, the command to add an icon box to a DataVis window, reads the icon box from disk when necessary. No user wants to wait five minutes to add a single procedure call to a program. Yet this occurred when an early version of DataVis tried to retrieve icon boxes from disk.

DataVis programs are complex objects with many interlocking sub-objects. One problem that made disk retrieval so slow was that an icon box contained pointers to other objects, which then pointed back to the icon box. These circular pointer chains took Smalltalk forever to write to disk.

The solution that I adopted to the problem of these long disk-access times was to replace an icon box's pointers to other objects with symbols. Each symbol was a key in a dictionary kept in memory, where the dictionary's value for that symbol-key was the object formerly pointed to.

This approach solved the problem of slow disk access, but created other problems. To break the circular pointer chains, in-memory dictionaries for icon boxes, procedure boxes, and five other kinds of objects had to be created. Objects had to be saved in these dictionaries. Saved versions in dictionaries had to be kept up to date. Because of the risk that the contents of memory might be lost, many of these objects had to be saved on disk. Disk files had to be kept up to date. Management of dictionary entries and disk files to remove old, unused entries and disk files was a constant challenge. Objects (for example, icon boxes) had to be given unique names so their dictionary entries would not accidentally overwrite each other. Code for many commands commonly given by users had to be modified to read or write dictionary entries and disk files. In addition, the indirection necessitated by frequently looking up symbols in dictionaries slowed down the speed at which the current implementation of DataVis can execute a DataVis program. In short, the whole area of in-memory dictionaries and disk files created many headaches and frustration. This whole experience points out the importance of object-oriented databases and persistent objects.

6.2.3 Use

On the whole, DataVis performs as advertised. That is, DataVis is a good language, but is not implemented well. The DataVis language makes it possible to write programs for visualization; it provides an extensible, powerful visualization tool; it presents a consistent data flow model and visual interface with both its library programs and its programming language; and program visualization, along with the fact that DataVis is a visual language, helps DataVis to present information on-screen more effectively and more helpfully to the user than a textual language like C or Fortran could ever do. However, as with all large software systems, some aspects of DataVis could be improved.

Storing icon boxes and other objects on disk and retrieving them from disk is acceptably fast, and is much faster than it would be without all the in-memory dictionaries.

The speed of the current implementation makes it too slow to use at interactive speeds (2-3 seconds, at most, to generate an arrow plot, for example). This does not mean DataVis is irrelevant to scientific visualization; it means that more programming work would be needed to speed up DataVis and to turn the current implementation, which is a research prototype, into a piece of commercial-quality software. However, DataVis does not currently have scientists using it. The prototype's slow speed is one reason for this.

Little effort has been expended to make DataVis fast. A change to an earlier version of DataVis resulted in a 50% speedup in the current implementation. However, with further study it should be possible to make the implementation considerably faster, given substantial additional time and effort to spend on improving the implementation. DataVis could be made faster by compiling DataVis functions into Smalltalk (such functions are presently interpreted), by using a more efficient language than Smalltalk (C++ is a good possibility), and by using a faster machine (the current DataVis prototype runs on a Tektronix workstation that is based on the Motorola 68020 processor).

Smalltalk does bear some responsibility for the lack of execution speed in the current implementation of DataVis. Smalltalk is a language whose fastest current implementation achieves only 10% of the speed of C [Cham89]. This is a well-known problem and is the reason why research is being done to make Smalltalk faster [John88]. Despite the slowness of existing versions of Smalltalk, the decision to use Smalltalk to implement DataVis was a good one. Smalltalk's programming environment is excellent, and is still superior to environments available for C++. Further, Smalltalk is an object-oriented language and as such, gave me two advantages. First, an object-oriented language, with its modularity, packaging of data objects with code, and inheritance, (not to mention Smalltalk's large library of existing code), made it easier to implement a large, complex system such as DataVis. Second, because Smalltalk, like any real object-oriented language, provides inheritance, DataVis can inherit code and the structure of certain objects from code frameworks such as Foible. Foible [Jind90] is an object-oriented code framework designed to aid in the production of visual programming languages. Foible provides some basic functionality for editing links and simpler types of boxes.

Implementing a visual language takes a very large amount of effort because much sweat and toil must be expended on simply building a specialized editor for the language. The implementation of DataVis alone—aside from the Foible code framework for visual languages which DataVis uses—took three years and produced 36,000 lines and 1.4 Megabytes of Smalltalk source code. At least 50% of the code for DataVis deals with such mundane tasks as adding, deleting, moving, and resizing boxes and links. Such a huge effort is necessary because visual counterparts for vi and emacs do not exist. That is, general-purpose editors, debuggers, interpreters, and compilers for visual programming languages are not readily available (presumably because they have not been made yet). Nor are tools for making editors, debuggers, interpreters, and compilers for visual languages readily available yet. However, research is being done on making the job of producing a visual language into an easier task. One approach is to use grammars to describe a visual language [Goli90]. Another approach, which I took with DataVis, is to use a framework for visual languages, such as Foible.

Implementing DataVis showed some weaknesses in Foible, and pointed out some features needed in a framework like Foible for a language designer/implementor to use it to build a visual programming language quickly. The first necessary ability in a visual-language framework is for the language implementor to be able to easily add new editing operations to the language, and to easily modify existing editing operations. "Editing operations" include adding, deleting, moving, and resizing the basic visual elements of the language. For DataVis, there are two such visual elements: boxes and links. Foible makes it easy to add and change operations on simple boxes (i.e., boxes that do not contain other boxes) and on links. However, it does not make it easy to operate on complex boxes (boxes that contain other boxes), and implementing procedure and iteration boxes and inconsistency took a lot of time.

Another essential ability that a framework for visual languages (at least, a framework for *data flow* visual languages) must have is support for boxes that contain other boxes. Such boxes are very important in a non-toy data flow visual language. For example, in DataVis, such boxes include iteration boxes, procedure boxes (which are used to define both first-order and higher-order functions), and complex boxes. These three kinds of boxes are associated with constructs that provide much of DataVis's power; thus it is important that a framework to support such boxes. By "support" I mean both creating these kinds of boxes, and the editing operations on these boxes that were discussed above. The current version of Foible does not provide such support.

A visual-language framework needs to support color. A black-and-white visual language is like an eagle with clipped wings: it may get off the ground, but it will never soar. The present version of Foible does not support color. I wrote Smalltalk code to handle color and boxes that contain other boxes, since Foible did not handle either of these two items.

Finally, I learned that a framework should support editing of bitmaps/images like those found in the graphic name portion (recall Figure 6.1) of icon boxes, and in the name area (upper left corner) of procedure boxes. Foible simply uses the rather primitive color-bitmap editor provided by Tektronix's color Smalltalk. A more sophisticated tool would be a great help in quickly constructing a visual programming language.

An upcoming version of Foible (a.k.a. InVaider) [Cont92] will make it easier to add and change editing operations on visual elements, since these versions will be based on HotDraw [John92], a structured-drawing editor implemented in Smalltalk. (Unidraw [Vlis89] [Vlis90] is a more widely-known system that is similar to HotDraw, but is implemented in C++.) The new version of Foible will also support color and boxes that contain other boxes.

6.3 Scaling Up DataVis

The following extensions will help DataVis to scale up into a commercial system that can handle complicated (i.e., 3D) data sets and graphic displays. (Note that DataVis already has the necessary mechanisms—procedural abstraction and iteration—to handle complicated *algorithms*, as opposed to complicated *data sets* and *graphic displays*.) Many of these extensions were not included in the current implementation due to time limitations. The extensions to DataVis mentioned in this section are primarily changes needed to convert the present implementation of DataVis from a prototype into a real product, rather than areas for future research. The next section covers topics for future research.

DataVis could use a record structure, both to hold data objects of disparate types, and to help users define their own data types so objects of those new types could flow across links in DataVis programs. The lack of such a record structure is a weakness in DataVis. However, adding records to a visual programming language does not constitute new research, since several visual languages (e.g. PiP [Raed84], PROGRAPH [Matw85] [Cox89]) have already implemented records.

Also, the fact that one must `define procedure` in a procedure box before one can `add procedure` to get an icon box representing a call to that procedure encourages the bad practice of bottom-up programming. The DataVis programmer who is determined to follow good rules of programming design—specifically, top-down design, where one adds the call to a procedure *before* defining the procedure itself—can get around this problem by using a number box as a temporary stand-in for an icon box. Nevertheless, this is just a stopgap measure, and DataVis should provide better support for top-down design.

Another useful item would be a new kind of **fieldIterator**. Such a DataVis library function would iterate over rows in a field, rather than over the individual elements in a field.

Facilities to handle incomplete or missing scientific data would be useful. Also desirable are more DataVis library functions and programs to produce the many graphic displays used in scientific visualization and to handle more types of input data. In particular, it would be nice if DataVis had library functions and programs to handle 3D scalar and vector fields. At the moment, DataVis is too slow to handle 3D data, but it would not be hard to add support to 3D when it becomes faster.

DataVis also needs predefined functions to read and write common data formats, such as NCSA's Hierarchical Data Format. Another useful extension to DataVis would be the ability to compress, save on disk, and retrieve from disk intermediate data objects, such as images and fields, that are created during the execution of a DataVis program. At present only fields created by using Smalltalk code before the DataVis program runs, as well as images created by DataVis, may be written to and read from disk. In general, most of these functions would be implemented primarily in DataVis, with a few new primitive functions being implemented in Smalltalk.

Specifically, a new kind of view (i.e., window) could be created that would be usable both for viewing and managing (i.e. reading/writing to disk) images created as the *end product* of executing a DataVis program, and for viewing and managing *intermediate data-object* images created during the execution of a DataVis program. Such a view could be called a *viewing monitor*, *spy box*, or *inspector*. This view could be placed on a link carrying images to display each image at run time. This would make it easier to look at images, without changing the DataVis program of interest by adding to it a function like **displayAt:**.

Better performance (i.e., a faster running speed) would also be helpful in scaling DataVis up. The next section discusses performance in more detail.

6.4 Areas For Future Research in Visual Programming Languages and Software Systems for Scientific Visualization

The problem of visual languages' quickly consuming much screen space, although ameliorated by procedural abstraction and higher-order functions, still exists. Research aimed at finding new ways to address the screen space problem would be useful.

Often a language designer finds it necessary to add control flow constructs to data flow visual languages: iteration constructs based on control flow, sequential-execution constructs, and CASE statements. The associated research topic is deciding which control flow constructs, if any, should be in a visual data flow language. DataVis has iteration boxes, which are not entirely based on control flow, and inconsistency, which is related to data flow, but does not have a sequential-execution construct. (Iteration boxes have certain aspects of control flow, because they act as loops. Yet at the same time, iteration boxes also have aspects of data flow because their execution can be stopped by the exhaustion of their input data—for example, a field entering the iteration box through a parallel port—not just by the standard control flow mechanism of fulfilling a termination condition (e.g. $x > 10$).) Also, higher-order functions should be considered together with control flow constructs, because higher-order functions allow a language designer to abstract away from control flow constructs (e.g. by packaging up a certain kind of loop as a particular predefined higher-order library function). The application domain and intended users of the language are important considerations in this decision—that is, which, if any, control flow constructs to put into a data flow visual language. Specifically, the IF...THEN statement and the CASE statement are two such control-flow constructs.

DataVis's data flow IF...THEN statement, even with inhibitory arrows, is awkward, clumsy, and consumes too much screen space. The inhibitory arrows are certainly an improvement; yet even with them, this is still not a satisfactory construct. Further research is needed to discover a more elegant, compact conditional construct for visual data flow languages which use inconsistency.

DataVis needs a CASE statement. I proposed one, but due to time limitations, it was not implemented.

Much research needs to be done on making visual languages run faster. This goal is an important step toward making visual languages useful tools for non-computer-scientist users to do real applications, rather than just letting visual languages be academic curiosities for computer scientists. In the case of the particular visual language DataVis, the language is saddled with three factors which make it execute slowly: a naive interpreter (for DataVis), an inefficient underlying language (Smalltalk), and a slow workstation (a Tektronix 4319, based on a Motorola 68020 processor). Improvement of any or all of these factors would result in faster DataVis programs. Only the first factor is an area that needs research.

One way that DataVis could support top-down design is to provide debugging boxes. A debugging box, when executed, pops up a window to ask the user, "What should my output value be?". The user could put a debugging box into a DataVis program under development, in place of an icon box whose procedure box had not yet been defined. The user could even run the program containing the debugging box to test parts of the program other than the incomplete function.

Multimedia consists of adding moving video clips and still photographs—both in color—as well as CD-quality sound to the text and graphics already on the screens of PCs and workstations. Multimedia has a lot of potential to help the scientist visualize his or her data, as well as much potential to broaden the menu of existing techniques (i.e. two dimensions, color, text, graphics) for expressing visual programs. However, multimedia has not yet been used much for visualization or for visual programming. The same is true for user interfaces based on virtual reality and for alternate input devices (moles, foot pedals, space balls): it is possible that they can make it much easier to visualize scientific data and to work with visual programs, but not much research has yet been done on applying them to visualization or to visual programming. (However, see [Najo91] for CUBE, a 3D visual language with a virtual-reality-based user interface.)

Finally, finding new appropriate application domains for data flow visual languages is an area worthy of investigation. Some possible application domains include multimedia systems, hypertext and hypermedia, and medical imaging.

All of these domains deal with data that can be represented graphically—i.e., audio and video data, for multimedia systems; nodes of information, and links between nodes, for hypertext and hypermedia systems; and 2D images and 3D graphics, for medical imaging. Since data flow visual languages seem able to deal well with graphical representations of data in other application domains (e.g. scientific visualization, image processing), it seems plausible that data flow visual languages are also good candidates to tackle the above application domains.

With regard to multimedia systems, data flow visual languages could be used to edit audio and video data. A data flow visual language could be used to synchronize video clips with audio clips (e.g. narration, music). for this to work, the data flow computational model would have to be expanded to handle time, since time is an important part of video and audio data.

Further, for multimedia systems, video data could be represented visually on the screen (e.g., by displaying key frames from the video clip). Audio could be handled with digital signal processing, which Cantata has already shown to be an application domain amenable to a data flow visual language. Although a user of a data flow visual language for creating multimedia systems might not be interested in the low-level DSP details of processing audio data, the language could abstact away and hide such details from the user, while retaining the visual, data flow paradigm. This would let the user see a data object that represented a "chunk' of sound as it passed through various high-level transformation functions (e.g. **increaseVolume**, **increasePitch**, **filterOutHighFrequencyNoise**) and high-level editing functions (e.g. **overlayAndMixTwoSounds**, **fade1stSoundOutAndSecondSoundIn**, etc.).

For that matter, it may be possible to make high-level editing functions in a data flow visual language for *video* data that are analogous to some of the example *audio* editing functions just mentioned. Such video editing functions could be based on the movie paradigm, and could embody various kinds of editing and filming done with movies: e.g., **zoom**, **tilt**, **pan**, **dissolve**, **fadeToBlack**, etc.

Medical imaging is based on 2D image processing (which data flow visual language such as VPL, VIPEX, Cantata, and VIVA have handled well), with extensions to 3D graphics in such areas as volume rendering. Since DataVis seems capable of handling such 3D extensions—they are, after all, a subarea of scientific visualization—it stands to reason that other data flow visual languages may be useful in attacking problems in the medical-imaging domain.

It would be interesting to see whether languages geared toward these domains (multimedia systems, hypertext and hypermedia, and medical imaging) can use color effectively for types, whether they can perform well without records, whether they need recursion, and whether they can make use of a visual-language framework.

6.5 Conclusion

Data flow visual programming languages work well for visualization of scientific data, due to:

• the natural match between visualization and the data flow computational model, which matches the scientist's thinking and which allows for insertion of viewing monitors;

• the ability of a data flow visual programming language to minimize the drawback of a visual language's heavy use of screen space, while maximizing the benefits of program visualization;

• the ready ability that data flow visual languages have to make it easy for the user to extend or modify library programs, by presenting the scientist with a layered language design, so that he or she can learn the system level by level, one easy step at a time;

• the flexibility and power inherent in any complete programming language, which lets the user modify and extend existing programs, not just connect predefined library functions;

• and the match between the visual representation of programs in a data flow visual language, and the visual nature of the graphic displays (a.k.a. images or bitmaps) which are produced for scientific visualization.

Although not every application domain fits well with data flow and deals with graphical data, there are many application domains that do, other than just scientific visualization. A language like DataVis might work well for these domains, too.

BIBLIOGRAPHY

[Ager82] Agerwala, T., and Arvind (1982) Data Flow Systems: Guest Editors' Introduction. *IEEE Computer* **15**, 10-13.

[Borg90a] Borges, J. A. (1990) Multiparadigm Visual Programming Languages. Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.

[Borg90b] Borges, J. A. and Johnson, R. E. (1990) Multiparadigm Visual Programming Languages. *IEEE Workshop on Visual Programming Languages.* Skokie, IL, October 4-6, 1990. pp. 233-240.

[Cham89] Chambers, C., Ungar, D., and Lee, E. (1989) An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based On Prototypes. *Proceedings ACM OOPSLA '89*, printed as *ACM Sigplan Notices* **24**. New Orleans, LA, October 1-6, 1989. pp. 49-70.

[Clar88] Claris Corporation (1988) *MacDraw II* Claris Corp., Mountain View, CA, 367 pp.

[Cont92] Conti, J. (1992) InVaider: A Framework for Visual Programming Languages. M.S. Thesis, Department of Computer Science, University of Illinois. Urbana, IL 61801.

[Cox89] Cox, P. T., Giles, F. R., and Pietrzykowski, T. (1989) PROGRAPH: A Step Towards Liberating Programming From Textual Conditioning. *IEEE Workshop on Visual Languages.* Rome, Italy, October 4-6, 1989. pp. 150-156.

[Davi82] Davis, A. L. and Keller, R. M. (1982) Data Flow Program Graphs. *IEEE Computer* **15**, 175-182.

[Denn75] Dennis, J. B. (1975) First Version of a Data Flow Procedure Language. Technical Report MIT/LCS/TM–61, Laboratory for Computer Science, MIT.

[Durr82] Durret, J. and Trezona, J. (1982) How To Use Color Displays Effectively. *Byte* **7**, 50-53.

[Dyer90] Dyer, D. S. (1990) A Dataflow Toolkit for Visualization. *IEEE Computer Graphics and Applications*, **10** July 1990. pp. 60-69.

[Fole87] Foley, J. D. (1987) Interfaces for Advanced Computing. *Scientific American* **257**, 127-135.

[Glin84] Glinert, E. P. and Tanimoto, S. L. (1984) Pict: An Interactive Graphical Programming Environment. *IEEE Computer* **11**, 7-25.

[Goli90] Golin, E. J. and Reiss, S. P. (1990) The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing* **1**, 141-157.

[Haars88] Haarslev, V. and Moller, R. (1988) Visualization of Experimental Systems. *IEEE Workshop on Visual Languages.* Pittsburgh, PA, October 10-12, 1988. pp. 175-182.

[Haars90] Haarslev, V. and Moller, R. (1990) VIPEX: Visual Programming of Experimental Systems. In: *Visual Languages and Visual Programming* (S.-K. Chang, ed.) Plenum Publishing Corporation, 1990.

[Haeb88] Haeberli, P. E. (1988) ConMan: A Visual Programming Language for Interactive Graphics. *Proceedings ACM SIGGRAPH '88*, printed as *ACM Computer Graphics* **22**. Atlanta, GA, August 1-5, 1988. pp. 103-111.

[Hils91] Hils, D. D. (1991) DataVis: A Visual Programming Language For Scientific Visualization. *Proceedings 1991 ACM Computer Science Conference.* San Antonio, TX. March 5-7, 1991. pp. 439-448.

[Hils92] Hils, D. D. (1992) Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing* **3**, March 1992, 69-101.

[Hira86] Hirakawa, M., Monden, N., Yoshimoto, I., Tanaka, M., and Ichikawa, T. (1986) HI-VISUAL: A Language Supporting Visual Interaction in Programming. In: *Visual*

Languages, (Chang, S.K., Ichikawa, T., and Ligomenides, P. A., eds.) New York: Plenum Press.

[Hira87] Hirakawa, M., Iwata, S., Yoshimoto, I., Tanaka, M., and Ichikawa, T. (1987) HI-VISUAL Iconic Programming. *IEEE Workshop on Visual Languages.* Linkoping, Sweden, August 19-21, 1987. pp. 305-314.

[Hira88] Hirakawa, M., Iwata, S., Tahara, Y., Tanaka, M., and Ichikawa, T. (1988) A Framework for Construction of Icon Systems. *IEEE Workshop on Visual Languages.* Pittsburgh, PA, October 10-12, 1988. pp. 70-77.

[Hira89] Hirakawa, M., Yoshimi, M., Tanaka, M., and Ichikawa, T. (1989) A Generic Model for Constructing Visual Programming Systems. *IEEE Workshop on Visual Languages.* Rome, Italy, October 4-6, 1989. pp. 124-129.

[Hira90] Hirakawa, M., Yoshimi, M., and Ichikawa, T. (1990) A Universal Language System for Visual Programming. *IEEE Workshop on Visual Languages.* Skokie, IL, October 4-6, 1990. pp. 156-161.

[Hira91] Hirakawa, M., Nishimura, Y. Nishimura, Kado, M., and Ichikawa, T. *IEEE Workshop on Visual Languages.* Kobe, Japan. October 8-11, 1991. pp. 254-259.

[Holt90] Holt, C. M. (1990) *viz*: A Visual Language Based on Functions. *IEEE Workshop on Visual Languages.* Skokie, IL, October 4-6, 1990. pp. 221-226.

[Holt91] Holt, C. M. (1991) Visualizing an Algebra of Objects. *IEEE Workshop on Visual Languages.* Kobe, Japan. October 8-11, 1991. pp. 225-230.

[Inga88] Ingalls, D., Wallace, S., Chow, Y. Y., Ludolph, F., and Doyle, K. (1988) Fabrik: A Visual Programming Environment. *Proceedings ACM OOPSLA '88.* September 1988, pp. 176-190.

[Jind90] Jindrich, W. A. (1990) FOIBLE: A Framework For Visual Programming Languages. M.S. Thesis, Department of Computer Science, University of Illinois. Urbana, IL 61801.

[John88] Johnson, R. E., Graver, J. O., and Zurawski, L. W. (1988) TS: An Optimizing Compiler For Smalltalk. *OOPSLA'88*, printed as *SIGPLAN Notices*, **23**. November 1988. pp. 18-26.

[John92] Johnson, R. E. (1992) Documenting Frameworks Using Patterns. To appear in *OOPSLA '92*.

[Kimu86a] Kimura, T. D., Choi, J. W., and Mack, J. M. (1986) A Visual Language for Keyboardless Programming. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, MO 63130.

[Kimu86b] Kimura, T.D. (1986) Determinacy of Hierarchical Dataflow Model: A Computation Model For Visual Programming. Technical Report WUCS-86-5, Department of Computer Science, Washington University, St. Louis, MO 63130.

[Kimu86c] Kimura, T. D., and McLain, P. (1986) Show and Tell User's Manual. Technical Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, MO 63130.

[Kimu86d] Kimura, T. D. (1986) Show and Tell Sample Programs. Technical Report, Department of Computer Science, Washington University, St. Louis, MO 63130.

[Kodo91] Kodosky, J., MacCrisken, J., and Rymar, G. (1991) Visual Programming Using Structured Data Flow. *IEEE Visual Workshop on Visual Languages*. Kobe, Japan. October 8-11, 1991. pp. 34-39.

[Lau91] Lau-Kee, D., Billyard, A., Faichney, R., Kozato, Y., Otto, P., Smith, M., and Wilkinson, I. (1991) VPL: An Active, Declarative Visual Programming System. *IEEE Visual Workshop on Visual Languages*. Kobe, Japan. October 8-11, 1991. pp. 40-46.

[Levi86] Levitt, D. (1986) Hook Up: An Iconic, Real-Time Data-Flow Language for Entertainment. Unpublished technical note, MIT Media Lab. Available from hip software corporation, 117 Harvard St. #3, Cambridge, MA 01239.

[Luca88] Lucas, H. C., Jr., Walton, E. J., and Ginzberg, M. J. (1988) Implementing Packaged Software. *MIS Quarterly* **12**, 537-549.

[Ludo88] Ludolph, F., Chow, Y. Y., Ingalls, D., Wallace, S., and Doyle, K. (1988) The Fabrik Programming Environment. *IEEE Workshop on Visual Languages*. Pittsburgh, PA, October 10-12, 1988. pp. 222-230.

[Matw85] Matwin, S., and Pietrzykowski, T. (1985) PROGRAPH: A Preliminary Report. *Computer Languages* **10**, 91-126.

[McCo87] McCormick, B. H., et. al. (eds.) (1987) Visualization in Scientific Computing. *ACM Computer Graphics*, 21.

[Myer90] Myers, B. A. (1990) Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* **1**, 97-123.

[Najo90] Najork, M. A., and Golin, E. (1990) Enhancing Show-and-Tell With a Polymorphic Type System and Higher-order Functions. *IEEE Workshop on Visual Languages*. Skokie, IL, October 4-6, 1990. pp. 215-220.

[Najo91] Najork, M. A., and Kaplan, Simon M. (1991) The CUBE Language. *IEEE Workshop on Visual Languages*. Kobe, Japan, October 8-11, 1991. pp. 218-224.

[Nati87] National Instruments Corporation (1987) *LabVIEW: A Demonstration*. National Instruments Corp., 12109 Technology Blvd., Austin, TX 78727-6204.

[NCSA89a] (1989) NCSA Image manual, version 2.1. National Center for Supercomputing Applications, Software Development, 405 N. Mathews Ave., Urbana, IL 61801. Also listed in the NCSA Technical Resources Catalog.

[NCSA89b] (1989) NCSA DataScope manual, version 1.1. NCSA Software Development, 405 N. Mathews Ave., Urbana, IL 61801.

[Niel89] Nielson, G. M. (editor). (1989) IEEE Computer special issue on Scientific Visualization, 10-101.

[OSU89] (1989) apE: The Animation Production Environment — Version 1.1 Release Notes. The Ohio Supercomputer Graphics Project, The Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, OH 43212-1154.

[Palm92] Palmer, T. C. (1992) A Language For Molecular Visualization. *IEEE Computer Graphics and Applications* **12**, 23-31.

[Raed84] Raeder, G. (1984) Programming in Pictures. Ph.D. dissertation, University of Southern California, Los Angeles, CA. Technical Report TR-84-318.

[Raed85] Raeder, G. (1985) A Survey of Current Graphical Programming Techniques. *IEEE Computer* **18**, 11-25.

[Rasu90] Rasure, J., Argiro, D., Sauer, T., and Williams, C. S. (1990) A Visual Language and Software Development Environment for Image Processing. *International Journal of Imaging Systems and Technology* **2**, 183-199.

[Rasu91] Rasure, J., and Williams, C. S. (1991) An Integrated Data Flow Language and Software Development Environment. *Journal of Visual Languages and Computing* **2**.

[Rowe88] Rowell, J. (1988) *Picture Perfect: Color Output In Computer Graphics.* Part No. 070-6559-00; Tektronix, Inc., Wilsonville Industrial Park, P.O. Box 1000, Wilsonville, Oregon 97070, 1988.

[Sale89] Salem, J.B. and Sethian, J. A. (1989) Animation of Interactive Fluid Flow Visualization Tools on a Data Parallel Machine. *International Journal of Supercomputer Applications* **3**, 10-39.

[Same89] Samet, Hanan. (1989) *The Design and Analysis of Spatial Data Structures.* Reading, Mass.: Addison-Wesley.

[Same90] Samet, Hanan. (1990) *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Reading, Mass: Addison-Wesley.

[Sant90] Santori, M. (1990) An Instrument That Isn't Really. *IEEE Spectrum* **27** August 1990. pp. 36-39.

[Shir89] Shirley, P., and Neeman, H. (1989) Volume Visualization at the Center for Supercomputing Research and Development. Technical Report 849, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

[Smit88] Smith, D. N. (1988) Visual Programming in the Interface Construction Set. *IEEE Workshop on Visual Languages.* Pittsburgh, PA, October 10-12, 1988. pp. 109-120.

[Tani90a] Tanimoto, S. L. (1990) VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing* **1**, 127-139.

[Tani90b] Tanimoto, S. L. (1990) Towards a Theory of Progressive Operators For Live Visual Programming Environments. *IEEE Workshop on Visual Languages.* Skokie, IL, October 4-6, 1990. pp. 80-85.

[Trei90] Treinish, L. A. (1990) Panel on Interaction Issues in Visualization: Requirements, Techniques, and Devices. In *Proceedings of IEEE Visualization '90.* San Francisco, CA, October 23-26, 1990. pp. 393-400.

[Upso89] Upson, C., Faulhaber, Jr., T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., and van Dam, A. (1989) The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications* **9**, July 1989. pp. 30-42.

[Vand90] VandeWettering, M. (1990) The Application Visualization System–AVS 2.0. *Pixel* **1** July/August 1990. pp. 30-33. Published by Pixel Communications, Inc., 56 Herman Court, Watsonville, CA 95076.

[Vlis89] Vlissides, J. M., and Linton, M. A. (1989) Unidraw: A Framework for Building Domain-Specific Graphical Editors. *Proceedings ACM User Interface Software and Technologies '89 Conference.* November 1989, pp. 81-94.

[Vlis90] Vlissides, J. M. (1990) Generalized Graphical Object Editing. Ph.D. dissertation, Stanford University.

[Vose86] Vose, G. M., and Williams, G. (1986) LabVIEW: Laboratory Virtual Instrument Engineering Workbench. *Byte* **11**, 84-92.

[Weth91] Wetherbe, J. C. (1991) Executive Information Requirements: Getting It Right. *MIS Quarterly* **15**, 51-65.

[Wilh90] Wilhelmson, R. B., Jewett, B. F., Shaw, C., Wicker, L. J., Arrott, M., Bushnell, C. B., Bajuk, M., Thingvold, J., and Yost, J. B. (1990) A Study of the Evolution of a Numerically Modeled Severe Storm. *International Journal of Supercomputer Applications* **4**, 20-36.

[Will90] Williams, C. S., and Rasure, J. R. (1990) A Visual Language for Image Processing. *IEEE Workshop on Visual Languages.* Skokie, IL, October 4-6, 1990. pp. 86-91.

[Yeun88] Yeung, R. (1988) MPL—A Graphical Programming Environment For Matrix Processing Based On Logic and Constraints. *IEEE Workshop on Visual Languages.* Pittsburgh, PA, October 10-12, 1988. pp. 137-143.

[Yosh86] Yoshimoto, I., Monden, N., Hirakawa, M., Tanaka, M., and Ichikawa, T. (1986) Interactive Iconic Programming Facility in HI-VISUAL. *IEEE Computer Society Workshop on Visual Languages.* June 1986. pp. 34-41.

APPENDIX A

Obtaining Color Figures

This thesis contains black-and-white photocopies of many original figures that were in color. (Some of the original figures were themselves black-and-white). To obtain a set of those original figures that were in color (11, 8.5 x 11" color sheets), send a check for $21.45, payable to Kinko's; a note to Larry Cope asking for the manilla folder of color figures from Dan Hils's Ph.D. thesis, and giving the thesis title: "A Visual Programming Language for Visualization of Scientific Data"; and your U.S.-mail address, to:
Kinko's
c/o Larry Cope
613 S. Wright St.
Champaign, Illinois 61820.
Kinko's phone number is (217) 398-0003. Kinko's will mail a set of color photocopies of the original color figures to you. The $21.45 is $1.95 for each of the 11 color photocopies, plus $1.00 for postage and handling. The "c/o Larry Cope" line is important: if you forget it, the other Kinko's personnel will not know what color originals you are referring to.

VITA

Daniel D. Hils attended John Carroll University in Cleveland, Ohio, where he received a B.S. degree in Computer Science. He graduated from John Carroll University in 1985, summa cum laude and from the Honors Program.

Dr. Hils received the M.S. degree in Computer Science in 1985 from the University of Illinois at Urbana-Champaign.

For his doctoral work, Dr. Hils's advisor was Professor Ralph Johnson. Dr. Hils received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1992.

Dr. Hils is currently employed at IBM's Information Design and Development division in Research Triangle Park, North Carolina. His interests include multimedia systems, visual programming languages, visualization of scientific data, computer graphics, virtual reality, and object-oriented programming languages and object-oriented design.