# VISUALIZING SCIENTIFIC COMPUTATIONS: A SYSTEM BASED ON LATTICE-STRUCTURED DATA AND DISPLAY MODELS

by

WILLIAM LOUIS HIBBARD

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1995

# VISUALIZING SCIENTIFIC COMPUTATIONS: A SYSTEM BASED ON LATTICE-STRUCTURED DATA AND DISPLAY MODELS

William Louis Hibbard

Under the supervision of Professor Charles R. Dyer

At the University of Wisconsin-Madison

**Abstract**

In this thesis we develop a system that makes scientific computations visible and enables physical scientists to perform visual experiments with their computations. Our approach is unique in the way it integrates visualization with a scientific programming language. Data objects of any user-defined data type can be displayed, and can be displayed in any way that satisfies broad analytic conditions, without requiring graphics expertise from the user. Furthermore, the system is highly interactive.

In order to achieve generality in our architecture, we first analyze the nature of scientific data and displays, and the visualization mappings between them. Scientific data and displays are usually approximations to mathematical objects (i.e., variables, vectors and functions) and this provides a natural way to define a mathematical lattice structure on data models and display models. Lattice-structured models provide a basis for integrating certain forms of scientific metadata into the computational and display semantics of data, and also provide a rigorous interpretation of certain expressiveness conditions on the visualization mapping from data to displays. Visualization mappings satisfying these expressiveness conditions are lattice isomorphisms. Applied to the data types of a scientific programming language, this implies that visualization mappings from data aggregates to display aggregates can always be decomposed into mappings of data primitives to display primitives.

These results provide very flexible data and display models, and provide the basis for flexible and easy-to-use visualization of data objects occurring in scientific computations.

Charles R. Dyer, Professor, Computer Sciences, University of Wisconsin - Madison

# Abstract

In this thesis we develop a system that makes scientific computations visible and enables physical scientists to perform visual experiments with their computations. Our approach is unique in the way it integrates visualization with a scientific programming language. Data objects of any user-defined data type can be displayed, and can be displayed in any way that satisfies broad analytic conditions, without requiring graphics expertise from the user. Furthermore, the system is highly interactive.

In order to achieve generality in our architecture, we first analyze the nature of scientific data and displays, and the visualization mappings between them. Scientific data and displays are usually approximations to mathematical objects (i.e., variables, vectors and functions) and this provides a natural way to define a mathematical lattice structure on data models and display models. Lattice-structured models provide a basis for integrating certain forms of scientific metadata into the computational and display semantics of data, and also provide a rigorous interpretation of certain expressiveness conditions on the visualization mapping from data to displays. Visualization mappings satisfying these expressiveness conditions are lattice isomorphisms. Applied to the data types of a scientific programming language, this implies that visualization mappings from data aggregates to display aggregates can always be decomposed into mappings of data primitives to display primitives.

These results provide very flexible data and display models, and provide the basis for flexible and easy-to-use visualization of data objects occurring in scientific computations.

## Acknowledgments

I am greatly indebted to my advisor, Chuck Dyer, for showing me a different way of thinking about Computer Science problems, and for his consistent good nature.

Special thanks are due to Amir Assadi, Miron Livny, Tom Reps and Greg Tripoli for taking the time to review this work as members of my thesis committee, and to Tom DeFanti for his input as a guest committee member.

I want to thank Larry Landweber for suggesting that I pursue a doctorate and for introducing me to Chuck Dyer, long after I thought graduate school was behind me forever. I also want to thank Francis Bretherton, Bob Fox and John Anderson, the directors of the Space Science and Engineering Center where I am employed, for their support and encouragement. I especially want to thank Verner Suomi, the founder of the Space Science and Engineering Center, for his profound positive influence on my life over a period of many years.

Special thanks are due to Brian Paul, my principal collaborator in system development, and to Andre Battaiola, Dave Santek and Marie-Francoise Voidrot-Martinez for their collaboration in systems development.

I am indebted to Bob Rabin, Roland Stull, Bob Aune, Wilt Sanders, Dick Edgar, Mike Botts, Chris Crosiar, and all the other users of our systems for their helpful suggestions for improving our systems.

Marriage to AJ is the best thing that has happened in my life. The real work of this thesis was done at our home, and thus was always pleasant. Thanks to my sweet mother for always encouraging education and to my father for teaching me to figure things out for myself. This thesis is dedicated to Laura and Tommy - thanks to Jeannie and John for bringing them into the world.

# Contents

**Abstract** ii

**Acknowledgments** iii

**1. Introduction** 1

1.1 Goals for Scientific Visualization 3

1.2 State of the Art in Scientific Visualization 10

1.2.1 The Data Flow and Object-Oriented Approaches 10

1.2.2 Data Models 13

1.2.3 Display Models 16

1.2.4 Automating the Design of Data Displays 18

1.3 Major Contributions 22

1.4 Thesis Outline 24

**2. System Design for Visualizing Scientific Computations** 25

2.1 A Scientific Computing Environment 25

2.2 Scientific Data 31

2.3 Scientific Displays 35

2.4 Mapping Data to Displays 37

**3. An Analysis of Mappings from Data to Displays** 40

3.1 An Analytic Approach Based on Lattices 41

3.1.1 Basic Definitions for Ordered Sets 42

3.1.2 Scientific Data Objects as Approximations of Mathematical Objects44

3.1.3 A Mathematical Structure Based on the Precision of Scientific data 46

3.1.4 Data Display as a Mapping Between Lattices 56

3.2 A Scientific Data Model 59

3.2.1 Interpreting the Data Model as a Lattice 64

3.2.2 Defining the Lattice Structure 66

3.2.3 Embedding Scientific Data Types in the Data Lattice 70

3.2.4 A Finite Representation of Data Objects 74

3.3 A Scientific Display Model 75

3.4 Scalar Mapping Functions 79

3.4.1 Structure of Display Functions 79

3.4.2 Behavior of Display Functions on Continuous Scalars 82

3.4.3 Characterizing Display Functions 85

3.4.4 Properties of Scalar Mapping Functions 87

3.5 Principles for Scientific Visualization 91

**4. Applying the Lattice Model to the Design of Visualization Systems 94**

4.1 Integrating Metadata with a Scientific Data Model 96

4.2 Interacting with Scientific Displays 105

4.3 Visualizing Scientific Computations 123

4.4 System Organization 144

**5. Applying the Lattice Model to Recursive Data Type Definitions** **148**

5.1 Recursive Data Type Definitions 148

5.2 The Inverse Limit Construction 149

5.3 Universal Domains 151

5.4 Display of Recursively Defined Data Types 153

**6. Conclusions** **155**

6.1 Main Contributions and Limitations 155

6.2 Future Directions 158

**A. Definitions for Ordered Sets** **161**

**B. Proofs for Section 3.1.4** **165**

**C. Proofs for Section 3.2.2** **170**

**D. Proofs for Section 3.2.3** **178**

**E. Proofs for Section 3.2.4** **186**

**F. Proofs for Section 3.4.1** **190**

**G. Proofs for Section 3.4.2** **204**

**H. Proofs for Section 3.4.3** **215**

**I. Proofs for Section 3.4.4** **223**

**Bibliography** **232**

# List of Figures

1.1. Image data displayed in four different ways. 6

1.2. Interactive display of the output of a numerical weather model. 8

1.3. The place of visualization in the computational process. 10

1.4. A simple rendering pipeline for three-dimensional graphics. 11

1.5. Bertin's display model. 17

2.1 The place of visualization in the computational process. 27

2.2. A snapshot of an executing shallow fluid simulation model. 29

3.1. Order relation of a continuous scalar. 48

3.2. Approximating real functions by arrays. 49

3.3. Order relation of arrays. 50

3.4. Least precise image in sequence of four. 51

3.5. Second image in sequence of four, ordered by precision. 52

3.6. Third image in sequence of four, ordered by precision. 53

3.7. Most precise image in sequence of four. 54

3.8. Meaning of the expressiveness conditions. 58

3.9. Order relation of a discrete scalar. 60

3.10. Order relation of tuples. 62

3.11. Embedding a tuple type into a lattice of sets of tuples. 65

3.12. Embedding an array type into a lattice of sets of tuples. 66

3.13. Defining an order relation on sets of tuples. 70

3.14. The roles of display scalars in a display model. 77

3.15. Mappings from scalars to display scalars. 82

3.16. The behavior of a display function on a continuous scalar. 85

4.1. An image displayed in a Cartesian coordinate system. 102

4.2. An image displayed in a spherical Earth coordinate system. 103

4.3. Three-dimensional radar data. 104

4.4. X-ray events from interstellar gas. 106

4.5. A *goes_sequence* object displayed as a terrain. 111

4.6. A *goes_sequence* object displayed in four different ways. 115

4.7. Three views of chaos. 120

4.8. Visualizing the computations of a bubble sort algorithm. 126

4.9. Visually experimenting with algorithms. 127

4.10. Visually tracing back to the causes of computational errors. 128

4.11. A close-up view of two regions of a *goes_partition* object. 132

4.12. A close-up view restricted to "cloudy" pixels. 133

4.13. Three *ir_image_partition* objects displayed as terrains. 134

4.14. Three *histogram* objects displayed as graphs. 135

4.15. A two-dimensional histogram of X-ray events. 140

4.16. Visualizing the three criteria used to select cumulus clouds. 143

4.17. VisAD system organization. 147

5.1. A type construction operator represented by a function. 151

# Chapter 1

# Introduction

Physical scientists observe nature, formulate laws to fit the observations, and predict future observations in order to test their laws. Mathematics is the language of observations, laws and predictions, but the complexity of modern science demands that mathematical calculations be automated using computers. The number of observations of nature dictates that they are analyzed by computer algorithms, and the number of computations required to predict nature dictates that predictions are made by numerical simulation models running on computers. Thus computers have become essential tools for scientists for both observing and simulating nature.

In spite of their essential role, computers are also barriers to scientific understanding. Unlike hand calculations, automated computations are invisible, and, because of the enormous numbers of individual operations in automated computations, the relation between an algorithm's input and output is often not intuitive. This problem was discussed in a report to the National Science Foundation (McCormack, DeFanti and Brown, 1987) and is illustrated by the behavior of meteorologists responsible for forecasting weather. Even in this age of computers, many meteorologists manually plot weather observations on maps and then draw iso-level curves of temperature, pressure and other fields by hand (special pads of maps are printed for just this purpose). Similarly, radiologists use computers to collect medical data, but are notoriously reluctant to apply image processing algorithms to those data. To these scientists with life and death responsibilities, computer algorithms are black boxes that increase rather than reduce risk.

The barrier between scientists and their computations is being bridged by *scientific visualization* techniques that make computations visible. Scientific visualization is itself a computational process that transforms the data objects of scientific computations into visible images on a computer display screen. Scientific visualization is difficult because of the variety and complexity of scientific data, because the

variety of scientific problems implies that scientists need to see the same data in many different ways, and because scientists need tools that are easy to use so that they can concentrate on understanding their computations rather than understanding their visualization tools.

The size of scientific data sets is often used to justify the development of scientific visualization, and it is true that scientists need to be able to see large data sets. However, the more important motive for visualization is the invisibility of automated computations. To see this, consider the volumes of satellite images of the Earth. A pair of GOES (Geostationary Operation Environmental Satellite) located at East and West stations over the U.S. generate one 1024 by 1024 image every four seconds. NASA's Earth Observing System, as planned, will generate about five 1024 by 1024 images per second. These data volumes are so large that they will overwhelm any scientist trying to look at them all. Furthermore, these images are quantitative measurements rather than just pictures. The real value of these images must be extracted by automated computations that can process the images faster than a person can coherently look at them. Thus the work of Earth scientists is to develop algorithms for this automated processing, and the proper role of visualization is helping scientists to understand how their algorithms work and how to improve them.

**1.1 Goals for Scientific Visualization**

Scientific data exist in a wide variety of structures. A few examples include two-dimensional images:

```
type image = array [row] of array [column] of radiance;
```

three-dimensional grids:

```
type grid = array [row] of array [column] of array [level] of temperature;
```

time sequences of images and grids:

```
type image_sequence = array [time] of image;
type grid_sequence = array [time] of grid;
```

images and grids with multiple values per pixel:

```
type multi_image = array [row] of array [column] of
                structure {ir_radiance; vis_radiance};
type multi_grid = array [row] of array [column] of array [level] of
                structure {pressure; temperature; humidity};
```

irregularly located data such as observations made by ships or aircraft:

```
type observations = array [index] of structure {latitude; longitude; altitude; pressure};
```

one-dimensional and multi-dimensional histograms derived from other data:

```
type histogram_1d = array [temperature] of count;
type histogram_2d = array [temperature] of array [pressure] of count;
```

and partitions of images and grids into spatial regions:

```
type image_partition = array [region] of image;
type grid_partition = array [region] of grid;
```

Furthermore, physical systems are observed by collections of instruments so the observed state of a physical system is a complex combination of data sensed by different types of instruments. Similarly, simulations generate complex combinations of data describing interacting physical systems (e.g., atmospheric physics and chemistry, ocean physics and chemistry, and land and ocean surface processes). Scientific data are made more complex because of scientists' need to precisely document where, when and how they were obtained (this documentation is a form of *metadata*, and must be considered as part of scientific data). The first goal of this thesis is to develop visualization techniques that

1. Can be applied to the data of a wide variety of scientific applications.

Scientists need to see the same data displayed in different ways, depending on what kinds of information they are looking at. For example, Figure 1.1 shows a time sequence of multi-variate image data displayed in four different ways. The upper-left window shows radiance values as colors, which is appropriate for seeing spatial patterns and textures. The upper-right window shows infrared radiances as a

terrain (colored by visible radiances), appropriate for seeing slopes. The time sequence can be animated in the upper-right and upper-left windows, which is appropriate for seeing motion. Alternatively, the time sequence is stacked up along the vertical axis in the lower-right window, which is appropriate for looking closely at rates of motion and changes in shape and intensity. Information about the spatial locations of pixels is not shown in the lower-left window, producing a colored three-dimensional scatter diagram which is appropriate for seeing correlations among infrared radiance, visible radiance, variance and texture (variance and texture are derived from infrared radiance). Each of the four views presented in Figure 1.1 is appropriate for seeing a different aspect of the same data. More generally, the primary reason scientists use scientific visualization is to find unexpected patterns in data, since expected patterns can just be measured and characterized by statistical calculations applied to data. And flexibility in the ways that data are displayed is critical in the search for unexpected patterns. Thus the second goal of this thesis is to develop visualization techniques that

2. Can produce a wide variety of different visualizations of data appropriate for different needs.

Figure 1.1. A time sequence of multi-variate image data displayed in four different ways. (color original)

Because of the large volumes of scientific data it is often impossible to display a data object in a single image or even a single animation sequence. Instead, scientists need to interactively explore large data objects. For example, Figure 1.2 shows a snapshot of an interactive animated display of the output of a numerical weather model. The white object is a balloon seven kilometers high in the shape of a squat chimney that floats in the air above a patch of tropical ocean. The purpose of the numerical simulation is to verify that, once air starts rising in the chimney, the motion will be self-sustaining and create a perpetual rainstorm. The vertical color slice shows the distribution of heat (and when animated shows the flow of heat), the yellow streamers show the corresponding flow of air up through the chimney, and the blue iso-surface shows the precipitated cloud ice (a cloud water iso-surface would obscure the view down the chimney, so it is not shown in this snapshot). Viewers of this visualization can interactively move the color slice in the three-dimensional box of atmosphere, can interactively release new streamers in the air flow, can interactively change the value of the cloud ice iso-surface, and can rotate and zoom the box in three dimensions. They can choose different combinations of fields to display, and can choose the ways that each field is depicted (e.g., color slice, iso-surface, contour slice). Such interactivity is critical for allowing scientists to search through large
amounts of data for unexpected patterns. Hence, the third goal of this thesis is to develop visualization techniques that

3. Enable users to interactively alter the ways data are viewed.

Figure 1.2. A snapshot of an interactive animated display of the output of a numerical weather model. (color original)

Because visualization is used for communicating results of observations and computations to scientists, they need to be able to control it themselves (that is, they cannot delegate expertise with visualization to support staff). In order not to distract scientists from the difficult task of understanding data, visualization must be easy to control. Thus the fourth goal of this thesis is to develop visualization techniques that

4. Require minimal effort by scientists.

As stated at the start of this section, the rationale for visualization is scientists' need to see the results of computations. Thus visualization is intimately connected with computation. Just as the complexity of data requires that the visualization process should be interactive, the complexity of computation requires that the overall computational process, which includes visualization, should be interactive. Figure 1.3 illustrates the interactive cycle of the computation process. If these three activities are done in separate software systems, then scientists must repeatedly switch between systems and manage the movement of information between these systems. This user overhead can be reduced by integrating all three activities in one system. Furthermore, visualization can be especially useful during program execution, allowing users to dynamically monitor intermediate results of computations and respond by immediately adjusting parameters of those computations. This is sometimes called *computational steering*. The fifth goal of this thesis is to develop visualization techniques that

5. Can be integrated with a scientific programming environment.

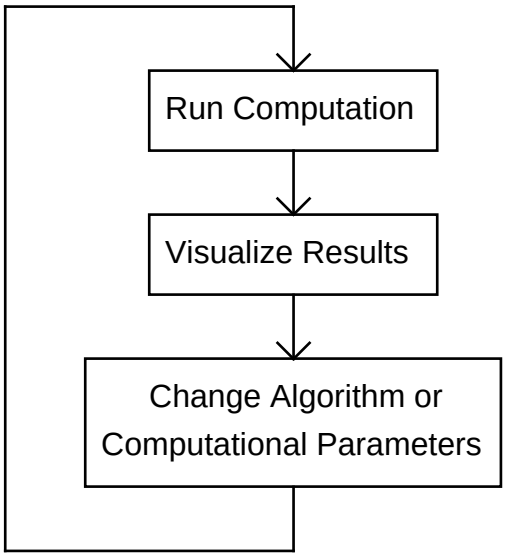


Figure 1.3. The place of visualization in the computational process.

### 1.2 State of the Art in Scientific Visualization

Here we consider the state of the art in scientific visualization and how well current techniques achieve our goals.

#### 1.2.1 The Data Flow and Object-Oriented Approaches

Visualization research has focused primarily on developing specialized visualization techniques suited to specific types of data. However, some research has sought common patterns in the ways that displays are computed. For example, the *rendering pipeline* is a widely applicable abstraction for the ways that data are transformed into displays. Figure 1.4 illustrates a simple rendering pipeline:

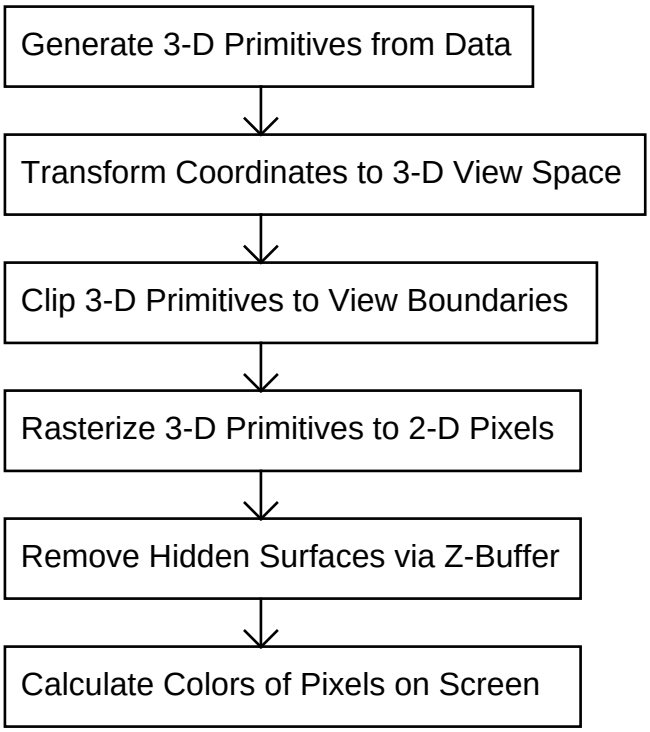


Figure 1.4. A simple rendering pipeline for three-dimensional graphics.

The FRAMES system abstracted the rendering pipeline to let users specify display processes as sequences of UNIX filters (Potmesil and Hoffert, 1987). The GRAPE system introduced branching into these data transformations and let users define display processes as acyclic graphs of modules (Nadas and Fournier, 1987). The ConMan system provided a graphical user interface for specifying display processes as networks of modules (Haeberli, 1988). This idea has been adopted as the basis of several widely-used data flow visualization systems, such as AVS (Upson et al., 1989) and Khoros (Rasure et al., 1990). These data flow systems provide large libraries of modules that implement basic computational and display operations, and also provide graphical user interfaces for synthesizing complex visualization algorithms from these module libraries.

The recognition that different kinds of displays are generated by similar sets of operations also led to the object-oriented approach to synthesizing visualization mappings. The object-oriented approach uses

inheritance and polymorphism to exploit the common properties and natural hierarchy of data displays. The Powervision system, for example, used an object-oriented language to support interactive development of image processing algorithms (McConnell and Lawton, 1988). The system defined a set of primitive generic functions for accessing data objects (for example, for iterating over parts of objects, for checking boundary conditions, etc.). Algorithms for synthesizing displays were expressed in terms of these generic functions. As users defined new object classes they could apply existing display algorithms to those classes as long as the new classes included definitions for the generic functions for accessing data objects.

The SuperGlue system was developed as a programming environment for developing scientific visualization applications based on Scheme, C and the GNU Emacs editor (Hultquist and Raible, 1992). It defined a class hierarchy for various types of scientific data objects and displays. User extensions to this class hierarchy could take advantage of inheritance and polymorphism to simplify their definition.

The VISAGE system implemented a hierarchy of over 500 classes for both process objects and data objects (Schroeder, Lorenson, Mantanaro and Volpe, 1992). The process objects implemented the visualization process as data flow networks of simpler processes. The data objects implemented a variety of scientific data organizations and a variety of display organizations.

While these systems have been useful to scientists, their approach to generality is through the enumeration of data types and the enumeration of display techniques. Thus these systems have become very large and complex. Furthermore, scientists must spend considerable effort to produce visualizations using these systems. While scientists could explore different ways of displaying data by interactively changing the data flow networks that transform data into displays, in practice they do not. Rather, support staff design data flow networks and scientists use them to generate fixed types of displays from data. Similarly for the object-oriented systems. The developers of the VISAGE system described one visualization application of their system that required 12,000 lines of code specific to the application. Scientists need visualization techniques that let them change the way that they look at data without understanding complex programs or data flow networks.

### 1.2.2 Data Models

Rather than approaching generality by enumerating data types and display techniques, we can achieve generality through the abstraction of data and displays. That is, by developing broadly applicable abstract models of scientific data and displays, we can systematically study the ways that visualization processes transform data into displays. A *data model* defines a set of data objects, the way data objects are organized in the set, and operations on the data objects (often by reference to their internal structures). Data models have been the subject of several recent workshops and publications (Treinish, 1991; Haber, 1991; Robertson et al., 1994, Lee and Grinstein, 1994). *Display models* are similar to data models and are discussed in the next section.

The requirements for a scientific data model can be understood in terms of the role of data in science. Scientists design mathematical models of nature. These models identify numerical variables (e.g., *time*, *altitude*, *temperature*) and functional relations between these variables (e.g., *temperature* as a function of *time*). These models define the states of nature as vectors of variables and functions. For example, the state of a point in the atmosphere is a vector of variables such as *temperature*, *pressure*, *humidity* and *wind velocity*, and the state of the entire atmosphere is a vector of functions. We use the term *mathematical objects* to denote the numbers, functions and vectors of mathematical models. When scientists want to use their mathematical models to analyze a set of observations, or to simulate a physical system, they implement their models as computer programs. Mathematical objects are represented by *scientific data objects* in these implementations, and therefore a *scientific data model* should reflect the ways that scientific data objects represent mathematical objects. There are many ways to define scientific data models. However, any scientific data model will incorporate the following components:

1. The types of *primitive values* occurring in data objects. These represent primitive variables defined in mathematical models of nature. Thus a data model may define a *floating-point* type to represent

real variables such as *time* and *temperature*, may define an *integer* type to represent integer variables such as an *event_count*, and may define a *string* type to represent names such as city or state names. A rigorous data model specifies the relations and operations defined on values of primitive types. The definition of a type of primitive value may include arithmetical operations, string operations, an order relation, a metric or a topology.

2. The ways that primitive values are *aggregated* into data objects. These aggregates represent complex mathematical objects, such as vectors, functions, vectors of functions, and so on. There are a variety of approaches to defining data aggregates. In the C programming language, vectors can be represented by *structures*, functions can be represented by *arrays*, and *pointers* can be used to define complex networks of values. Most programming languages provide a few simple data structuring rules that can be combined to define a wide variety of data aggregates. On the other hand, most scientific analysis and visualization systems support specific types of aggregates based on particular application needs. These may include two-dimensional images (as generated by satellites and other observing systems), three-dimensional grids (as generated by numerical simulations and some observing systems), and vector and polygon lists (generated by applying visualization operators to images and grids, and by map makers).

3. *Metadata* about the relation between data and the physical things they represent. For example, given a meteorological temperature value, metadata includes the fact that it is a temperature, its scale (Fahrenheit, Kelvin, etc.), its spatial and temporal location in the Earth's atmosphere, and whether it is a point sample or an average over space and time. Temperature values have limited accuracy, whether sensed by an instrument or computed by a weather model, and an estimate of accuracy is another form of metadata. Because instruments and observing systems are fallible, an expected data value may not be defined at all, so *missing* data indicators are a form of metadata. If a temperature

is observed by an instrument, there may be metadata about the instrument (for example, aperture, pointing direction, filters, etc.). If a temperature is computed from other values, there may be metadata about the algorithm used to compute it and the source of the algorithm's inputs.

The term *metadata* has several different meanings. It is sometimes denotes information about the organization of data, in which case it may be called *syntactic metadata*. Here it denotes information about the meaning of data, and may be called *semantic metadata*. We can think of metadata as secondary data that are critical to the usefulness of primary data. For example, while a satellite image may primarily consist of an array of pixel radiance values, those data are scientifically useless without other arrays that specify the Earth locations of pixels, how pixel values correspond to physical radiances, and so on.

### 1.2.3 Display Models

Just as we can define systematic models of scientific data, we can define systematic models of scientific displays. In particular, it is useful to note that computer programs generate displays in the form of data objects. Bertin's detailed display model, first published in 1967, illustrates how a display model addresses the issues of primitives and aggregates (Bertin, 1983). Bertin defined a display as an aggregate of *graphical marks*, and identified eight *primitive variables* of a graphical mark: two spatial coordinates of the mark in a graphical plane (he restricted his attention to static two-dimensional graphics), plus the mark's size, value, texture, color, orientation, and shape. Bertin defined diagrams, networks and maps as spatial aggregates of graphical marks. Figure 1.5 illustrates Bertin's display model.

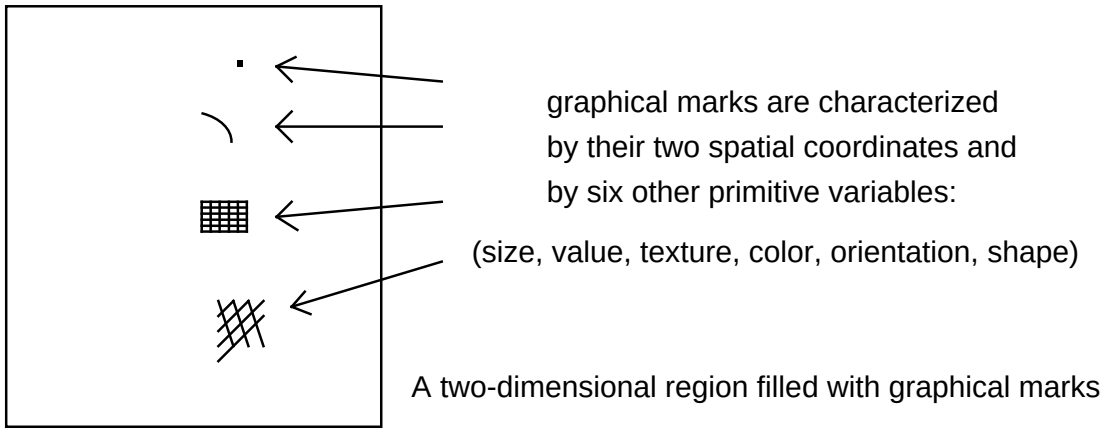


Figure 1.5. Bertin's display model. He modeled displays as sets of graphical marks in a two-dimensional spatial region.

Bertin's display model was limited to static two-dimensional displays. This corresponds to what can be physically displayed on a two-dimensional screen at one time. However, computer-generated displays generate the illusion of three dimensions and show motion by changing screen contents at short intervals. We can even regard various forms of user interaction as an integral part of the display. Thus we distinguish between physical and logical display models. We let *V'* denote the set of physical displays, which are two-dimensional and static, and we let *V* denote the set of logical displays, which are three-dimensional, animated and interactive. The mapping $RENDER : V \rightarrow V'$ includes traditional graphics operations such as iso-surface generation, volume rendering, projection from three to two dimensions (rotate, zoom and translate), clipping, hidden-surface removal, shading, compositing, and animation (these operations could be implemented in a rendering pipeline, as illustrated in Figure 1.4). A changing set of mappings, $RENDER : V \rightarrow V'$, expresses the three-dimensional, animated, interactive nature of logical displays in *V*.

Visualization is a process that maps data objects to displays. We let *U'* denote a set of mathematical objects, and we let *U* denote a set of data objects used to represent them. Then the overall visualization process may be viewed as a sequence of mappings $U' \rightarrow U \rightarrow V \rightarrow V'$. The mapping from *U'* to *U* expresses the way that scientists implement their mathematics on computers, and the mapping from *V* to *V'* is the generally well understood physical display generation process (Foley and Van Dam, 1982; Lorensen and Cline, 1987). Thus we will concentrate our interest on the mapping $D : U \rightarrow V$. In order to optimize the generality of visualization techniques to different scientific applications, we seek scientific data models *U* whose primitive values are defined in terms of abstract mathematical properties, whose aggregates are constructed using a few simple rules that can be combined in complex ways, and that integrate a variety of metadata. We also seek display models *V* that are abstract and that include interactive displays.

While the proper abstractions for *U* and *V* are necessary for display techniques that are flexible and easy to use, the proper abstraction for the mapping *D* is also necessary. In the next section we describe efforts to automate the choice of this mapping.

### 1.2.4 Automating the Design of Data Displays

As described in Section 1.2.1, the object-oriented and data flow approaches define natural methodologies for designing programs (or data flow diagrams) for transforming data into displays, but they still require considerable programming effort from their users. In response to scientists' need for visualization techniques that are easy to use, there have been a variety of efforts to automate the design of algorithms for producing data displays. This goal is often called *automating the design of data displays*, since the research focuses on automating the choice among the many different ways of displaying the same data.

Mackinlay sought to automate the design of displays for data from relational database systems (Mackinlay, 1986). His technique combined a relational data model with Bertin's display model. A

relation is a set of tuples of values. Sets of primitive values called *domains* are defined for each position in a relation's tuples. Mackinlay classified domains as nominal (without an order relation), ordinal (with an order relation but without a metric or arithmetical operations) or quantitative (with a metric and arithmetical operations). These primitive values are aggregated into sets of tuples to form relations. Mackinlay's data model also allowed functional dependencies to be defined between the domains of a relation (these are restrictions on the sets of tuples that may form relations).

Mackinlay modeled displays as sentences in a graphical language. Sentences were sets of 2-tuples, where each tuple pairs a graphical mark with a two-dimensional screen location. He also attached attributes to graphical marks for specifying their size, color, orientation, etc. The values of these attributes are similar to the primitive values Bertin used for graphical marks. Thus, in Mackinlay's model a display could be interpreted as a set of tuples, where each tuple contains two screen coordinates and the values of the various attributes of a graphical mark.

Mackinlay defined expressiveness and effectiveness criteria for the mapping from data relations to display sentences. The expressiveness criteria require that a display sentence:

1. Encodes all the facts in a set (that is, the set of facts about a data relation), and

2. Encodes only the facts in a set.

The effectiveness criteria provide a way to choose between different display sentences that satisfy the expressiveness criteria. For example, an effectiveness criterion may specify that quantitative information is easier to perceive when encoded as spatial position rather than as color. Mackinlay also solicited visualization goals from the user. Effectiveness criteria and visualization goals were expressed formally in terms of predicates and functions applied to relational data and display sentences. These were used as the basis for a backtracking search for an optimal display.

Mackinlay's display model was static and two-dimensional and therefore too limiting for scientific visualization. Furthermore, while the relational model can, in theory, be used for scientific data, it does not naturally fit the ways that scientific data are aggregated. Robertson (Robertson, 1991), Senay and Ignatius (Senay and Ignatius, 1991; Senay and Ignatius, 1994), and Beshers and Feiner (Beshers and Feiner, 1992) all sought automated techniques for designing displays for scientific data.

Robertson's data model classified primitive values as either nominal or ordinal. Nominal values were further classified as single or multiple valued (that is, sets of values) and ordinal values were classified as discrete or continuous (this is a classification of the topology of primitive value sets). Primitive values were aggregated as distributions over an *n*-dimensional space. Robertson modeled displays as two-dimensional and three-dimensional surfaces and their attributes (for example, color and texture). His methodology solicited a set of visualization goals from the user, in terms of the scales of the user's interest (that is, point, local or global) in different variables, and in terms of the user's interest in correlations between various pairs of variables. Data displays were generated by matching data attributes and relations to display attributes and relations, according to the user's visualization goals.

Senay and Ignatius' data model classified primitive values as qualitative or quantitative, and aggregated primitive values as functional dependencies between variables. Their data model included metadata for coordinate systems and data sampling. Senay and Ignatius modeled displays using Bertin's graphical marks, and using specific aggregates of marks (for example, icons and iso-surfaces). These were further classified as to whether they encoded a single variable or multiple variables. Displays were generated by applying production rules for matching data characteristics with display characteristics.

Beshers and Feiner's data model consisted of functions from one set of real variables to another. Their display model sought to overcome the limitation to three spatial axes by embedding small spatial coordinate systems (that is, small sets of graphical axes) within larger spatial coordinate systems. Their display model formalized interactive exploration of data by allowing the user to move small coordinate

systems around within larger coordinate systems. Their technique searched through a large set of possible designs, evaluating them based on a set of user-defined visualization tasks.

While all of these efforts sought to automate the design of displays, their display models were limited to specific types of displays and they enumerated specific display techniques as the search spaces for their automated techniques. That is, their focus was to automate the user's task of choosing among enumerated sets of visualization techniques. In the next section we describe an alternative approach that defines certain general analytic conditions on the mapping from data to displays, and then derives visualization mappings that satisfy those conditions.

In each of the previous automated approaches described above, displays were designed based on information about visualization goals provided by the user. Obviously, some form of user input is necessary for users to be able to control display design. However, user interface issues are notoriously complex and it is not obvious that an encoding or parameterization of visualization goals is the most effective way for users to control visualization systems. It may be most effective to allow users to make their own translation from their goals to some other form of controls over visualization. In particular, an interactive system that lets users experiment with various ways of displaying their data may be more effective than an automated system. An interactive system enables users to experiment with small changes to their display controls and to see the effect of those changes on the way that their data are displayed. Such experimentation is also often the fastest way for scientists to learn how a visualization system works.

### 1.3 Major Contributions

The main contributions of this thesis can be summarized as follows:

1. Development of a system for scientific visualization that enables a wide variety of visual experiments with scientific computations. This system integrates visualization with a scientific programming language that can be used to express scientific computations. This programming language supports

a wide variety of scientific data types and integrates common forms of scientific metadata into the computational and display semantics of data. Any data object defined in a program in this language can be visualized in a wide variety of ways during and after program execution. The controls for data display are simple and independent of data type. Displays are controlled by a set of simple mappings rather than program logic. These mappings are independent of data type and separate from a user's scientific programs, which is a clear distinction from previous visualization systems that require scientists to embed calls to visualization functions in their programs. Furthermore, computation and visualization are highly interactive. In particular, the selection of data objects for display and the controls for how they are displayed are treated like any other interactive display control (e.g., interactive rotation). Previous visualization systems require a user to alter his program in order to make such changes. The generality, integration, interactivity and ease-of-use of this system all enhance the user's ability to perform visual experiments with their algorithms.

2. Introduction of a systematic approach to analyzing visualization based on lattices. We define a set $U$ of data objects and a set $V$ of displays and show how a lattice structure on $U$ and $V$ expresses a fundamental property of scientific data and displays (namely that they are approximations to the physical world). The visualization repertoire of a system can be defined as a set of mappings of the form $D : U \rightarrow V$. It is common to define a system's visualization repertoire by enumerating such a set of functions. However, an enumerated repertoire is justified only by the tastes and experience of the people who decide what functions to include in the set. In contrast, we interpret certain well-known expressiveness conditions on the visualization mapping $D : U \rightarrow V$ in terms of a lattice structure, and define a visualization repertoire as the set of functions that satisfy those conditions. Such a repertoire is justified by the generality of the expressiveness conditions. We show that visualization mappings satisfy these conditions if and only if they are lattice isomorphisms. Lattice

structures can be defined for a wide variety of data and display models, so this result can be applied to analyze visualization repertoires in a wide variety of situations.

3. Demonstration of a specific lattice structure that unifies data objects of many different scientific types in a data model *U*, and demonstration that the same lattice structure can express interactive, animated, three-dimensional displays in a display model *V*. These models integrate certain kinds of scientific metadata into the computational and display semantics of data. In the context of these scientific data and display models, we show that the expressiveness conditions imply that mappings of data aggregates to display aggregates can always be factored into mappings of data primitives to display primitives. We show that our display mappings are complete, in the sense that we characterize all mappings satisfying the expressiveness conditions.

### 1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2 we describe the architecture of a system for scientific visualization based on the goals described in Section 1.1. As described in Section 1.2, current visualization systems approach the goals for flexibility by enumerating different data types and different types of displays. In Chapter 3 we develop an alternate approach to flexibility based on defining very general conditions on the mapping from data to displays, and we analyze the repertoire of functions that satisfy those conditions. We summarize the results of this analysis in terms of a set of principles for visualization. In Chapter 4 we continue the presentation of our visualization system architecture based on those principles. In Chapter 5 we discuss how the analysis of Chapter 3 might be extended to data and display models appropriate for general programming languages. Chapter 6 summarizes the conclusions of this thesis.

# Chapter 2

# System Design for Visualizing Scientific Computations

In Section 1.1 we defined five broad goals for scientific visualization. Specifically, we seek visualization techniques that

1. Can be applied to the data of a wide variety of scientific applications.

2. Can produce a wide variety of different visualizations of data appropriate for different needs.

3. Enable users to interactively alter the ways data are viewed.

4. Require minimal effort by scientists.

5. Can be integrated with a scientific programming environment.

In this chapter we develop a system architecture for visualizing scientific computations based on these goals. This architecture is implemented in a system called VisAD (Visualization for Algorithm Development).

## 2.1 A Scientific Computing Environment

The purpose of scientific visualization is to make invisible computations visible. Thus, for example, Figure 1.2 is a visualization of a simulation of the Earth's atmosphere. This image includes depictions of heat (the red and green vertical slice), air flow (the yellow ribbons), precipitated cloud ice (the blue-green iso-surface), and a chimney-shaped balloon (the white object) floating over a patch of

tropical ocean (the blue square). This image shows just one instant from the sequence of changing atmospheric states produced by the simulation. The total volume of data produced by this simulation is enormous, and would be impossible to understand without such visualizations.

In order to make such complex computations visible, our fifth goal was to develop visualization techniques that "*Can be integrated with a scientific programming environment*." Our design meets this goal by including a scientific programming language as part of the visualization system. This goal could be met in other ways, for example by providing a library of functions for displaying data that is callable from common scientific programming languages. However, the size and complexity of scientific computations and data motivated our third goal that visualization techniques "*Enable users to interactively alter the ways data are viewed*." In particular we noted in Section 1.1 that the user feedback cycle illustrated in Figure 2.1 may be applied interactively to running computations. This argues for a system architecture that can flexibly and intimately integrate the user interfaces for programming, computation and display. This can best be achieved by integrating a scientific programming language with a visualization system.

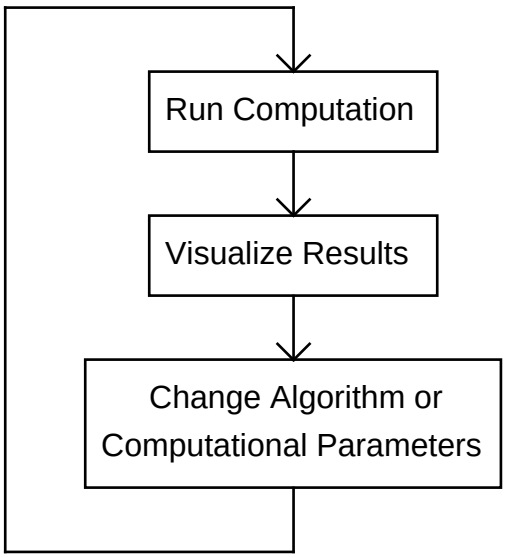

Figure 2.1 The place of visualization in the computational process (this is a copy of Figure 1.3).

Robert Aune's simulation of a two-dimensional shallow fluid (Haltiner and Williams, 1980) illustrates how the integration of visualization with a programming language enables the feedback loop in Figure 2.1 to be applied to running computations. The VisAD implementation of the shallow fluid model is described by the following pseudo-code:

```
loop over model time steps {
    /* get the user's interactive controls of the model */
    parameter1 = slider("name1", low1, high1, default1);
    . . .
    parameterN = slider("nameN", lowN, highN, defaultN);
    /* compute the next state of the model */
    new = shalstep(oldest, old, parameter1, ..., parameterN);
    oldest = old; /* save previous model state */
    old = new;
} /* end of loop for simulation time steps */
```

Figure 2.2 shows a screen snapshot of the VisAD system running this program. The system generates the icons seen in the lower-left corner of the screen based on the calls to the *slider* function. As the program runs, the user is free to set values on these icons, which are returned by the calls to the *slider* function. These values are passed to the Fortran function *shalstep*, which computes a new fluid state from the states for the previous two time steps. The window in the lower-right corner of the screen is a visualization of the current state of the simulated fluid. Together, *slider* icons and this visualization enable the user feedback loop illustrated in Figure 2.1 to be applied to the running shallow fluid simulation.

Figure 2.2. A snapshot of an executing shallow fluid simulation model. Part of a VisAD program is seen in the text window on the left, slider icons used to interact with the simulation are seen in the lower-left, and a visualization of the data object *new* is seen in the lower-right window. (color original)

Figure 2.2 also illustrates the integration of user interfaces for programming, computation and display. The white window on the left side of the screen contains the text of the fluid simulation program. The long dark horizontal bar highlights the program statement currently being executed, and the short dark horizontal bars highlight occurrences of the name of the data object being displayed (in this case, the name is *new*). The user selects data objects for display by picking their names in this text window (i.e., pointing and clicking at their names with the mouse). The user similarly sets program execution breakpoints by picking program statements in this window.

Our visualization system design provides an interactive interpreted language in order to let scientists perform visual experiments with their algorithms and computations. However, an interpreted language is relatively inefficient. Furthermore, scientists may already have large amounts of software written in Fortran and C. Thus the VisAD system supports dynamic linking between its interpreted language and these common compiled languages.

We considered a visual programming language for our system, similar to those used in data flow visualization systems. Such languages provide a graphical user interface for designing the data and control flow of programs. However, we chose a text based user interface for an interpreted language because it is more familiar to scientists and can express large and complex algorithms more compactly. Our choice is supported by the relative popularity of the IDL (Interactive Data Language) system among physical scientists, compared to the data flow visualization systems. In fact, if the source code of the IDL system was freely available we would have strongly considered using it as the scientific programming environment integrated for the VisAD system.

One powerful effect of integrating visualization with a scientific programming language is the ability to visually trace computations by watching displays of many different data objects. If an algorithm is not producing correct results, such integration allows users to step through their computations, visually comparing the inputs and outputs of short segments of code in order to find a bug. This capability requires that visualization be applied to any selected data object occurring in a program, and thus provides

additional motivation for our first goal that scientific visualization techniques "*Can be applied to the data of a wide variety of scientific applications*." Thus in the next section we study the nature of scientific data.

**2.2 Scientific Data**

Physical scientists formulate mathematical models of nature to simulate complex events and to analyze observations. Models of the Earth's atmosphere and oceans provide one good class of examples. *Temperatures*, *pressures*, *latitudes*, *altitudes* and *times* are expressed as numbers. The primitive elements of mathematical models are numerical variables used to represent such physical quantities. These primitive variables are then combined in various ways to build the complex objects of mathematical models. For example, the state of a infinitesimal parcel of air may be described by the vector:

*parcel* = {*temperature*, *pressure*, *water-concentration*,
*wind-velocity-x*, *wind-velocity-y*, *wind-velocity-z*}

The values of *temperature* and other primitive variables vary over space, and may be described by the functions:

*temperature* = *temperature-field*(*latitude*, *longitude*, *altitude*)
*pressure* = *pressure-field*(*latitude*, *longitude*, *altitude*)
*water-concentration* = *water-concentration-field*(*latitude*, *longitude*, *altitude*)
*wind-velocity-x* = *wind-velocity-x-field*(*latitude*, *longitude*, *altitude*)
*wind-velocity-y* = *wind-velocity-y-field*(*latitude*, *longitude*, *altitude*)
*wind-velocity-z* = *wind-velocity-z-field*(*latitude*, *longitude*, *altitude*)

The state of the atmosphere may be described by the vector of functions:

*state* = {*temperature-field*(*latitude*, *longitude*, *altitude*),
*pressure-field*(*latitude*, *longitude*, *altitude*),
*water-concentration-field*(*latitude*, *longitude*, *altitude*),
*wind-velocity-x-field*(*latitude*, *longitude*, *altitude*),
*wind-velocity-y-field*(*latitude*, *longitude*, *altitude*),
*wind-velocity-z-field*(*latitude*, *longitude*, *altitude*)}

Finally, the state of the atmosphere varies over *time*, and a history of the atmosphere may be described by the function:

*state* = *state-history*(*time*)

We refer to these mathematical variables, vectors and functions as *mathematical objects*. The dynamics of the Earth's atmosphere may be modeled by sets of (partial differential) equations involving these mathematical objects, and, in general, physical scientists' mathematical models are expressed in terms of such mathematical objects.

Recording and analyzing actual observations and predicting actual events require implementations of mathematical models by hand or automated computations. Whereas mathematical models include infinite precision real numbers and functions with infinite domains, computer memories are finite. Thus computer implementations of mathematical models are approximations. For example, real numbers are usually approximated by floating point numbers, and functions are usually approximated by finite arrays. That is, values in the infinite set of real numbers are commonly approximated by values taken from a finite set of roughly 2 ^ 32 values between -10 ^ 38 and +10 ^ 38 (the set of 32-bit floating point values) and the infinite sets of values of functions are commonly approximated by finite subsets of those values (for

example, atmospheric models usually define discrete values for *temperature*, *pressure* and other state variables at finite grids of locations within the atmosphere).

Thus we interpret data objects as representing mathematical objects. There are a variety of mathematical types (for example, primitive variables, vectors, functions, vectors of functions, and so on) so we define a variety of types of data objects appropriate for representing mathematical objects. Specifically, we define primitive data types for representing primitive mathematical variables - these could be integer or floating point types. We define vector types for representing mathematical vectors - these are called *records*, *structures* or *tuples* in different programming languages. We define array types for representing mathematical functions - these are finite sets of samples of function values. We use these as the data types of the scientific programming language that is integrated with our visualization system.

As an example, we define the following data types for representing the mathematical types defined earlier. These types could be used for an implementation of an atmospheric model in the VisAD programming language.

type *temperature* = real;

type *pressure* = real;

type *water-concentration* = real;

type *wind-velocity-x* = real;

type *wind-velocity-y* = real;

type *wind-velocity-z* = real;

type *parcel* = structure{*temperature*; *pressure*; *water-concentration*;
*wind-velocity-x*; *wind-velocity-y*; *wind-velocity-z*;}

type *latitude* = real;

type *longitude* = real;

type *altitude* = real;

type *temperature-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *temperature*;

type *pressure-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *pressure*;

type *water-concentration-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *water-concentration*;

type *wind-velocity-x-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *wind-velocity-x*;

type *wind-velocity-y-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *wind-velocity-y*;

type *wind-velocity-z-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *wind-velocity-z*;

type *state* =

structure {*temperature-field*; *pressure-field*; *water-concentration-field*;

*wind-velocity-x-field*; *wind-velocity-y-field*; *wind-velocity-z-field*;}

type *time* = real;

type *state-history* = array [*time*] of *state*;

These examples illustrate the ways that data types are defined in the VisAD programming language.

As in Section 1.2.3, we let *U* denote the set of data objects used to represent mathematical objects in *U'*. Scientific displays can be viewed as a special kind of data object so, as in Section 1.2.3, we let *V* denote a set of display objects. Next we consider the nature of scientific displays.

**2.3 Scientific Displays**

The same data may be visualized in many different ways, as illustrated in Figure 1.1. Thus our second goal was to develop visualization techniques that "*Can produce a wide variety of different visualizations of data appropriate for different needs*." In order to satisfy this goal, our visualization system should include a flexible and general display model.

Bertin's display model was limited to static two-dimensional images. While his model was adequate as a description of the instantaneous contents of a workstation screen, it fails to express the dynamic, three-dimensional and interactive character of scientific displays. Thus we distinguish between a set *V'* of static two-dimensional images (i.e., physical displays) and a set *V* of logical displays. For a given

physical display device, *V'* is a finite and fixed set of static two-dimensional images (for example, it may be the set of 1024 by 1024 arrays of pixels with 8 bits of intensity for each of red, green and blue). Because *V'* is finite, a visualization mapping $D : U \rightarrow V'$ cannot be injective (i.e., one to one). This would be a severe constraint on any effort to analyze mappings from data to displays. On the other hand, we can define a infinite set *V* of logical displays that

1. Are three-dimensional.

2. Are animated.

3. Have infinite extents in space and time.

4. Have varying resolution in space and time.

5. Are generated by a variety of rendering techniques.

The meaning of logical displays in *V* is defined by a function $RENDER : V \rightarrow V'$. The *RENDER* function projects three-dimensional displays onto a two-dimensional screen, removes hidden objects during this projection process, clips displays to finite screen boundaries, simulates scene lighting, simulates transparency and reflection, animates sequences of static images, and so on. The logical display model may include generic scalar and vector fields, in which case the *RENDER* function may implement the calculation of iso-surfaces and plane slices to represent scalar fields, and of arrows and streamlines to represent vector fields. We note that there are many possible functions $RENDER : V \rightarrow V'$, depending on parameters of the projection from three to two dimensions, on parameters of simulated lighting, on the place in an animation sequence, and so on. By giving users control over these parameters, and thus control

over the choice of the function *RENDER* : $V \rightarrow V'$, we define the interactive nature of logical displays in *V*. For example, control over the projection from three to two dimensions lets users interactively rotate, pan and zoom logical displays.

The *RENDER* function implements the traditional operations of computer graphics which have been extensively studied (Foley and Van Dam, 1982; Wyvill, McPheeters and Wyvill, 1986; Lorensen and Cline, 1987).

**2.4 Mapping Data to Displays**

We have described a scientific data model *U* containing data objects of various types, and a display model *V* containing interactive, animated, three-dimensional displays. Visualization is a computational process that transforms data into displays and can be described as a function of the form $D : U \rightarrow V$. The *visualization repertoire* of our system can be described as a set of functions of this form. In order to satisfy the goal of developing visualization techniques that "*Can produce a wide variety of different visualizations of data appropriate for different needs*" we seek to define a broad visualization repertoire. As described in Section 1.2, current systems define visualization repertoires by enumerating such sets of functions. However, with an enumerated repertoire there is no way to be sure that it includes all useful ways of displaying data. An enumerated repertoire is justified only by the tastes and experience of those who decide what functions to include in the enumeration.

In contrast, we seek to define a visualization repertoire as the set of all functions satisfying Mackinlay's expressiveness conditions (Mackinlay, 1986). These conditions say that displays express all facts about data objects, and only those facts. In the next chapter we show how these conditions can be rigorously interpreted in terms of lattice structures defined on data and display models. We have noted that scientific data objects are approximate representations of mathematical objects. We define a lattice structure on our data model *U* based on a way of comparing how data objects approximate mathematical objects, and define a similar lattice structure on our display model *V*. We then define our system's

visualization repertoire as the set of visualization functions $D : U \rightarrow V$ that satisfy the expressiveness conditions, as interpreted in the lattice structure.

This approach to defining a visualization repertoire has a number of advantages, including:

1. The repertoire is complete, in the sense that it includes all visualization functions satisfying the expressiveness conditions.

2. A single function $D : U \rightarrow V$ can be applied to display data objects of any type in the unified data model $U$, simplifying the user interface for controlling displays. That is, one set of display controls can be applied to display any data object defined in a program. Because display controls are independent of data type, they are naturally separate from a user's scientific algorithms. This is a clear distinction from previous visualization systems that require calls to visualization functions to be embedded into scientific programs. In Chapter 3 we show that selection of a function satisfying the expressiveness conditions can be controlled by a conceptually simple user interface.

3. Lattice structures can be defined for a wide variety of data and display models, so our approach can easily be extended to other scientific data and display models. In Chapter 5 we outline how the approach may even be extended to a data model appropriate for a general-purpose programming language.

4. A lattice-structured data model provides a natural way to integrate various forms of scientific metadata into the computational and display semantics of scientific data. This reduces the user's need to explicitly manage the relation between data and associated metadata.

# Chapter 3

# An Analysis of Mappings from Data to Displays

Current scientific visualization systems enumerate different ways of displaying data, or require users to write programs (possibly as data flow diagrams or in object-oriented programming languages) to control how data are displayed. These approaches either lack flexibility or require significant effort from users. In contrast, we take a more systematic approach, analyzing the ways that data can be displayed from basic principles.

In this chapter we describe our approach to scientific visualization, based not only on an abstract view of data and displays, but also an abstract view of the visualization mapping between them. First, we recognize that visualization is a computational process that defines a mapping from a large set of data objects to a large set of displays. Thus, rather than analyzing visualization in terms of the way an individual data object is displayed, we analyze visualization in terms of its effect on sets of data objects. (In fact, it is arguable that data objects only have meaning in relation to other data objects, just as the significance of the number pi can be explained only in relation to other mathematical objects). Thus we let the symbol $U$ represent a set of data objects, let the symbol $V$ represent a set of displays, and let $D : U \rightarrow V$ represent the mapping from data to displays. We define a visualization repertoire as the set of all such visualization mappings that satisfy certain analytic conditions.

The simplest example of an analytic condition on $D$ expresses the uniqueness requirement that different data objects have different displays, so that users can distinguish different data objects from their displays. This is just the condition that $D$ be injective (one to one). It can be expressed as follows:

(3.1) $\forall u, u' \in U.\ u = u' \Leftrightarrow D(u) = D(u')$

Eq. (3.1) is a very weak condition on *D*. For example, if *U* is a set of two-dimensional images and if *V* = *U* (that is, the display model *V* is also a set of two-dimensional images), then any permutation of images satisfies Eq. (3.1). However, it is easy to construct a permutation *D* of images such that the display *D*(*u*) generally does a poor job of communicating information about the data object *u* to users. Thus we seek stronger conditions on *D*.

In general, any condition on *D* must be defined in terms of mathematical structures on *U* and *V*. For example, Eq. (3.1) expresses a condition in terms of the mathematical structure of equality on *U* and *V*. The advantage of Eq. (3.1) is that it can be applied very broadly to visualization because all data and display models include an equality relation. Therefore we also seek to define stronger conditions on *D* that express fundamental properties of scientific data objects and displays.

### 3.1 An Analytic Approach Based on Lattices

In this thesis we focus on the observation that, for most scientific computations, computer data objects and displays are finite approximations to mathematical models of nature. That is, real numbers have infinite precision and functions of real variables have infinite domains, whereas the computer data objects that represent these mathematical objects are finite and therefore approximate. Because scientific data objects and displays are approximations, we can define an order relation between them based on the precision of approximation (for example, a high resolution image is more precise than a low resolution image as an approximation to a radiance field). This order relation allows us to define lattice structures on data and display models, and to define analytic conditions on visualization mappings based on the lattice structures.

#### 3.1.1 Basic Definitions for Ordered Sets

Since our analytic approach to visualization draws on the theory of ordered sets, we first review some basic definitions from this theory (Davey and Priestly, 1990; Gierz, et al., 1980; Gunter and Scott,

1990; Schmidt, 1986; Scott, 1971; Scott, 1976; Scott, 1982). Appendix A contains a more complete list of definitions.

**Def.** A *partially ordered set* (*poset*) is a set $D$ with a binary relation $\leq$ on $D$ such that, $\forall x, y, z \in D$

(a) $x \leq x$ "reflexive"

(b) $x \leq y \,\&\, y \leq x \Rightarrow x = y$ "anti-symmetric"

(c) $x \leq y \,\&\, y \leq z \Rightarrow x \leq z$ "transitive"

**Def.** An *upper bound* for a set $M \subseteq D$ is an element $x \in D$ such that $\forall y \in M.\ y \leq x$.

**Def.** The *least upper bound* of a set $M \subseteq D$, if it exists, is an upper bound $x$ for $M$ such that if $y$ is another upper bound for $M$, then $x \leq y$. The least upper bound of $M$ is denoted *sup* $M$ or $\bigvee M$. *sup*$\{x,y\}$ is written $x \vee y$.

**Def.** A *lower bound* for a set $M \subseteq D$ is an element $x \in D$ such that $\forall y \in M.\ x \leq y$.

**Def.** The *greatest lower bound* of a set $M \subseteq D$, if it exists, is a lower bound $x$ for $M$ such that if $y$ is another lower bound for $M$, then $y \leq x$. The greatest lower bound of $M$ is denoted *inf* $M$ or $\bigwedge M$. *inf*$\{x,y\}$ is written $x \wedge y$.

**Def.** A subset $M \subseteq D$ is a *down set* if $\forall x \in M. \forall y \in D.\ y \leq x \Rightarrow y \in M$. Given $M \subseteq D$, define $\downarrow M = \{y \in D \mid \exists x \in M.\ y \leq x\}$, and given $x \in D$, define $\downarrow x = \{y \in D \mid y \leq x\}$.

**Def.** A subset $M \subseteq D$ is an *up set* if $\forall x \in M.\ \forall y \in D.\ x \leq y \Rightarrow y \in M$. Given $M \subseteq D$, define $\uparrow M = \{y \in D \mid \exists x \in M.\ x \leq y\}$, and given $x \in D$, define $\uparrow x = \{y \in D \mid x \leq y\}$.

**Def.** A subset $M \subseteq D$ is *directed* if, for every finite subset $A \subseteq M$, there is an $x \in M$ such that $\forall y \in A.\ y \leq x$.

**Def.** If $D$ and $E$ are *posets*, we use the notation $(D \to E)$ to denote the set of all functions from $D$ to $E$.

**Def.** If $D$ and $E$ are *posets*, a function $f{:}D \to E$ is *monotone* if $\forall x, y \in D.\ x \leq y \Rightarrow f(x) \leq f(y)$. We use the notation $MON(D \to E)$ to denote the set of all monotone functions from $D$ to $E$.

**Def.** If $D$ and $E$ are *posets*, a function $f{:}D \to E$ is an *order embedding* if $\forall x, y \in D.\ x \leq y \Leftrightarrow f(x) \leq f(y)$.

**Def.** Given *posets* $D$ and $E$, a function $f{:}D \to E$, and a set $M \subseteq D$, we use the notation $f(M)$ to denote $\{f(d) \mid d \in M\}$.

**Def.** A *poset* $D$ is a *lattice* if for all $x, y \in D$, $x \vee y$ and $x \wedge y$ exist in $D$.

**Def.** A *poset* $D$ is a *complete lattice* if for all $M \subseteq D$, $\bigvee M$ and $\bigwedge M$ exist in $D$.

**Def.** If $D$ and $E$ are lattices, a function $f{:}D \rightarrow E$ is a *lattice homomorphism* if for all $x, y \in D$, $f(x \wedge y) = f(x) \wedge f(y)$ and $f(x \vee y) = f(x) \vee f(y)$. If $f{:}D \rightarrow E$ is also a bijection then it is a *lattice isomorphism*.

### 3.1.2 Scientific Data Objects as Approximations of Mathematical Objects

In Section 2.2 we described the nature of scientific data as representing mathematical objects. We noted that data objects are usually approximations to mathematical objects, as for example floating point numbers approximate real numbers and arrays are finite samplings of functions of a real variable.

The importance of the approximate nature of scientific data is reflected in the common use of semantic metadata to document the how scientific data approximate mathematical variables and functions. For example, in Section 2.2 we defined a data type:

type *temperature-field* =

array [*latitude*] of array [*longitude*] of array [*altitude*] of *temperature*;

Data objects of type *temperature-field* are approximate representations of the mathematical function:

*temperature* = *temperature-field*(*latitude*, *longitude*, *altitude*)

One important form of scientific metadata describes the locations of samples of *temperature-field* arrays. Furthermore, temperature values in the array are represented by finite numbers of bits, and another important form of metadata describes the correspondence between finite bit patterns and real numbers. Such metadata may be implicit in the specification of a floating point number standard, but may also be explicit, as in the case of coded 8-bit or 10-bit satellite radiances. Metadata may describe how data values are spatial or temporal averages of physical variables; this metadata quantifies how data values approximate mathematical values. Metadata may explicitly document numerical precision by providing

error bounds for values that approximate real numbers. Metadata may define *missing* data codes used to indicate failures of observing instruments or numerical exceptions; we view such *missing* data codes as documenting values that have the least possible precision.

Other metadata provide indirect information about how precisely data objects approximate mathematical objects. Values produced by simulations may include metadata about the name and version number of the model that produced them, about the data used to initialize the model, about parameter settings of the model, and so on. Values produced by observations may include metadata about which sensors produced them, and may also include, for example, observations of the instruments themselves for calibration, sensor temperatures, angles to the sun or other navigation landmarks, and so on. These detailed metadata are often the basis of complex computations for estimating sampling and accuracy characteristics of values.

The approximate nature of scientific data is a fundamental property of that data that can serve as the basis for a mathematical order structure on a scientific data model. As explained in the next section, data objects can be ordered based on how precisely they approximate mathematical objects. This order relation provides us with a mathematical structure on data and display models that can be used as the basis for defining analytic conditions on visualization mappings.

### 3.1.3 A Mathematical Structure Based on the Precision of Scientific Data

We assume a set $U'$ of mathematical objects and a set $U$ of data objects. There are only a countable number of data objects (objects that can be stored inside a computer are limited to finite strings over finite alphabets) but an uncountable number of mathematical objects. Thus each data object generally represents a large set of mathematical objects. Given a data object $u \in U$, let $math(u) \subseteq U'$ be the set of mathematical objects represented by $u$. Given two data objects $u$ and $u'$, if $math(u') \subseteq math(u)$ then $u'$ represents a more restricted set of mathematical objects than $u$ does and we can say that $u'$ is more precise than $u$. Thus we define an order relation on $U$ by:

(3.2) $u \leq u' \Leftrightarrow math(u') \subseteq math(u)$

For example, a *missing* value (which we indicate by the symbol $\perp$) can represent (i.e., is consistent with) all mathematical values, so $\perp \leq x$ where $x$ is any data value.

Similar order relations have been defined for reasoning about partial information in data base management systems (Read, Fussell and Silberschatz, 1993) and in the study of programming language semantics (Scott, 1971). There is no algorithmic way to separate non-terminating programs from terminating programs, so the set of meanings of programs must include an *undefined* value for non-terminating programs. This value is less precise than any of the values that a program would produce if it did terminate so it is natural to define an order relation between program meanings where *undefined* $\leq x$ for all program values $x$. In order to define a correspondence between the ways that programs are constructed, and the sets of meanings of programs, Scott elaborated this order relation into an elegant lattice theory for the meanings of programs (Scott, 1982). He equated

"$x \leq y$" with "$x$ approximates $y$."

Thus Scott's order relation is similar to the order relation defined by Eq. (3.2), and the *undefined* value in programming language semantics is analogous with the *missing* value used in scientific computations. (We note that the source of *undefined* values is non-terminating computations whereas the sources of *missing* values are sensor failures and numerical exceptions). There are many other examples of how the order relation defined in Eq. (3.2) may be applied. Metadata about accuracy often take the form of error bars, which are intervals around values. Real intervals have been studied as a computational data model for real numbers (Moore, 1966), and have been applied to computer graphics (Duff, 1992; Snyder, 1992). An interval represents any real number it contains, so Eq. (3.2) indicates that smaller intervals are "greater than" containing intervals. We can combine the *missing* value and real intervals in a simple data model for approximations of real numbers. The order relation on this data model is illustrated in Figure

3.1. Note that the set of real intervals is not countable, but an implementation of the real interval data model could be restricted to the set of rational intervals. From now on we will not require that *U* be countable, but will recognize that an actual implementation of *U* can only include a countable number of data objects.

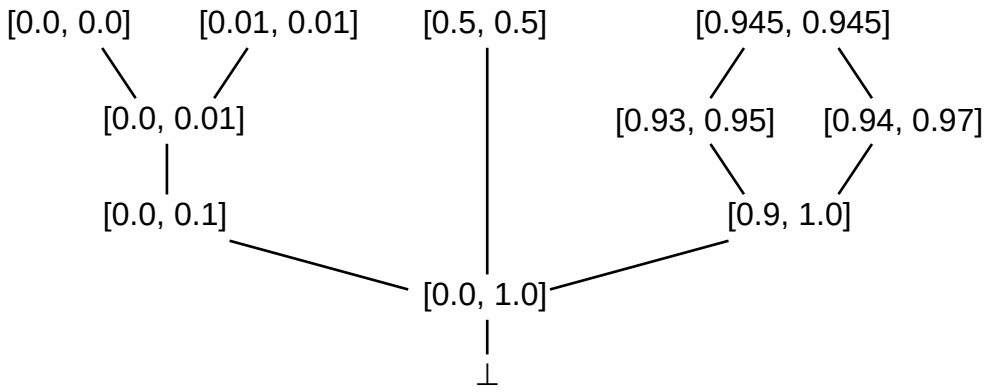


Figure 3.1. Order relation of a continuous scalar. Closed real intervals are used as approximate representations of real numbers, ordered by the inverse of containment (that is, containing intervals are "less than" contained intervals). We also include a least element ⊥ that corresponds to a missing data indicator. This figure shows a few intervals, plus the order relations among those intervals. The intervals in the top row are all maximal, since they contain no smaller interval.

We can extend the data model in Figure 3.1 to real functions by defining array data objects that are sets of pairs of real intervals. The first interval in a pair represents a domain value of the function, and the second interval represents the corresponding range value. The two intervals define a rectangle that contains at least one sample from the graph of the represented function. For example, the set of pairs

(3.3) {([1.1, 1.6], [3.1, 3.4]), ([3.6, 4.1], [5.0, 5.2]), ([6.1, 6.4], [6.2, 6.5])}

contains three samples of a function. The domain value of a sample lies in the first interval of a pair and its range value lies in the second interval of a pair, as illustrated in Figure 3.2.

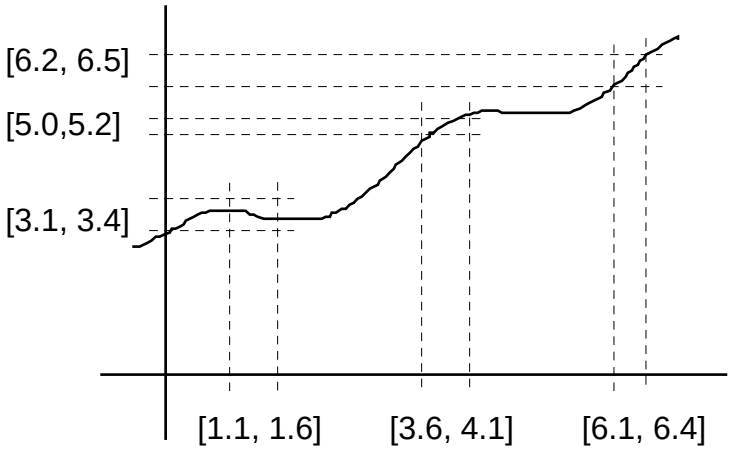


Figure 3.2. Approximating real functions by arrays.

An array represents any function whose graph contains a point in each of the rectangles defined by its pairs. Adding more samples to an array restricts the set of functions that the array can represent. Similarly, replacing pairs of intervals by pairs of more precise intervals restricts the set of functions that the array can represent. Thus we can define an order relation between arrays, as illustrated in Figure 3.3. Note that the empty set is the least value of this data model since it can represent any real function.

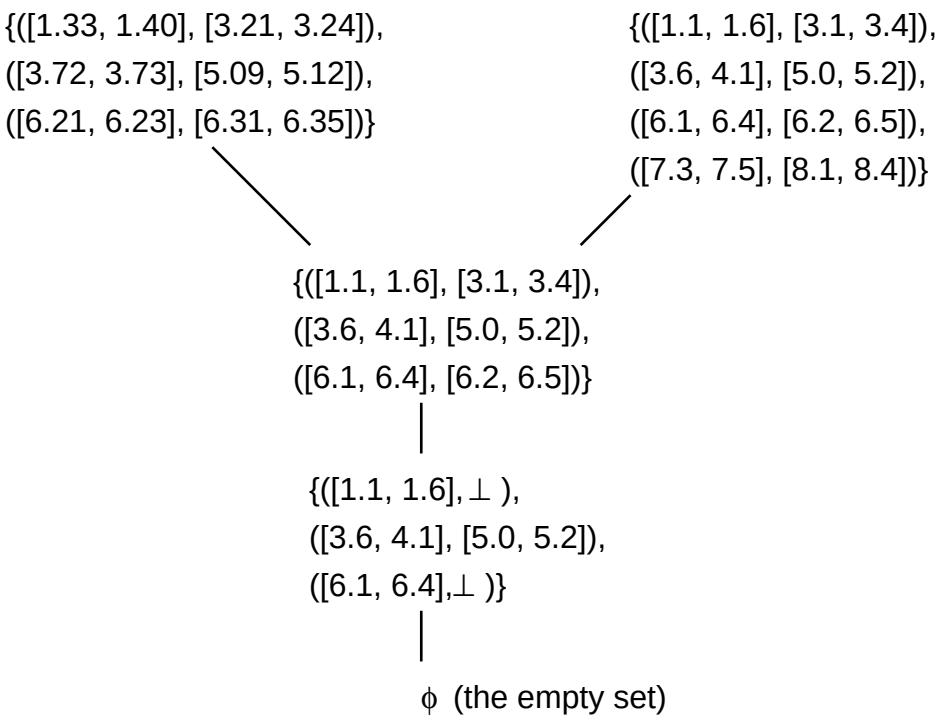


Figure 3.3. Order relation of arrays.

The sequence of satellite images in Figures 3.4 through 3.7 provides a practical illustration of an order relation based on precision. Each of these images contains a finite number of pixels that are samples of a continuous Earth radiance field. The higher resolution images are more precise approximations to the radiance field, and the sequence of images form an ascending chain in the order relation.

Figure 3.4. Least precise image in sequence of four.  (color original)

Figure 3.5. Second image in sequence of four, ordered by precision. (color original)

Figure 3.6. Third image in sequence of four, ordered by precision. (color original)

Figure 3.7. Most precise image in sequence of four.  (color original)

These examples of data models for approximating two simple types of mathematical objects, real numbers and real functions, show how Eq. (3.2) can be used to define order relations. In these examples we defined different sets of data objects to represent different mathematical types. However, a scientific application may include many data types, and it is impractical to provide a separate data model $U$ and a separate analysis of visualization functions $D : U \to V$ for each different data type. Thus it is desirable to define data models that include many different data types.

In the study of programming language semantics, objects of many different types have been embedded in lattices called *universal domains* (Scott, 1976). In Section 3.2 we will show how scientific data objects of many different types can be embedded in a single lattice. Thus we assume that our data model $U$ is a lattice. We further assume that $U$ is a complete lattice. Any ordered set can be embedded in a complete lattice by the Dedekind-MacNeille completion (Davey and Priestly, 1990), so this is not a very strong assumption. Scott showed how to define a topology on ordered sets (Gunter and Scott, 1990) and in this topology least upper bounds play a role analogous to limits. Thus we can think of the assumption that $U$ is complete as meaning that it contains the mathematical objects that are the limits of sets of approximating finite data objects. Complete lattices are a convenient mathematical context for studying visualization functions, as long as we remember that actual implementations of data models are restricted to countable subsets of $U$.

The notion of precision of approximation also applies to displays. Displays have finite resolution in space, color and time (that is, animation). Two-dimensional images and three-dimensional volume renderings are composed of finite numbers of pixels and voxels, each implemented with a finite number of bits, and changing in discrete steps over time. Computer displays are finite approximations to idealized mathematical displays (that is, displays defined in terms of real-valued functions) and it is possible to define an order relation between displays based on the precision of these approximations. Thus we assume that our display model $V$ is also a complete lattice.

### 3.1.4 Data Display as a Mapping Between Lattices

Data objects provide information about mathematical objects, and Eq. (3.2) says that the order relations on $U$ and $V$ provide measures of the information in data objects and displays (that is, how precisely they specify mathematical objects). The purpose of visualization is to communicate information about data objects, and we will express this purpose as conditions on $D : U \rightarrow V$ defined in terms of the order relations on $U$ and $V$. In order to define conditions on $D$ we draw on the work of Mackinlay (Mackinlay, 1986). He studied the problem of automatically generating displays of relational information and defined *expressiveness conditions* on the mapping from relational data to displays. His conditions specify that a display expresses a set of facts (that is, an instance of a set of relations) if the display encodes all the facts in the set, and encodes only those facts.

In order to interpret the expressiveness conditions we define a fact about data objects as a logical predicate applied to $U$ (that is, a function of the form $P : U \rightarrow \{false, true\}$). However, since data objects are approximations to mathematical objects, we limit facts about data objects to approximations of facts about mathematical objects. In particular, we would like to avoid predicates that define inconsistent information about mathematical objects. For example, if $u_1 \leq u_2$ then $u_1$ and $u_2$ are approximations to the same mathematical object (or objects), so we will disallow any predicates that define $P(u_1) = true$ and $P(u_2) = false$. We can do this by restricting our interpretations of facts about data objects to monotone predicates of the form $P: U \rightarrow \{undefined, false, true\}$, where $undefined < false$ and $undefined < true$. Furthermore, a monotone predicate of the form $P: U \rightarrow \{undefined, false, true\}$ can be expressed in terms of two monotone predicates of the form $P: U \rightarrow \{undefined, true\}$, so we will limit facts about data objects to monotone predicates of the form $P: U \rightarrow \{undefined, true\}$.

The first part of the expressiveness conditions says that every fact about data objects is encoded by a fact about their displays. We interpret this as follows:

**Condition 1.** For every monotone predicate $P$: $U \rightarrow$ {*undefined*, *true*}, there is a monotone predicate $Q$: $V \rightarrow$ {*undefined*, *true*} such that $P(u) = Q(D(u))$ for each $u \in U$.

This requires that $D$ be injective [if $u_1 \neq u_2$ then there are $P$ such that $P(u_1) \neq P(u_2)$, but if $D(u_1) = D(u_2)$ then $Q(D(u_1)) = Q(D(u_2))$ for all $Q$, so we must have $D(u_1) \neq D(u_2)$].

The second part of the expressiveness conditions says that every fact about displays encodes a fact about data objects. We interpret this as follows:

**Condition 2.** For every monotone predicate $Q$: $V \rightarrow$ {*undefined*, *true*}, there is a monotone predicate $P$: $U \rightarrow$ {*undefined*, *true*} such that $Q(v) = P(D^{-1}(v))$ for each $v \in V$.

We show in Appendix B that Condition 2 implies that $D$ is a monotone bijection (that is, one-to-one and onto) from $U$ onto $V$. Thus Condition 2 is too strong since it requires that every display in $V$ is the display of some data object in $U$, under $D$. Since $U$ is a complete lattice it contains a maximal data object $X$ (the least upper bound of all members of $U$). For all data objects $u \in U$, $u \leq X$. Since $D$ is monotone this implies

$D(u) \leq D(X)$. We use the notation $\downarrow D(X)$ for the set of all displays less than $D(X)$. $\downarrow D(X)$ is itself a complete lattice and for all data objects $u \in U$, $D(u) \in \downarrow D(X)$. Hence we can replace $V$ by $\downarrow D(X)$ in Condition 2 in order to not require that every $v \in V$ is the display of some data object. We modify Condition 2 as follows:

**Condition 2'.** For every monotone predicate $Q$: $\downarrow D(X) \rightarrow$ {*undefined*, *true*}, there is a monotone predicate $P$: $U \rightarrow$ {*undefined*, *true*} such that $Q(v) = P(D^{-1}(v))$ for each $v \in \downarrow D(X)$.

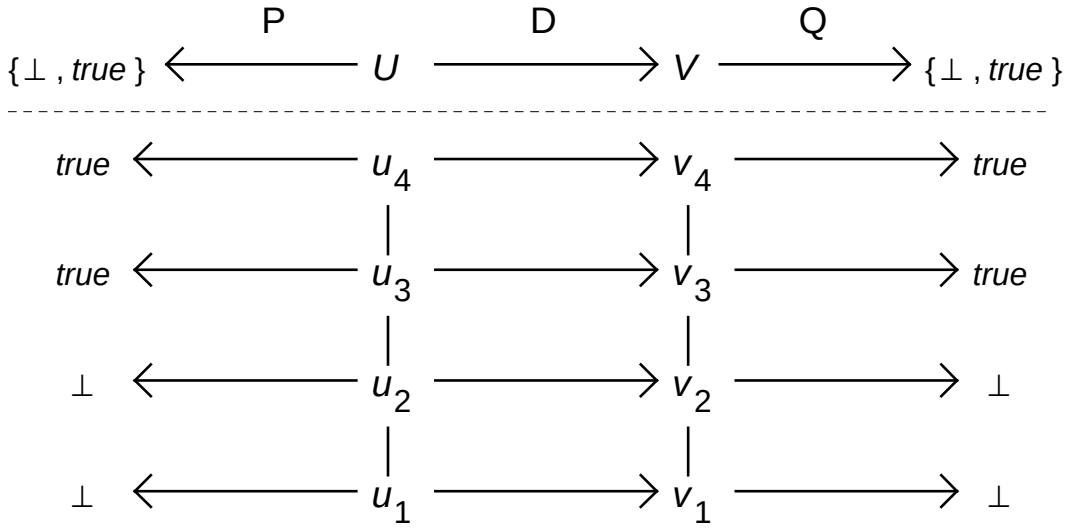


Figure 3.8. The expressiveness conditions specify that $D : U \to V$ defines a correspondence between monotone predicates on $U$ and $V$.

These two conditions quantify the relation between the information in data objects and the information in their displays. Figure 3.8 shows $D$ mapping the chain $u_1 < u_2 < u_3 < u_4$ in $U$ to the chain $v_1 < v_2 < v_3 < v_4$ in $V$, and shows the values of the monotone predicates $P$ and $Q$ on these chains. The expressiveness conditions define a correspondence between such predicates. We now use them to define a class of functions.

**Definition.** A function $D$: $U \to V$ is a *display function* if it satisfies Conditions 1 and 2'.

In Appendix B we prove the following result about the class of display functions:

**Prop. B.3.** A function $D$: $U \to V$ is a display function if and only if it is a lattice isomorphism from $U$ onto $\downarrow D(X)$.

This result may be applied to any complete lattice model of data and displays. In the rest of this chapter we will explore its consequences in a more specific setting.

### 3.2 A Scientific Data Model

The scientific data model developed in Section 2.2 defined a set of data types for representing mathematical types. We define *scalar* types for representing variables, *tuple* types for representing vectors, and *array* types for representing functions. Based on the ideas developed in Section 3.1.3, metadata that describe how precisely data objects approximate the mathematical objects that they represent are integrated into this data model.

The data model defines two kinds of primitive values, one appropriate for representing real numbers and the other appropriate for representing integers or text strings. We call these two kinds of primitive values *continuous* scalars and *discrete* scalars. A continuous scalar takes the set of closed real intervals as values, ordered by the inverse of containment. Figure 3.1 illustrated the order relations between values of a continuous scalar. A discrete scalar takes any countable (possible finite) set as values, without any order relation between them (since no integer is more precise than any other). Figure 3.9 illustrates the order relations between values of a discrete scalar. The value sets of continuous and discrete scalars also always include a minimal value ⊥ corresponding to a *missing* data indicator.

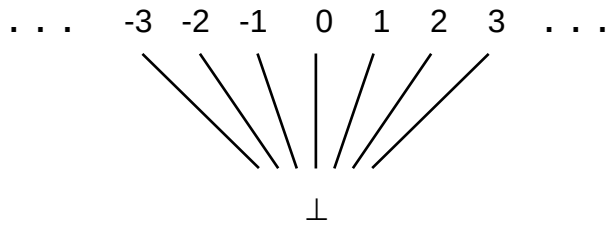


Figure 3.9. Order relation of a discrete scalar.

The data model does not specify a particular set of scalars. Rather the data model can be adapted to a particular scientific application by defining a finite set *S* of scalar types to represent the mathematical variables of the application (for example, *time*, *latitude*, *temperature*, *pressure*). These scalar types are aggregated into a set *T* of complex data types according to three rules:

1. Any continuous or discrete scalar in *S* is a data type in *T*.

2. If $t_1$, ..., $t_n$ are types in *T* defined from disjoint sets of scalars, then *struct*{$t_1$;...;$t_n$} is a *tuple* type in *T* with *element* types $t_i$. Data objects of tuple types (that is, data types constructed as tuples) contain one data object of each of their element types.

3. If *w* is a scalar type in *S* and *r* is a type in *T* such that *w* does not occur in the definition of *r*, then (*array* [*w*] *of r*) is an *array* type with *domain* type *w* and *range* type *r*. Data objects of array types (that is, data types constructed as arrays) are finite samplings of functions from the primitive variable represented by their domain type *w* to the set of values represented by their range type *r*. That is, a data object of an array type is a set of data objects of its range type, indexed by values of its domain type.

Each data type in *T* defines a set of data objects. Continuous and discrete scalars define sets of values as described previously. The set of objects of a tuple type is the cross product of the sets of objects of its element types. A tuple of data objects represents a tuple of mathematical objects, and the precision of the approximation depends on the precision of each element of the tuple. One tuple is more precise than another if each element is more precise. That is, $(x_1, ..., x_n) \leq (y_1, ..., y_n)$ if $x_i \leq y_i$ for each *i*. Figure 3.10 illustrates the order relations between tuples.

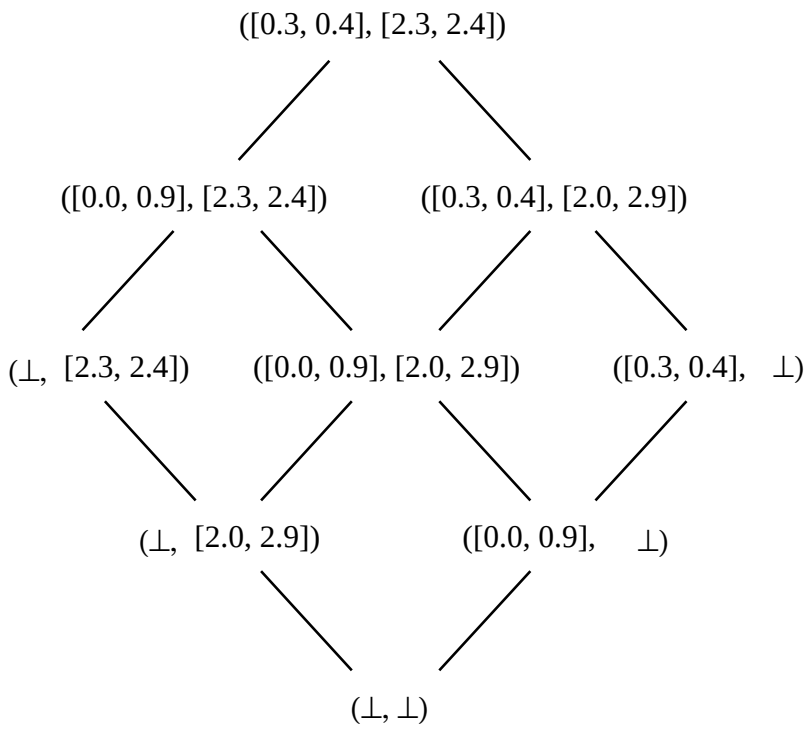


Figure 3.10. Order relation of tuples. Tuples are members of cross products. This figure shows a few elements in a cross product of two sets of continuous scalar values, plus the order relations among those elements. In a cross product, the least element is the tuple of least elements of the factor sets.

The set of objects of an array type is similar to a function space. However, an array's domain type generally defines an infinite set of values, whereas arrays are limited to finite subsets of domain values. For each finite subset of domain values, define the space of functions from this finite set to the set of objects of the array's range type. Then the set of objects of an array type is the union of such function spaces taken over all finite subsets of the domain's value set. We will make this definition rigorous in Section 3.2.3. The order relation between array objects was illustrated in Figure 3.3 and is precisely defined in Section 3.2.3.

While the development of this data model is complex, it offers several advantages over more commonly used data models. First, a wide variety of scientific data can be expressed in this data model by building hierarchies of tuples and arrays. Thus a system based on this data model can be applied to a wide variety of scientific applications and can be used to combine data from different sources. This is a significant advantage over most existing scientific visualization systems.

Second, this data model integrates several forms of scientific metadata, including:

1. Each scalar type is identified by the name of the primitive mathematical variable that it represents.

2. An array data object is a finite sampling of a mathematical function, and contains a set of objects of the array's range type, indexed by values of the array's domain scalar type. These index values specify how the array samples the function being represented.

3. The interval values of continuous scalars are approximations to real numbers in a mathematical model, and the sizes of these intervals provide accuracy metadata about the approximations.

4. Any scalar object may take the value $\perp$, corresponding to a *missing* data indicator.

Most previous systems require users to store such metadata in separate data objects and to manage the relation between data and metadata explicitly in their programs. A system based on this data model can integrate metadata into the computation and display semantics of data, and thus reduce the burden on users.

In the next three sections we show how to define a lattice structure for this data model. This lattice structure can be used to analyze visualization mappings from this data model to a lattice-structured display model and thus define a repertoire of visualization functions for a system based on this data model.

### 3.2.1 Interpreting the Data Model as a Lattice

We treat the visualization process as a function from a set of data objects to a set of display objects. Our data model defines a different set of data objects for each different data type, suggesting that a different visualization function must be defined for each different data type. However, we can define a lattice of data objects and natural embeddings of data objects of all data types into this lattice. This lattice provides us with a unified data model $U$ for data objects of all data types in $T$. Thus a visualization function $D : U \rightarrow V$ applies to all data types in $T$ and our analysis of the properties of these visualization functions will be independent of particular data types.

In Section 1.2 we saw that many current visualization techniques achieve generality by enumerating sets of data types and display techniques. The lattice $U$ provides an alternative to this approach by defining a unified data model and enabling a unified analysis of visualization functions for different data types.

Define a tuple space $X$ as the cross product of the sets of values of the scalar types in $S$, and define a member of the data lattice as a subset of the tuple space $X$. In Section 3.2.2 we show how to define an order relation on this lattice, and in Section 3.2.3 we show how the data objects of our scientific data model are embedded in this lattice.

To get an intuition of how the embedding works, consider a data lattice $U$ defined from the three scalars *time*, *temperature* and *pressure*. Objects in the lattice $U$ are sets of tuples of the form (*time*, *temperature*, *pressure*). Consider the tuple data type *struct*{*temperature*; *pressure*}. Data objects of this type are tuples of the form (*temperature*, *pressure*), and we can embed them in the lattice $U$ as illustrated in Figure 3.11.

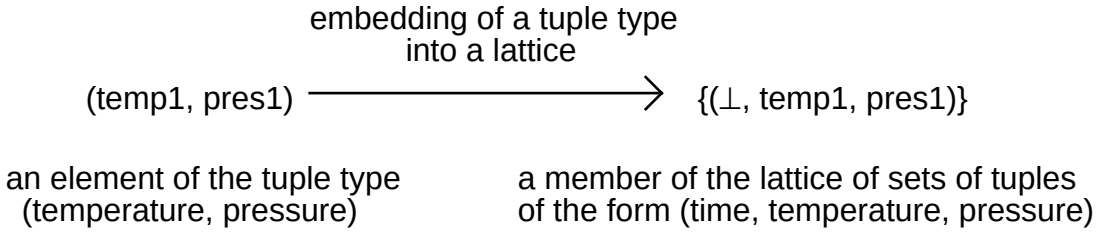


Figure 3.11. Embedding a tuple type into a lattice of sets of tuples.

Similarly, we can embed array data types in the data lattice. For example, consider the same lattice *U* defined from the three scalars *time*, *temperature* and *pressure*, and consider an array data type (*array* [*time*] *of temperature*). A data object of this type is a set of pairs of the form (*time*, *temperature*). We can embed such data objects into the lattice *U* as illustrated in Figure 3.12.

The basic ideas presented in Figures 3.11 and 3.12 can be combined to embed complex data types, defined as hierarchies of tuples and arrays, in data lattices. This will be formalized in Section 3.2.3. These embeddings enable a unified, lattice-structured data model so that visualization mappings apply to data objects of all data types. This is important for a visualization system based on this lattice model because it implies that the user interface for controlling how data are displayed is independent of data type.

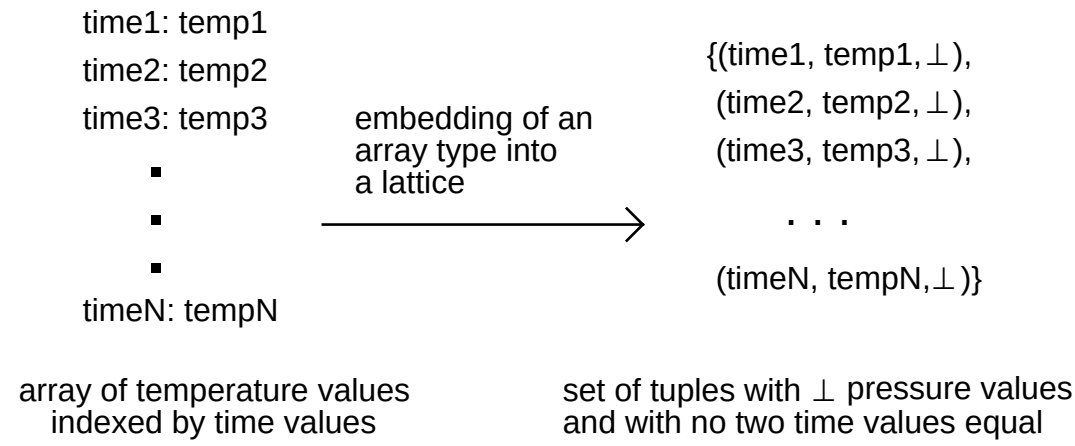


Figure 3.12. Embedding an array type into a lattice of sets of tuples.

### 3.2.2 Defining the Lattice Structure

Now we can develop a rigorous definition of our lattice model for scientific data. We will define lattices of data objects and displays in terms of scalar types. We use the symbol **R** to denote the real numbers. A scalar type *s* is either discrete or continuous and defines a set $I_S$ of values of type *s*.

**Def.** A *discrete scalar s* defines a countable value set $I_S$ that includes a least element $\perp$ and that has discrete order. That is, $\forall x, y \in I_S. (x \leq y \Rightarrow x = \perp)$. Figure 3.9 illustrates the order relation on $I_S$.

**Def.** A *continuous scalar s* defines a value set $I_S = \{\perp\} \cup \{[x, y] \mid x, y \in \mathbf{R} \ \& \ x \leq y\}$ (that is, the set of closed real intervals, plus $\perp$) with the order defined by: $\perp < [x, y]$ and $[u, v] \subseteq [x, y] \Leftrightarrow [x, y] \leq [u, v]$. Figure 3.1 illustrates the order relation on $I_S$.

Given a continuous scalar *s*, the closed real intervals in $I_S$ represent real numbers with limited accuracy. A real interval is "less than" its sub-intervals since sub-intervals give more precise information. Given a set *A* of closed real intervals, if the intersection $\bigcap A$ is non-empty then $\bigvee A$ is equal to that intersection (it is a closed interval), otherwise $\bigvee A$ is undefined. $\bigwedge A$ is the smallest closed interval containing the union $\bigcup A$, or $\perp$ if the union is unbounded.

It is interesting to note that, given a continuous scalar *s*, the order relation on $I_S$ encodes information about the ordering and topology of real numbers through the containment structure of intervals.

We use the notation $\mathbf{X}A$ for the cross product of members of the set *A*. We can now define an ordered set of tuples of scalar values, as follows:

**Def.** Let *S* be a finite set of scalars. Then the cross product $X = \mathbf{X}\{I_S \mid s \in S\}$ is the set of tuples with an element from each $I_S$. Let $a_S$ denote the *s* component of a tuple

$a \in X$. Define an order relation on $X$ by: for $a, b \in X$, $a \leq b$ if $\forall s \in S.\ a_S \leq b_S$. Figure 3.10 illustrates this order relation on tuples.

Let $POWER(X) = \{A \mid A \subseteq X\}$ denote the power set of $X$ (that is, the set of all subsets of $X$). As discussed briefly in Section 3.2.1, we use the sets of tuples in $POWER(X)$ as models for scientific data objects. It is well known that it is difficult to define an order relation on $POWER(X)$ that is consistent with the order relation on $X$ and is consistent with set inclusion (Schmidt, 1986). For example, if $a, b \in X$ and $a < b$, we would expect that $\{a\} < \{b\}$. Thus we might define an order relation between subsets of $X$ by:

(3.4) $\forall A, B \subseteq X.\ (A \leq B \Leftrightarrow \forall a \in A.\ \exists b \in B.\ a \leq b)$

However, given $a < b$, Eq. (3.4) implies that $\{b\} \leq \{a, b\}$ and $\{a, b\} \leq \{b\}$ are both true, which contradicts $\{b\} \neq \{a, b\}$. As explained by Schmidt, this problem can be resolved by defining an equivalence relation on $POWER(X)$. The equivalence relation is defined in terms of the Scott topology, which defines open and closed sets as follows:

**Def.** A set $A \subseteq X$ is *open* if $\uparrow A \subseteq A$ and, for all directed subsets $C \subseteq X$, $\bigvee C \in A \Rightarrow C \cap A \neq \phi$.

**Def.** A set $A \subseteq X$ is *closed* if $\downarrow A \subseteq A$ and, for all directed subsets $C \subseteq A$, $\bigvee C \in A$. We use $CL(X)$ to denote the set of all closed subsets of $X$.

Note that the complement of an open set is closed, and vice versa. Also, $X$ and $\phi$ are both open and closed.

**Def.** Define a relation $\leq_R$ on *POWER*(*X*) as: $A \leq_R B$ if for all open $C \subseteq X$, $A \cap C \neq \phi \Rightarrow B \cap C \neq \phi$. Also define a relation $\equiv_R$ on *POWER*(*X*) as: $A \equiv_R B$ if $A \leq_R B$ and $B \leq_R A$.

As we show in Appendix C, $\equiv_R$ is an equivalence relation. Clearly, if $A \equiv_R B$ and $C \equiv_R D$, then $A \leq_R C \Leftrightarrow B \leq_R D$, so the equivalence classes of $\equiv_R$ are ordered by $\leq_R$. In Appendix C we also show that the equivalence classes of $\equiv_R$ form a complete lattice, ordered by $\leq_R$. These equivalence classes are our models for data objects. However, it is not necessary to work directly with equivalence classes. Given an equivalence class *E* of the $\equiv_R$ relation, let $M_E = \bigcup E$. As shown in Appendix C, $M_E$ is closed and $E \leftrightarrow M_E$ defines a one-to-one correspondence between equivalence classes of $\equiv_R$ and closed sets. Thus we use $U = CL(X)$ as our data lattice. The following proposition from Appendix C explains how *sups* and *infs* are calculated in this lattice.

**Prop. C.8.** If *W* is a set of equivalence classes of the $\equiv_R$ relation, then $\bigwedge W$ is defined and equals *E* such that $M_E = \bigcap\{M_w \mid w \in W\}$. Similarly $\bigvee W$ is defined and equals *E* such that $M_E$ is the smallest closed set containing $\bigcup\{M_w \mid w \in W\}$. Thus the equivalence classes of the $\equiv_R$ relation form a complete lattice and, equivalently, *CL*(*X*) is a complete lattice. If *W* is finite and $E = \bigvee W$, then $M_E = \bigcup\{M_w \mid w \in W\}$.

To summarize, $U = CL(X)$ is a complete lattice whose members are in one to one correspondence with the equivalence classes of $\equiv_R$. The lattice *U* is our data model. Figure 3.13 illustrates the order relation on *CL*(*X*). In the next section we show that the data types of a scientific programming language can be naturally embedded in *U*.

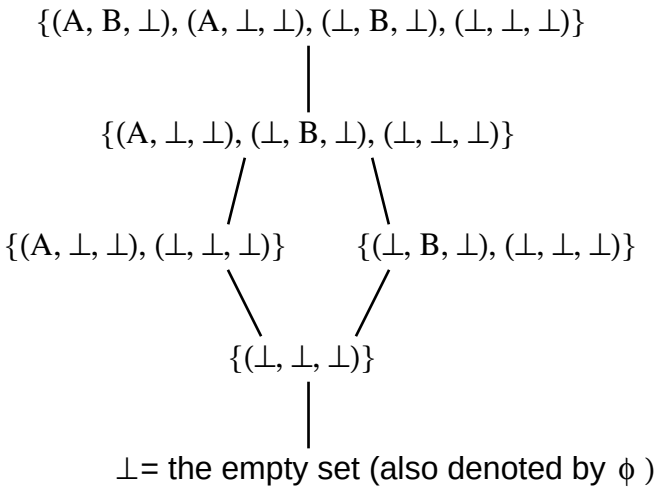


Figure 3.13. Defining an order relation on sets of tuples. The sets are all down sets and are ordered by set containment. We assume that the three scalars that define these tuples are discrete, so that the down sets in this figure are all finite.

### 3.2.3 Embedding Scientific Data Types in the Data Lattice

In this section we formalize the data model presented in Section 3.2.1.

**Def.** A set $T$ of *data types* can be defined from the set $S$ of scalars as follows. Two functions, $SC : T \rightarrow POWER(S)$ and $DOM : T \rightarrow POWER(S)$, are defined with $T$, as follows:

(3.5) $s \in S \Rightarrow s \in T$ (that is, $S \subset T$)

$SC(s) = \{s\}$

$DOM(s) = \phi$.

(3.6) (for $i = 1,...,n.\ t_i \in T$) & ($i \neq j \Rightarrow SC(t_i) \cap SC(t_j) = \phi$) $\Rightarrow$ $struct\{t_1;...;t_n\} \in T$

$SC(struct\{t_1;...;t_n\}) = \bigcup_i SC(t_i)$

$DOM(struct\{t_1;...;t_n\}) = \bigcup_i DOM(t_i)$

(3.7) $w \in S$ & $r \in T$ & $w \notin SC(r) \Rightarrow (array\ [w]\ of\ r) \in T$

$SC((array\ [w]\ of\ r)) = \{w\} \cup SC(r)$

$DOM((array\ [w]\ of\ r)) = \{w\} \cup DOM(r)$

The type $struct\{t_1;...;t_n\}$ is a *tuple* with *element* types $t_i$, and the type (*array* [*w*] *of r*) is an *array* with *domain* type *w* and *range* type *r*. *SC*(*t*) is the set of scalars occurring in *t*, and *DOM*(*t*) is the set of scalars occurring as array domains in *t*. Note that each scalar in *S* may occur at most once in a type in *T*.

In an actual implementation of a programming language, data objects must be represented as finite strings over finite alphabets, so only a countable number of data objects can be defined. Thus we define countable sets of values for scalar types and complex data types.

**Def.** For each scalar $s \in S$, define a countable set $H_S \subseteq I_S$ such that, for all $a, b \in H_S$, $a \wedge b \in H_S$, $a \vee b \in I_S \Rightarrow a \vee b \in H_S$, and for all $a \in I_S$ there exists $A \subseteq H_S$ such that $a = \bigvee A$ (that is, $H_S$ is closed under *infs* and under *sups* that belong to $I_S$, and any member of $I_S$ is a *sup* of a set of members of $H_S$). For discrete *s* this implies that $H_S = I_S$ (recall that we defined discrete scalars as having countable value sets). For continuous *s*, $H_S$ may be the set of rational intervals plus $\perp$. Note that, for continuous *s*, $H_S$ cannot be a *cpo*.

We can use the sets $H_S$ to define countable sets of finite data objects of all types. We define a tuple data object as a set containing one object of each of its element types. We define an array data object

as a function from a finite set of data objects of its domain type (which is a scalar type), to the set of data objects of its range type. Now we define countable sets of data objects of each type in *T*, and define functions that embed these data objects into the lattice *U*.

**Def.** Given a scalar *w*, let

$FIN(H_w) = \{A \subseteq H_w \backslash \{\bot\} \mid A \text{ finite and } \forall a, b \in A.\ \neg(a \leq b)\}$.

If *w* is a discrete scalar, then a member of $FIN(H_w)$ is any finite subset of $H_w$ not containing $\bot$. If *w* is a continuous scalar, then a member of $FIN(H_w)$ is any finite set of closed real intervals such that no interval contains another.

**Def.** For complex types $t \in T$ define $H_t$ by:

(3.8) $t = struct\{t_1;...;t_n\} \Rightarrow H_t = H_{t_1} \times ... \times H_{t_n}$

(3.9) $t = (array\ [w]\ of\ r) \Rightarrow H_t = \bigcup\{(A \to H_r) \mid A \in FIN(H_w)\}$

**Def.** Define an embedding $E_t : H_t \to U$ by:

(3.10) $t \in S \Rightarrow E_t(a) = \downarrow(\bot,...,a,...,\bot)$

(3.11) $t = struct\{t_1;...;t_n\} \Rightarrow E_t((a_1,...,a_n)) = \{b_1 \vee ... \vee b_n \mid \forall i.\ b_i \in E_{t_i}(a_i)\}$

(3.12) $t = (array\ [w]\ of\ r) \Rightarrow$

$$[a \in (A \to H_r) \Rightarrow E_t(a) = \{b \vee c \mid x \in A\ \&\ b \in E_w(x)\ \&\ c \in E_r(a(x))\}]$$

**Def.** For $t \in T$ define $F_t = E_t(H_t)$.

In Appendix D we show that $E_t$ does indeed map members of $H_t$ to members of *U*, and that this mapping is injective.

Recall that we use the notation $a_s$ for the *s* scalar component of a tuple $a \in \mathbf{X}\{I_s \mid s \in S\}$. Now $\mathbf{X}\{I_s \mid s \in S\}$ is not a lattice, so it is not obvious that $b_1 \vee ... \vee b_n$ in Eq. (3.11) and $b \vee c$ in Eq. (3.12) exist. However, as shown in Appendix D, for all $a \in H_t$ and for all $b \in E_t(a)$, $b_s = \perp$ unless $s \in SC(t)$. Thus $b_1 \vee ... \vee b_n$ in Eq. (3.11) exists since the types $t_i$ in Eq. (3.11) are defined from disjoint sets of scalars, and $b \vee c$ in Eq. (3.12) exists since the scalar *w* does not occur in the type *r*.

Because $E_t : H_t \rightarrow U$ is injective, we can define an order relation between the members of $H_t$ simply by assuming that $E_t$ is an order embedding. (If $E_t$ were not injective, it would map a pair of members of $H_t$ to the same member of *U*, and the assumption that $E_t$ is an order embedding would imply that the order relation on $H_t$ is not symmetric.)

**Def.** Given $a, b \in H_t$, we say that $a \leq b$ if and only if $E_t(a) \leq E_t(b)$.

Appendix D shows that the order relations on the sets $H_t$ implied by this definition have simple and intuitive structure. If *t* is a scalar, then this is the same as the order relation on $I_t$. If $t = struct\{t_1;...;t_n\}$ and if $(a_1,...,a_n), (b_1,...,b_n) \in H_t$, then $(a_1,...,a_n) \leq (b_1,...,b_n)$ if $\forall i.\ a_i \leq b_i$ (that is, the order relation between tuples is defined element-wise). If $t = (array\ [w]\ of\ r)$, if $a, b \in H_t$ and if $a \in (A \rightarrow H_r)$ and $b \in (B \rightarrow H_r)$, then

$a \leq b$ if $\forall x \in A.\ E_r(a(x)) \leq \bigvee\{E_r(b(y)) \mid y \in B\ \&\ x \leq y\}$ (that is, an array *a* is less than an array *b* if the embedding of the value of *a* at any sample *x* is less than the *sup* of the embeddings of the set of values of *b* at its samples greater than *x*).

In summary, in this section we have shown that data types appropriate for a scientific programming language can be embedded in our data model *U*. Thus, results about displaying data objects in *U* can be applied to the display of data objects of scientific algorithms.

### 3.2.4 A Finite Representation of Data Objects

If $S$ contains any continuous scalars, then most elements of $U = CL(X)$ contain infinite numbers of tuples. However, a closed set of tuples is only one member of an equivalence class of $\equiv_R$ as defined in Section 3.2.2. We can define an alternate representation of a data object as the set of maximal elements of a closed set, as follows:

**Def.** Given $A \in U$, define $MAX(A) = \{a \in A \mid \forall b \in A.\ \neg(a < b)\}$. That is, $MAX(A)$ consists of the maximal elements of $A$.

The following proposition from Appendix E tells us that the equivalence relation $\equiv_R$ defines a one-to-one correspondence between the closed sets in $U$ and the sets of their maximal elements.

**Prop. E.3.** $\forall A \in U.\ A \equiv_R MAX(A)$.

Thus, data objects in our data model can either be represented by closed sets, or by the sets of maximal elements of closed sets. As the following proposition from Appendix E shows, if $t$ is a data type in $T$, and if $A \in F_t$ is the embedding in $U$ of a data object of type $t$, then $MAX(A)$ is finite.

**Prop. E.5.** For all types $t \in T$ and all $A \in F_t$, $MAX(A)$ is finite.

Our lattice model of data is motivated by the observation that data objects are approximations to mathematical objects that may contain infinite amounts of information. Since our data lattice is complete it contains objects, definable as limits of objects of types in $T$, that are models for mathematical objects

containing infinite amounts of information. The sets of maximal tuples in these objects are generally not finite, so we cannot make the assumption that *MAX*(*A*) is finite when we apply Prop. B.3 to our scientific data model in Section 3.4. Thus working with sets of maximal tuples offers no real advantage over working with closed sets.

### 3.3 A Scientific Display Model

For our scientific display model we start with Bertin's analysis of static two-dimensional displays (Bertin, 1983). He modeled displays as sets of graphical marks, where each mark was described by an 8-tuple of graphical primitive values (that is, two screen coordinates, size, value, texture, color, orientation and shape). His idea of modeling a display as a set of tuple values is quite similar to the way we constructed the data lattice *U*. Therefore we define a finite set *DS* of *display scalars* to represent graphical primitives, we define $Y = \mathbf{X}\{I_d \mid d \in DS\}$ as the cross product of the value sets of the display scalars in *DS*, and we define *V* as the complete lattice of all closed subsets of *Y*. We interpret the maximal tuples of members of *V* as representing graphical marks (we show in Section 3.4.4 that for any type $t \in T$ and any data object $a \in H_t$, the display *D*(*a*) contains a finite number of maximal tuples), and we interpret the display scalar values in these maximal tuples as defining the graphical primitives of those graphical marks.

Bertin first published his display model in 1967, and it is limited to static two-dimensional displays. However, we can define a specific lattice *V* to model animated three-dimensional displays in terms of a set of seven continuous display scalars:

(3.13) $DS = \{x, y, z, red, green, blue, time\}$

A tuple of values of these display scalars represents a graphical mark. The interval values of *x*, *y* and *z* represent the locations and sizes of graphical marks in the volume, the interval values of *red*, *green* and

*blue* represent the ranges of colors of marks, and the interval values of *time* represent the duration of marks in an animation sequence, as illustrated in Figure 3.14.

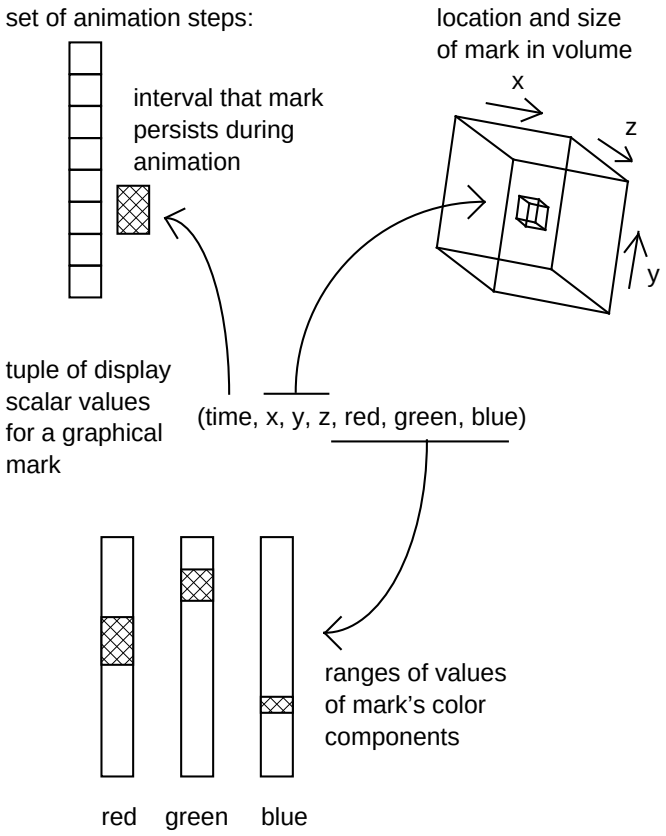


Figure 3.14. The roles of seven continuous display scalars (*x*, *y*, *z*, *red*, *green*, *blue*, *time*) in an animated three-dimensional display model.

The display lattice illustrated in Figure 3.14 models volume rendering and animation. Displays in $V$ are interactive in the sense that users control parameters to choose a function $RENDER : V \rightarrow V'$ that maps logical displays to physical displays (this function is described in Section 2.3). For the display lattice illustrated in Figure 3.14, users control the projection from three dimensions to two dimensions, and control animation sequencing. We can add more display scalars to *DS* to model other rendering techniques

and other user interaction techniques. For example, consider the display model defined by the following set of display scalars (where *n* and *m* are parameters of the display model):

(3.14) $DS = \{red, green, blue, transparency, reflectivity, vector_x, vector_y, vector_z, contour_1, ..., contour_n, x, y, z, animation, selector_1, ..., selector_m\}$

The *transparency* and *reflectivity* display scalars model parameters of volume rendering techniques. The $vector_x$, $vector_y$ and $vector_z$, display scalars model flow rendering techniques, and possibly interactive placement of seed points for tracing and rendering flow trajectories (a three-dimensional flow field is defined by the values of these display scalars attached to graphical marks). The $contour_1$, ..., $contour_n$ display scalars model iso-surface rendering techniques (iso-surfaces are rendered through the three-dimensional field defined by the values of these display scalars attached to graphical marks). The $selector_1$, ..., $selector_m$ display scalars explicitly model a user interaction technique. That is, a user interactively selects sets of values for each $selector_i$ (for *i* between 1 and *m*) and graphical marks are displayed only if their values for $selector_i$ overlap the user-selected set of values.

Display scalars can be defined for a wide variety of attributes of graphical marks, and need not be limited to such primitive values as spatial coordinates, color components and animation indices. For example, we may define a display model whose displays consist of sets of graphical icons (i.e., graphical shapes) distributed at various locations in a display screen. This display model could be defined using three display scalars: horizontal screen coordinate, vertical screen coordinate, and an icon identifier. In this display model a single value of the icon identifier display scalar would represent the potentially complex shape of a graphical icon. We could define another display model in which a set of display scalars form the parameters of two-dimensional ellipses. This display model would include five display scalars that represent the two-dimensional center coordinates, the orientations, and the lengths of major and minor axes of the ellipses.

The possibility that logical displays may be interactive suggests that we have great flexibility in the way we define a logical display model *V*, as long as we can define a family of mappings *RENDER* : *V* → *V'* parameterized by user controls. For example, we can build a display lattice *V* that models Beshers and Feiner's "worlds within worlds" visualization technique (Beshers and Feiner, 1992). This technique is an attempt to overcome the limitation to three spatial dimensions by nesting small coordinate systems within larger coordinate systems. Data are plotted as a set of small graphs, each including a small set of three axes. The location of the origin of a small coordinate system within a containing coordinate system determines the values of the containing coordinates for the plotted data. Users can interactively move the small graphs within the containing coordinate systems to see how plotted values change with respect to changes in the values of the containing coordinates. We can model this technique by defining a display lattice *V* in terms of two or more sets of three-dimensional graphics locations. The mapping *RENDER* : *V* → *V'* would be paramterized by the user's controls over the locations of small graphs.

The examples described above indicate that it is possible to define a wide variety of display models in terms of tuples of display scalars. Thus we do not focus on any particular display model. Rather, we just assume that there is a set *DS* of display scalars, and that our display model *V* consists of displays that are sets of maximal tuples of values of these display scalars.

The important point here is that the lattice model and its theoretical results are easily extensible to a wide variety of different display models. If a user can express rendering and interaction techniques in terms of a set of display scalars and user controls for the choice of the mapping *RENDER* : *V* → *V'*, then our lattice results are applicable to that model.

### 3.4 Scalar Mapping Functions

So far, we have defined a particular lattice structure appropriate for scientific data and displays. Now we apply the results of Section 3.1.4 to that structure.

### 3.4.1 Structure of Display Functions

Display functions are lattice isomorphisms. However, in the context of particular data and display models defined in the previous sections there is much more that we can say about them. Data objects of scalar types can be naturally embedded in the lattice $U$ (as we saw in Sections 3.3.2 and 3.3.3), and we can define similar embeddings of display scalar types in the lattice $V$. These embeddings can be defined as:

**Def.** For each scalar $s \in S$, define an embedding $E_S{:}I_S \rightarrow U$ by:

$\forall b \in I_S.\ E_S(b) = \downarrow(\perp,...,b,...,\perp)$ (this notation indicates that all components of the tuple are $\perp$ except $b$). Also define $U_S = E_S(I_S) \subseteq U$.

**Def.** For each display scalar $d \in DS$, define an embedding $E_d{:}I_d \rightarrow V$ by:

$\forall b \in I_d.\ E_d(b) = \downarrow(\perp,...,b,...,\perp)$. Also define $V_d = E_d(I_d) \subseteq V$.

These embedded scalars play a special role in the structure of display functions. As shown in Appendix F, a display function maps embedded scalar objects to embedded display scalar objects. Furthermore, the values of a display function on all of $U$ are determined by the values of the embedded scalar objects. The results of Appendix F are summarized by the following theorem about mappings from scalars to display scalars:

**Theorem. F.14.** If $D : U \rightarrow V$ is a display function, then we can define a mapping $MAP_D : S \rightarrow POWER(DS)$ such that for all scalars $s \in S$ and for all $a \in U_S$, there is $d \in MAP_D(s)$ such that $D(a) \in V_d$. The values of $D$ on all of $U$ are determined by the values of $D$ on the scalar embeddings $U_S$. Furthermore,

(a) If $s$ is discrete and $d \in MAP_D(s)$ then $d$ is discrete.

(b) If $s$ is continuous then $MAP_D(s)$ contains a single continuous display scalar.

(c) If $s \neq s'$ then $MAP_D(s) \cap MAP_D(s') = \phi$.

This theorem tells us that mappings of data aggregates to display aggregates can always be factored into mappings of data primitives (e.g., *time* and *temperature*) to display primitives (e.g., screen axes and color components). This has been accepted as intuitively true, as, for example, a time series of temperatures may be displayed by mapping *time* to one axis and *temperature* to another. However, Proposition F.14 tells us that all mappings that satisfy the expressiveness conditions must factor in this way. In Section 3.4.3 we present a precise statement of how such a factorization is a complete characterization of visualization mappings satisfying the expressiveness conditions.

Figure 3.15 provides examples of mappings from scalars to display scalars. The upper-right window of Figure 1.1 shows a display defined by these mappings. In this figure, *time* is mapped to *animation* so that the time sequence of images will be represented by animation (as opposed to being stacked up along a display axis, for example). *Line* and *element* are mapped to the *x* and *z* display axes and *ir* is mapped to the *y* axis, so that an image in the time sequence is displayed as a terrain (i.e., as a surface with *y* as a function of *x* and *z*). *vis* is mapped to *green*, so that this image terrain is colored green with intensity as a function of visible radiance.

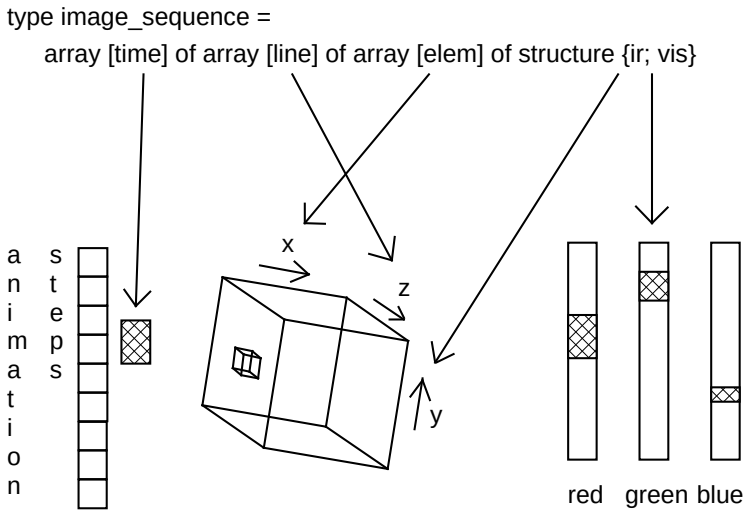

Figure 3.15. Mappings from scalars to display scalars.

### 3.4.2 Behavior of Display Functions on Continuous Scalars

In the previous section we saw that display functions map embedded continuous scalar objects to embedded continuous display scalar objects. Continuous scalar values are real intervals, so the values of display functions restricted to embedded continuous scalars can be analyzed in terms of their behavior as functions of real numbers. First, we define the values of display functions on embedded continuous scalars in terms of functions of real numbers.

**Def.** Given a display function $D{:}U \to V$ and a continuous scalar $s \in S$, by Prop. F.14 there is a continuous $d \in DS$ such that values in $U_S$ are mapped to values in $V_d$. Define functions $g_S : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ and $h_S : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ by:

$\forall \downarrow(\perp,...,[x, y],...,\perp) \in U_S$, $D(\downarrow(\perp,...,[x, y],...,\perp)) = \downarrow(\perp,...,[g_S(x, y), h_S(x, y)],...,\perp) \in V_d$.

Since $D(\{(\perp,...,\perp)\}) = \{(\perp,...,\perp)\}$ and $D$ is injective, $D$ maps intervals in $I_S$ to intervals in $I_d$, so $g_S(x, y)$ and $h_S(x, y)$ are defined for all $z$. Also define functions $g'_S : \mathbf{R} \to \mathbf{R}$ and $h'_S : \mathbf{R} \to \mathbf{R}$ by $g'_S(z) = g_S(z, z)$ and $h'_S(z) = h_S(z, z)$.

As shown in Appendix G, the functions $g_S$ and $h_S$ can be defined in terms of the functions $g'_S$ and $h'_S$, as follows. Given a display function $D{:}U \to V$, a continuous scalar $s \in S$, and $[x, y] \in I_S$, then

(3.15) $g_S(x, y) = inf\{g'_S(z) \mid x \le z \le y\}$ and

(3.16) $h_S(x, y) = sup\{h'_S(z) \mid x \le z \le y\}$.

In Appendix G we also show that the overall behavior of a display function on a continuous scalar must fall into one of two categories. Specifically, given a display function $D{:}U \rightarrow V$ and a continuous scalar $s \in S$, then either

(3.17) $\forall x, y, z \in \mathbf{R}.\ x < y < z$ implies that $g_S(x, z) = g_S(x, y)$ & $h_S(x, y) < h_S(x, z)$ and that $g_S(x, z) < g_S(y, z)$ & $h_S(y, z) = h_S(x, z)$,

or

(3.18) $\forall x, y, z \in \mathbf{R}.\ x < y < z$ implies that $g_S(x, z) < g_S(x, y)$ & $h_S(x, y) = h_S(x, z)$ and that $g_S(x, z) = g_S(y, z)$ & $h_S(y, z) < h_S(x, z)$.

If Eq. (3.17) applies, we say that $D$ is *increasing* on $s$. If Eq. (3.18) applies, we say that $D$ is *decreasing* on $s$.

Appendix G shows that these categories also apply to the functions $g'_S$ and $h'_S$. Given a display function $D{:}U \rightarrow V$, a continuous scalar $s \in S$, and $z < z'$, if $D$ is increasing on $s$ then $g'_S(z) < g'_S(z')$ and $h'_S(z) < h'_S(z')$, and if $D$ is decreasing on $s$ then $g'_S(z) > g'_S(z')$ and $h'_S(z) > h'_S(z')$.

These categories enable us to prove (see Appendix G) that the functions $g'_S$ and $h'_S$ must be continuous (in terms of the topology of the real numbers), and that they satisfy a number of other conditions, summarized in the following definition.

**Def.** A pair of functions $g'_S{:}\mathbf{R} \rightarrow \mathbf{R}$ and $h'_S{:}\mathbf{R} \rightarrow \mathbf{R}$ is called a *continuous display pair* if:

(a) $g'_S$ has no lower bound and $h'_S$ has no upper bound,

(b) $\forall z \in \mathbf{R}.\ g'_S(z) \leq h'_S(z)$, and

(c) $g'_S$ and $h'_S$ are continuous,

(d) either $g'_S$ and $h'_S$ are increasing:

$\forall z, z' \in \mathbf{R}.\ z < z' \Rightarrow g'_S(z) < g'_S(z') \ \&\ h'_S(z) < h'_S(z')$,

or $g'_S$ and $h'_S$ are decreasing:

$\forall z, z' \in \mathbf{R}.\ z < z' \Rightarrow g'_S(z) > g'_S(z') \ \&\ h'_S(z) > h'_S(z')$.

Given a display function $D{:}U \rightarrow V$ and a continuous scalar $s \in S$, then $g'_S$ and $h'_S$ are a continuous display pair. If we draw the graphs of the functions $g'_S$ and $h'_S$, these conditions tell us that their graphs must be smooth, both slanted up or both slanted down, with the graph of $h'_S$ above the graph of $g'_S$, no upper bound on the graph of $h'_S$, and no lower bound on the graph of $g'_S$. A display function maps closed real intervals in a continuous scalar to closed real intervals in a continuous display scalar, and the graphs of functions $g'_S$ and $h'_S$ can be used to determine this mapping of intervals by applying Eqs. (3.15) and (3.16). The behavior of $g'_S$ and $h'_S$ is illustrated in Figure 3.16.

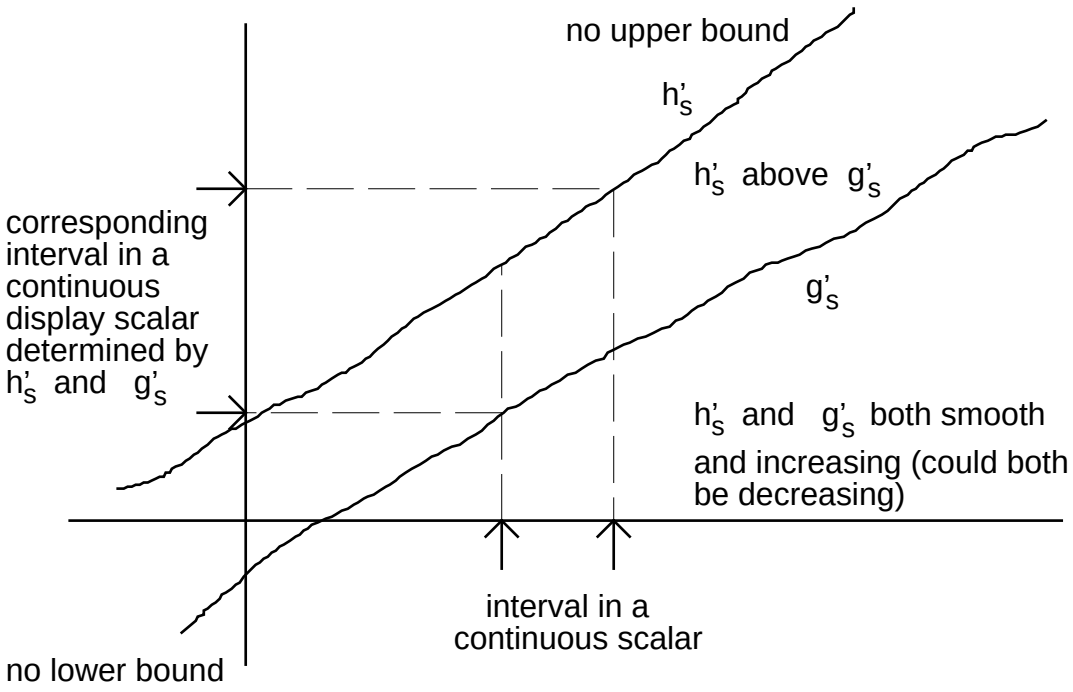

Figure 3.16. The behavior of a display function $D$ on a continuous scalar interpreted in terms of the behavior of functions $h'_S$ and $g'_S$.

### 3.4.3 Characterizing Display Functions

The results of the last two sections describe a variety of necessary conditions on display functions. Here we summarize those conditions, and show that they are also sufficient conditions for display functions.

**Def.** Given a finite set $S$ of scalars, a finite set $DS$ of display scalars, $X = \mathbf{X}\{I_s \mid s \in S\}$, $Y = \mathbf{X}\{I_d \mid d \in DS\}$, $U = CL(\mathrm{X})$, and $V = CL(\mathrm{Y})$, then a function $D{:}U \to V$ is a *scalar mapping function* if

(a) there is a function $MAP_D : S \to POWER(DS)$ such that

$\forall s, s' \in S.\ MAP_D(s) \cap MAP_D(s') = \phi$,

(b) for all continuous $s \in S$, $MAP_D(s)$ contains a single continuous $d \in DS$,

(c) for all discrete $s \in S$, all $d \in MAP_D(s)$ are discrete,

(d) $D(\phi) = \phi$ and $D(\{(\perp,...,\perp)\}) = \{(\perp,...,\perp)\}$,

(e) for all continuous $s \in S$, $g'_S$ and $h'_S$ are a continuous display pair,

for all $[u, v] \in I_S$, $g_S(u, v) = inf\{g'_S(z) \mid u \le z \le v\}$ and

$h_S(u, v) = sup\{h'_S(z) \mid u \le z \le v\}$,

and, given $\{d\} = MAP_D(s)$, then for all $[u, v] \in I_S \backslash \{\perp\}$,

$D(\downarrow(\perp,...,[u, v],...,\perp)) = \downarrow(\perp,...,[g_S(u, v), h_S(u, v)],...,\perp) \in V_d$,

(f) for all discrete $s \in S$, for all $a \in I_S \backslash \{\perp\}$,

$D(\downarrow(\perp,...,a,...,\perp)) = \mathrm{b} \in V_d$ for some $d \in MAP_D(s)$, where $b \neq \{(\perp,...,\perp)\}$,

and, for all $a, a' \in I_S \backslash \{\perp\}$, $a \neq a' \Rightarrow D(\downarrow(\perp,...,a,...,\perp)) \neq D(\downarrow(\perp,...,a',...,\perp))$

(g) for all $x \in X$, $D(\downarrow x) = \downarrow \bigvee\{y \mid \exists s \in S.\ x_S \neq \perp \ \&\ \downarrow y = D(\downarrow(\perp,...,x_S,...,\perp))\}$,

where $x_S$ represents tuple components of $x$, and using the values for $D$ defined in (e) and (f), and

(h) for all $u \in U$, $D(u) = \bigvee\{D(\downarrow x) \mid x \in u\}$, using the values for $D$ defined in (g).

This definition contains a variety of expressions for the value of $D$ on various subsets of $U$. Appendix H shows that these expressions are consistent where the subsets of $U$ overlap, and shows that $D$ is monotone. This definition says that scalar mapping functions factor into mappings from scalars (data primitives) to display scalars (primitives), and that the factor mappings on continuous scalars are continuous real functions. In Appendix H we also prove the following characterization of display functions:

**Theorem H.8.** $D : U \to V$ is a display function if and only if it is a scalar mapping function.

Appendix H also shows that the values of a scalar mapping function $D$ can be expressed in terms of an auxiliary function $D'$ from $X$ to $Y$. Specifically, for all $u \in U$,

(3.19) $D(u) = \{D'(x) \mid x \in u\}$.

where $D'$ is defined by

(3.20) $D'(x) = \bigvee\{(\perp,...,a_d,...,\perp) \mid s \in S \;\&\; x_S \neq \perp \;\&\; D(\downarrow(\perp,...,x_S,...,\perp)) = \downarrow(\perp,...,a_d,...,\perp)\}$

This decomposition can be used as a basis for implementing scalar mapping functions, and scalar mapping functions can be used as the basis of a user interface for controlling the display process. We will describe this further in Section 3.4.4.

Theorem H.8 can also be used as a precise definition of the search space of display functions for algorithms that attempt to automate the design of displays.

### 3.4.4 Properties of Scalar Mapping Functions

There is a problem with the interpretation of display objects in a display lattice. Closed sets generally contain infinite numbers of tuples, so we cannot interpret each tuple as a graphical mark. However, as described in Section 3.2.4, a closed set is just one member of an equivalence class of the $\equiv_R$ relation. A closed set $v \in V$ and its set of maximal tuples, $MAX(v)$, are both members of the same equivalence class and thus either can represent a display object. As shown in Appendix I, if $D$ is a display function and if
$v \in D(F_t)$ for some data type $t \in T$, then $MAX(v)$ contains a finite number of tuples. Thus, in order to physically render a display object $v \in V$, we interpret the finite set of tuples in $MAX(v)$ as graphical marks, rather than the possibly infinite set of tuples of $v$. Clearly, it is necessary for an implementation of the function $RENDER : V \rightarrow V'$ to assume a finite number of input tuples.

In order to compute values of scalar mapping functions we use the auxiliary function $D'$ from $X$ to $Y$ defined in Section 3.4.3. The values of $D'$ are determined by the function $MAP_D$, by the values of the functions $g'_S$ and $h'_S$ for continuous scalars $s \in S$, and by the values of $D$ on $U_S$ for discrete scalars $s \in S$. As shown in Appendix I, given
$t \in T$ and a data object $A \in F_t$, maximal tuples of $D(A)$ can be computed directly from the maximal tuples of $A$ by

(3.21) $MAX(D(A)) = \{D'(a) \mid a \in MAX(A)\}$

As shown in Appendix D, the maximal tuples of data objects of type $t \in T$ are computed by

(3.22) $t \in S \;\&\; A = \downarrow(\perp,...,a,...,\perp) \in F_t \Rightarrow$

$$MAX(A) = \{(\perp,...,a,...,\perp)\}$$

(3.23) $t = struct\{t_1;...;t_n\} \in T \;\&\; A = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} \in F_t \Rightarrow$

$$MAX(A) = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in MAX(A_i)\}$$

(3.24) $t = (array\ [w]\ of\ r) \in T \;\&\; A = \{a_1 \vee a_2 \mid g \in G \;\&\; a_1 \in E_w(g) \;\&\; a_2 \in E_r(a(g))\} \in F_t$

$$\Rightarrow MAX(A) = \{a_1 \vee a_2 \mid g \in G \;\&\; a_1 \in MAX(E_w(g)) \;\&\; a_2 \in MAX(E_r(a(g)))\}$$

These expressions for sets of maximal tuples and the auxiliary function *D'* provide a basis for implementing scalar mapping functions. Given a data object *A*, Eqs. (3.22) through (3.24) define a recursive procedure for calculating the maximal tuples of *A*, and Eq. (3.21) says that the function *D'* maps maximal tuples of *A* to maximal tuples of *D*(*A*).

In Section 3.3 we described displays as sets of graphical marks. However, we can also think of displays as defining functional relations from graphical space and time to color. That is, the color of a screen point is a function of its location on the screen and its place in an animation sequence. These two views of displays, as sets of graphical marks and as functions, are not consistent. For example, consider the display lattice illustrated in Figure 3.14. If a display in this lattice includes two tuples (*time*, *x*, *y*, *z*, $red_1$, $green_1$, $blue_1$) and (*time*, *x*, *y*, *z*, $red_2$, $green_2$, $blue_2$) where $red_1 \neq red_2$, $green_1 \neq green_2$ or $blue_1 \neq blue_2$, then these two tuples do not define a consistent function from space and time to color. In order to analyze the circumstances under which these two views are consistent, we divide display scalars into two groups: those that take the role of dependent variables in this functional relation and those that take the role of independent variables. For example, the set *DS* defined in Eq. (3.14) can be divided as follows:

Independent variables: *x*, *y*, *z*, *animation*, $selector_1$, ..., $selector_m$

Dependent variables: *red*, *green*, *blue*, *transparency*, *reflectivity*, $vector_x$, $vector_y$, $vector_z$, $contour_1$, ..., $contour_n$,

Thus we can ask whether a display function generates displays that define functional relations between independent and dependent variables in *DS*. Define a subset $V_{display} \subseteq V$ consisting of those display objects that do not contain multiple tuples with the same combination of values of independent variables. We will study the conditions under which displays of data objects are members of $V_{display}$.

First, define *DOMDS* = the independent variables in *DS*, and define $Y_{DOMDS} = \mathbf{X}\{I_d \mid d \in DOMDS\}$ and $Y = \mathbf{X}\{I_d \mid d \in DS\}$. Let $P_{DOMDS} : Y \rightarrow Y_{DOMDS}$ be the natural projection from *Y* onto $Y_D$ (that is, if $a \in Y$ and $b = P_{DOMDS}(a)$, then for all $d \in DOMDS$, $b_d = a_d$). Then we can define $V_{display}$ as follows:

**Def.** $V_{display} = \{A \in V \mid \forall b, c \in MAX(A).\ P_{DOMDS}(b) = P_{DOMDS}(c) \Rightarrow b = c\}$. That is, if *A* is an object in $V_{display}$, then multiple tuples in *A* do not share the same combinations of values for display scalars in *DOMDS*.

Appendix I defines conditions on *t* and *D* that ensure that displays of data objects of type *t* are in $V_{display}$. Specifically, *D* maps all data objects of type *t* to displays in $V_{display}$ if *D* maps all scalars in *DOM*(*t*) to display scalars in *DOMDS*. Symbolically, $MAP_D(DOM(t)) \subseteq DOMDS \Rightarrow D(F_t) \subseteq V_{display}$.

The inverse of this relation is almost true - we only need to disallow degenerate cases. Details are given in Appendix I.

In summary, in this section we have shown that the number of tuples in a display may be infinite, but that the number of maximal tuples is finite. We concluded that only maximal tuples should be

interpreted as graphical marks in an actual implementation. We have also described a recursive procedure for computing the set of maximal tuples in a data object and described how maximal tuples of displays are computed from maximal tuples of data objects. This provides a basis for implementing display functions.

We have also demonstrated conditions on data types and display functions so that display objects are consistent with a functional view of displays. An implementation could enforce these conditions on scalar mappings defined by users. We note, however, that the VisAD implementation described in Chapter 4 does not enforce these conditions. Rather, multiple tuples that are inconsistent with a functional view of display (i.e., occurring at the same location and time) are merged using a compositing operation (that is, the system computes the average colors of multiple tuples at the same location and time).

**3.5 Principles for Scientific Visualization**

In this chapter we analyzed the repertoire of visualization mappings from a lattice-structured data model to a lattice-structured display model. In this section we summarize the results of this analysis as a set of basic principles for visualization.

We showed how a lattice structure can express metadata about the ways that scientific data objects are approximate representations of mathematical objects. We also showed that this idea can be applied to scientific displays. Our first basic principle is that

1. Lattice-structured data models provide a natural way to integrate common forms of scientific metadata as part of data objects.

We gave an example of how a lattice-structured data model includes data objects of many different types, and we will describe another example in Chapter 5. Our second basic principle is that

2. Data objects of many different types can be unified into a single lattice-structured data model, so that visualization mappings (to a display model) are inherently polymorphic.

We have shown how lattice-structured data and display models can be adapted very generally by applying Eq. (3.2). We have shown that Mackinlay's expressiveness conditions on the visualization mapping can be interpreted in terms of such models and that these conditions imply that visualization mappings are lattice isomorphisms. Our third basic principle is that

3. Lattice-structured data models and display models may be defined in a very general set of scientific situations, and the lattice isomorphism result can be broadly applied to analyze the repertoire of visualization mappings between them.

We have shown how to define a lattice-structured data model that allows data aggregates to be defined as hierarchies of tuples and arrays. We have shown how a similar lattice structure can define a model for interactive, animated, three-dimensional displays. By applying the lattice isomorphism result in this context, we have established our fourth basic principle that

4. Mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives.

While our fourth principle has been accepted as intuitive in the past, here we have shown that it completely characterizes all visualization mappings that satisfy the expressiveness conditions.

# Chapter 4

# Applying the Lattice Model to the Design of Visualization Systems

In Chapter 2 we developed the following design components for the VisAD system for visualizing scientific computations:

1. That it is integrated with a scientific programming language. The system has an integrated user interface for programming, computation and display.

2. That the data types of that programming language are constructed as tuples and arrays from a set of scalar types. Data objects of these types represent mathematical variables, vectors and functions.

3. That its displays are interactive, animated and three-dimensional. These logical displays are mapped to physical displays by a variety of familiar rendering operations.

In this chapter we will continue that development, guided by the broad goals defined in Section 1.1, by the analysis of visualization repertoires in Chapter 3, and by the basic principles defined in Section 3.5. To review, our goals are to develop visualization techniques that

1. Can be applied to the data of a wide variety of scientific applications.

2. Can produce a wide variety of different visualizations of data appropriate for different needs.

3. Enable users to interactively alter the ways data are viewed.

4. Require minimal effort by scientists.

5. Can be integrated with a scientific programming environment.

The basic principles are

1. Lattice-structured data models provide a natural way to integrate common forms of scientific metadata as part of data objects.

2. Data objects of many different types can be unified into a single lattice-structured data model, so that visualization mappings (to a display model) are inherently polymorphic.

3. Lattice-structured data models and display models may be defined in a very general set of scientific situations, and the lattice isomorphism result can be broadly applied to analyze the repertoire of visualization mappings between them.

4. Mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives.

**4.1 Integrating Metadata with a Scientific Data Model**

Our first goal developed in Section 1.1 was that scientific visualization techniques "*Can be applied to the data of a wide variety of scientific applications*." Thus in Section 2.2 we developed a flexible way to define data types based on the assumption that data objects represent mathematical objects. However, as we described in Section 1.2.2, scientific data includes metadata as well as data types. The first principle of Chapter 3 tells us that a lattice-structured data model provides a natural way to integrate

common forms of scientific metadata as part of data objects, and thus handle a greater variety of data. In this section we describe the ways that our visualization design integrates metadata.

The VisAD system allows data types to be defined as tuple and array aggregates of named scalar types. Scalar types may be defined with any of the following primitive types:

1. Integers.

2. Text strings.

3. Real numbers (these values are always taken from a specified finite sampling of real numbers, and intervals around these values are implicit in the spacing between samples).

4. Pairs of real numbers (these values are always taken from a finite sampling of $\mathbf{R}^2$ and rectangles around values are implicit in the spacing between samples).

5. Triples of real numbers (these values are always taken from a finite sampling of $\mathbf{R}^3$ and rectangular solids around values are implicit in the spacing between samples).

These types of primitive values do not precisely correspond to the scalar types defined in Chapter 3. Integer and text string primitives do correspond to discrete scalars. Real number primitives correspond to the continuous scalars of Chapter 3, except that the intervals around values are implicit. They are included in our system as a compromise between the computational efficiency of real numbers and the explicit accuracy information of real intervals. Primitives for pairs and triples of real numbers do not correspond to the scalars of Chapter 3. They are included in our system because they occur commonly in scientific data and can be handled more efficiently as primitives. Furthermore, metadata are integrated at the level of

primitive values, so handling two- and three-dimensional real values as primitives enables the system to integrate a wider variety of metadata. Specifically, these primitives allow samplings of $\mathbf{R}^2$ and $\mathbf{R}^3$ that are not Cartesian products of samplings of **R**.

The system integrates the following forms of metadata:

1. Sampling information: Every value in a data object is taken from a finite sampling of primitive values. That is, the system includes internal structures that specify finite samplings of the five primitive types, and associates every primitive value with one of these structures. For array index values, this finite sampling determines the way the array samples a function's domain, and thus determines the size of the array.

2. Accuracy information: This is implicit in the resolution of samplings, rather than the explicit intervals described in Chapter 3.

3. *Missing* data indicators: Any value or sub-object in a data object may take the special value *missing* (indicating the lack of information).

4. Names for values: Every primitive value occurring in a data object has a scalar type, and hence a name (that is, the name of the scalar type).

The integration of metadata into data objects has important consequences for computational semantics. For example, consider the following data types appropriate for satellite images:

```
type radiance = real;
type earth_location = real2d;
```

```
type image = array [earth_location] of radiance;
```

and the following declarations of data objects:

```
earth_location loc;
image goes_east, goes_west, goes_diff;
```

The scalar data object *loc* will take a pair of real numbers as a value - the latitude and longitude of a location on the Earth. The array data object *goes_east* contains a finite set of samples of an Earth radiance field, indexed by {latitude, longitude} pairs. The value of the expression *goes_east*[*loc*] is an estimate of the value of this radiance field at the Earth location in *loc*. There are a variety of interpolation methods for making this estimate - the VisAD implementation simply takes the value of the sample in *goes_east* nearest to *loc*. If *loc* falls outside the range of samples of *goes_east*, the expression evaluates to *missing*.

Now consider the program fragment:

```
sample(goes_diff) = goes_east;
foreach (loc in goes_east) {
    goes_diff[loc] = goes_east[loc] - goes_west[loc];
}
```

The first line specifies that *goes_diff* will have the same sampling of array index values (that is, of pixel locations) that *goes_east* has. The *foreach* statement provides a way to iterate over the elements of an array. In this case it iterates *loc* over the pixel locations of the *goes_east* image. The expression *goes_east*[*loc*] - *goes_west*[*loc*] is evaluated by estimating the value of (the radiance field represented by) *goes_west* at *loc*, and then subtracting this value from *goes_east*[*loc*]. Any arithmetic operation with a

*missing* operand evaluates to *missing*, so *goes_diff*[*loc*] is set to *missing* if *goes_west*[*loc*] evaluates to *missing*. (Note that *missing* data are natural values for undefined arithmetic operations such as division by zero.)

The VisAD implementation provides vector operations, so this computation may also be expressed as:

```
goes_diff = goes_east - goes_west;
```

All the semantics of the previous program fragment are implicit in this statement.

Satellite images are finite arrays of pixels. Pixel radiances are typically represented by coded 8-bit or 10-bit values. The most important metadata accompanying satellite images are called *navigation*, which defines the Earth locations of pixels, and *calibration*, which defines the radiance values associated with coded pixel values. *Missing* data indicators are also important for satellite data since telemetry failures are common. Our visualization design can integrate all of these forms of metadata. Satellite navigation metadata can be integrated as the samplings associated with the *real2d* indices of image arrays, satellite calibration metadata can be integrated as the samplings associated with *real* radiance values in image arrays, and *missing* data are integrated with any data type. These forms of metadata are implicit in the computational semantics of the VisAD programming language. In Section 1.1 our fourth goal was that visualization techniques should "*Require minimal effort by scientists*." The programming example above shows that the integration of metadata into data objects relieves scientific programmers of the need to:

1. Keep track of *missing* data.

2. Manage the mapping, including interpolation, from array index values to physical values (such as Earth latitude and longitude).

3. Check bounds on array accesses.

The integration of metadata into data objects also affects their display semantics. For example, Figures 4.1 shows satellite image data displayed in a Cartesian Earth coordinate system defined by latitude and longitude. The system geographically registers this image data object using the integrated satellite navigation metadata, relieving the user of the need to manage the association between images and their navigation information when images are displayed. Figure 4.2 shows an image generated by a polar orbiting satellite, displayed in an Earth-centered spherical coordinate system.

The integration of *missing* data also affects display semantics. Figure 4.3 is a nearly edge-on view of a three-dimensional array of radar echoes. It is traditional to treat the lack of echoes as *missing* rather than zero, since information about spectrum and polarity is not available where there are no echoes. The *missing* values are simply invisible in Figure 4.3.

Figure 4.1. A satellite image displayed in a Cartesian Latitude / Longitude coordinate system. (color original)

Figure 4.2. An image from a polar orbiting satellite displayed in a three-dimensional Earth coordinate system. (color original)

Figure 4.3. Three-dimensional radar data. (color original)

The VisAD system integrates accuracy information with its data objects only implicitly as the resolution of value samplings. However, our system design could easily integrate this form of metadata explicitly by using real intervals as described in Section 3.2. Interval arithmetic could be used for the computational semantics of interval values (Moore, 1966), including the use of two and three-dimensional rectangles as values for two and three-dimensional real primitives.

The samplings associated with values can be exploited for a simple form of data compression. If a variable takes a value from a set of 255 samples plus *missing*, then that variable can be stored in a single byte. Thus programs can written as if satellite radiances are real numbers, but they may be stored as 8-bit codes in bytes.

**4.2 Interacting with Scientific Displays**

In Section 3.3 we discussed how a lattice-structured display model *V* can be defined in terms of a set of display scalars (i.e., graphical primitives). The graphical primitives of Bertin's display model were 2-D location, size, value, texture, color, orientation, and shape. Shape and texture are different from Bertin's other primitives in the sense that they can be composed as graphical aggregates. Thus we do not treat them as primitives in the VisAD display model. The fourth principle of Section 3.5 tells us that mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives. Thus shapes and textures in VisAD's displays represent shapes and textures in data according to this principle. For example, in Figure 4.4 an aggregate of primitive points form a complex shape. Each point corresponds to an individual observation of an X-ray emanating from interstellar gas. The overall shape of these points communicates a great deal about the functioning of the instrument that made these observations.

Figure 4.4. X-ray events from interstellar gas. (color original)

Bertin restricted his model to physical displays: static two-dimensional arrays of color. As discussed in Section 2.3, our design uses logical displays that may are animated, three-dimensional and interactive. We distinguish between a set *V'* of physical displays and a set of logical displays *V*. We define a mapping *RENDER* : $V \rightarrow V'$ that implements the traditional graphics pipeline for iso-surface extraction, projection from three to two dimensions, clipping, animation, and so on. The VisAD system's display model is defined in terms of the following display scalars:

(4.1) $DS = \{color, contour_1, ..., contour_n, x, y, z, animation, selector_1, ..., selector_m\}$

Using the terminology of Chapter 3, a maximal tuple in $Y = \mathbf{X}\{I_d \mid d \in DS\}$ represents a graphical mark in a display. Given a maximal tuple, its *x*, *y* and *z* values specify the corresponding graphical mark's location and size in a virtual three-dimensional graphics space, its *color* value specifies the mark's color, and its *animation* value specifies the mark's place and duration in an animated sequence of images, as illustrated in Figure 3.14. The $contour_i$ display scalars are similar to *color* in that they help determine how a mark appears, rather than where or when it appears. For each *i*, the $contour_i$ values in tuples are resampled to a value field distributed over a three-dimensional voxel array. These fields are depicted by iso-level surfaces and curves rendered through the voxel array. The $selector_i$ display scalars are similar to *animation* in that they help determine when a mark appears, rather than where or how it appears. The user selects a set of values for each $selector_i$, and only those tuples whose $selector_i$ interval values overlap with this set are included in the display. Note that just as the VisAD data model includes two- and three-dimensional real primitives, the display model includes the three-dimensional real primitive *color*, includes two- and three-dimensional real primitives for various combinations of graphical location (e.g., *xy_plane*), and allows *selector* scalars to take the dimensionality of the scalars mapped to them.

In Chapter 3 we developed a detailed analysis of the repertoire of visualization mappings from lattice-structured data models to lattice-structured display models. The data and display models of the

VisAD system do not precisely conform to the assumptions in Theorem H.8, so it cannot be applied to VisAD in exact form. However, the VisAD system does implement the essential structure of scalar mapping functions. Visualization mappings of aggregate data objects are factored into continuous functions from scalar types to display scalar types. VisAD deviates from the scalar mapping functions of Theorem H.8 by including continuous functions of two- and three-dimensional real scalars. Users control how data are displayed by defining a set of mappings from scalar types to display scalar types.

We can illustrate the way that mappings from scalar types to display scalar types control data displays by an example. The following data types are defined for a time sequence of satellite images:

```
type earth_location = real2d;
type ir_radiance = real;
type vis_radiance = real;
type variance = real;
type texture = real;
type time = real;
type image_region = integer;
type image =
    array [earth_location] of
        structure {
            ir_radiance;
            vis_radiance;
            variance;
            texture;
        }
type image_partition = array [image_region] of image;
type image_sequence = array [time] of image_partition;
```

Each *image* pixel contains infrared and visible radiances, and variance and texture values derived from infrared radiances. An *image_sequence* is a *time* sequence of *images*, each partitioned into rectangular regions (which are indexed by *image_region*). These types include seven scalars, so users control the way that data objects are displayed by defining mappings from these seven scalars to seven display scalars. In the VisAD system these mappings are defined using a simple text editor. Figure 4.5 shows a data object of the *image_sequence* type displayed as a colored terrain, after specifying the following mappings:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map variance to selector;
map texture to selector;
map image_region to selector;
map time to animation;
```

The user can use the same display scalar name *selector* in more than one mapping since the system differentiates multiple occurrences of *selector* into $selector_1$, $selector_2$, etc.

Note that the VisAD system supplies default continuous functions from scalars to display scalars when they are not included in the specification of scalar mappings (as they are not included in the above mappings). The default functions are linear from the range of samplings of the scalar values to the range of display scalar values. In practice these defaults almost always work well and make the user's task easier.

Figure 4.5. A *goes_sequence* object displayed as a terrain (i.e., a height function), with *ir* radiance mapped to terrain height (the *y* axis) and *vis* radiance mapped to *color*. All sixteen image *region* values are selected for display. The *time* sequence may be animated. (color original)

The second and fourth goals developed in Section 1.1 state that visualization techniques "*Can produce a wide variety of different visualizations of data appropriate for different needs*" and "*Require minimal effort by scientists*." The scalar mapping functions used in VisAD are effective at realizing these goals, and this effectiveness can be explained in terms of the basic principles developed in Section 3.5. The fourth principle tells us that mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives. Thus any way of displaying data that satisfies the effectiveness conditions can be specified by a set of mappings from scalars to display scalars. The second principle tells us that, because of the way that data objects of many different types are unified into a single lattice-structured data model, visualization mappings are inherently polymorphic. The fact that a single display mapping $D : U \rightarrow V$ applies to data objects of many types in $U$ has a beneficial impact on the VisAD system's user interface: a single set of scalar mappings control how all data objects in a user's program are displayed. Once a user defines a set of scalar mappings, he can select any data object for display merely by graphically picking its name. Display controls are separate from a user's scientific programs, unlike previous visualization systems that require calls to visualization functions to be embedded in programs.

In Section 3.4 we noted that our lattice-structured display model was inconsistent with a functional view of display (i.e., the view that a display defines a functional relation from location and time to color). We developed a set of constraints on scalar mapping functions (these constraints also depend on the type of the data object being displayed) that guarantee that they generate only displays that are consistent with a functional view of display. However, we have chosen not to enforce these constraints in the VisAD system. We use the VisAD system for experimenting with visualization ideas, and have generally opted against restrictions on what users may do.

For example, we have even used VisAD to experiment with visualization mappings that do not satisfy the expressiveness conditions. For example, we experimented with a way of mapping more than one scalar to a display scalar (display scalar values were calculated as the sum of values they would have

from each scalar alone). While this feature did produce some interesting images, we generally found that it was not used by scientists. This experience tends to confirm the value of the expressiveness conditions.

The third goal developed in Section 1.1 states that visualization techniques "*Enable users to interactively alter the ways data are viewed*." The VisAD design realizes this goals by making the specification of the mappings from data primitives to display primitives easily edited to change the way data are displayed. Figure 4.6 shows the *goes_sequence* data object from Figure 4.5 displayed according to four different sets of mappings. In the top-right window it is displayed according to the same seven mappings used in Figure 4.5, which are:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map variance to selector;
map texture to selector;
map image_region to selector;
map time to animation;
```

The display in the top-left window of Figure 4.6 can be generated by the following two changes to the above mappings:

```
map ir_radiance to color; /* red */
map vis_radiance to color; /* blue-green */
```

Notice that more than one data primitive can be mapped to *color* since it is a three-dimensional primitive. The user determines how *color* is factored into components using interactive color map icons like those shown in Figures 2.2 and 4.3.

Next, the display in the bottom-right window of Figure 4.6 can be generated by the following additional changes to the mappings:

```
map ir_radiance to selector;
map vis_radiance to color;
map time to z_axis;
```

Finally, the display in the bottom-left window of Figure 4.6 can be generated by the following changes to six of the seven mappings:

```
map earth_location to selector;
map ir_radiance to x_axis;
map vis_radiance to y_axis;
map variance to z_axis;
map texture to color;
map time to animation;
```

Actually, the VisAD system allows data objects to be displayed according to four different sets mappings simultaneously, and this was capability used to generate Figure 4.6.

Figure 4.6. A *goes_sequence* object displayed according to four different sets of mappings. The top-right is the same as Figure 4.5, the top-left maps *ir* (red) and *vis* (blue-green) to *color*, the bottom-right maps *ir* to *selector* and *time* to the *y* axis, and the bottom-left maps *ir*, *vis* and *variance* to the *x*, *y* and *z* axes, maps *texture* to *color*, and maps *lat_lon* to selector. (color original)

Flexibility in the ways that data are displayed can be useful for comparing data objects of different types, as illustrated by the following example. In 1963 E. N. Lorenz developed a set of differential equations that exhibit turbulence in a very simple two-dimensional atmosphere (Lorenz, 1963). Roland Stull of the Atmospheric and Oceanic Sciences Department of the University of Wisconsin-Madison teaches an Atmospheric Turbulence course and has applied the VisAD system to an algorithm that integrates Lorenz's equations in order to illustrate turbulence to students in his course. The data types defined for this algorithm are:

```
type atmos_location = real2d;
type temperature = real;
type stream_function = real;
type atmos = array [atmos_location] of
    structure {
        temperature;
        stream_function;
    }
type phase_x = real;
type phase_y = real;
type phase_z = real;
type time = real;
```

```
type phase_point =
    structure {
        phase_x;
        phase_y;
        phase_z;
    }
type phase_history = array [time] of phase_point;
```

The Lorenz equations describe temperature and air flow in a rectangular cell of a two-dimensional atmosphere. The algorithm integrates the Lorenz equations as a path through a three-dimensional phase space, recorded in a data object of type *phase_history*. This object is displayed in both the lower-left and upper-left windows in Figure 4.7. The lower-left window is defined by the mappings:

```
map atmos_location to selector;
map temperature to selector;
map stream_function to selector;
map phase_x to x_axis;
map phase_y to y_axis;
map phase_z to z_axis;
map time to selector;
```

The lower-left window shows two data objects displayed in different colors: red and blue-green (the system automatically picks a different solid color for displays of data objects that don't include any scalar values mapped to *color*). The *phase_history* object, displayed as a path of red points, winds chaotically between two lobes (this three-dimensional shape is called the Lorenz attractor). A data object of type *phase_point* is

also displayed in this window as a single blue-green point, marking the point on the phase space path corresponding to the rectangular cell of the two-dimensional atmosphere displayed in the right window in Figure 4.7. That window shows a data object of type *atmos* displayed using the mappings:

```
map atmos_location to xy_plane;
map temperature to color;
map stream_function to contour;
map phase_x to selector;
map phase_y to selector;
map phase_z to selector;
map time to selector;
```

The color field indicates *temperature*, where warm areas are red and cool areas are blue. The contours of the *stream_function* are parallel to air motion, and their spacing indicates wind speed. The direction of air flow can be inferred from the knowledge that warm air rises. As the program executes, this window shows the changing dynamics of the cell of atmosphere, and the lower-left window shows the motion of the corresponding phase space point. This animation makes it clear that the two lobes of the Lorenz attractor in phase space correspond to clockwise and counterclockwise rotation in the two-dimensional atmosphere cell.

The upper-left window in Figure 4.7 shows the *phase_history* object displayed using the mappings:

```
map atmos_location to selector;
map temperature to selector;
map stream_function to selector;
```

```
map phase_x to x_axis;
map phase_y to y_axis;
map phase_z to selector;
map time to z_axis;
```

In the upper-left window two dimensions of the winding path in phase space are plotted against *time*, illustrating the apparently random (that is, chaotic) temporal distribution of alternations between the two phase space lobes.

Figure 4.7. Three views of chaos. The right window shows temperatures and wind stream lines in a cell of a two-dimensional atmosphere. The bottom-left window shows the trajectory of atmospheric dynamics through a three-dimensional phase space. The top-left window shows this trajectory in two phase space dimensions versus time. (color original)

The third goal developed in Section 1.1 states that visualization techniques "*Enable users to interactively alter the ways data are viewed*." Achieving this goal depends not only on the ease with which users can control displays, but also on how quickly the system can generate displays. The transformation of data objects into physical displays is factored into the two mappings $D : U \rightarrow V$ and $RENDER : V \rightarrow V'$, where $V$ is a logical display model and $V'$ is a physical display model. Logical displays in $V$ are sets of tuples of display scalar values, and physical displays in $V'$ are two-dimensional arrays of colored pixels. The *RENDER* function can be computed quickly since it is essentially the traditional graphics pipeline whose operations are commonly implemented in hardware. Thus we have focused our optimizations on the function $D$.

The function $D$ is specified by a set of mappings from scalars to display scalars. Based on the embedding of data objects in the lattice $U$ described in Section 3.2, a data object $u$ is interpreted as a set of tuples of scalar values. Each tuple in $u$ is transformed to a tuple in $D(u)$ according to the mappings from scalars to display scalars. The VisAD implementation of $D$ exploits both parallel and vector techniques in order to achieve interactive response times. First, the tuples belonging to a data object can be processed independently and thus are partitioned among M processes which execute in parallel. (These execute in a shared memory model, which is common on modern workstations and relatively easy to port.) Second, the important branches in the algorithm for processing tuples depend on data types rather than data values. Thus large sets of tuples take the same path through the algorithm and can be processed in groups of N, allowing computations to be optimized in tight loops over vectors of values for entire groups. Typical values are M = 4 and N = 256. While such parallelization and vectorization techniques are not novel, they are quite effective in producing a fast implementation of the function $D$.

As discussed in Section 1.2.3, display objects in $V$ are inherently interactive. Users have the following interactive controls over the mapping $RENDER : V \rightarrow V'$:

1. Control over the projection from a three-dimensional space to a two-dimensional display screen (i.e., rotate, pan and zoom in three dimensions).

2. Control over time sequencing for scalars mapped to *animation*.

3. Control over color maps for scalars mapped to *color*.

4. Control over the iso-levels of scalars mapped to the $contour_i$ scalars.

5. Control over the selected sets of values for scalars mapped to the $selector_i$ scalars.

Users also have the following interactive controls over the mapping $D : U \rightarrow V$ and the selection of data objects:

1. Control over the way that data are displayed, by selecting, for each scalar, which display scalar it is mapped to.

2. Control over the mathematical mapping from scalar values to display scalar values. This is particularly useful for scalars mapped to spatial coordinates (i.e., $x$, $y$ and $z$) and to *color*.

3. Control over which data objects are displayed. (Note that multiple data objects can be displayed simultaneously. Ultimately, display objects in $V$ are transformed into lists of three-dimensional vectors and triangles for rendering, and multiple data objects are combined merely by merging their sets of vectors and triangles.)

A key to design of the VisAD system is that it treats the definition of scalar mappings (items 1 and 2 above) and the selection of data objects for display (item 3 above) like any other interactive display control. This is in contrast to the automated techniques of Mackinlay (Mackinlay, 1986), Robertson (Robertson, 1991), and Senay and Ignatius (Senay and Ignatius, 1991; Senay and Ignatius, 1994). They each solicited a set of visualization goals from the user, and then searched for a display design that satisfied these goals. The automated approach is motivated by the desire to minimize the user's effort to generate data displays. However, a set of scalar mappings is no more complex than a set of visualization goals. Furthermore, the scalar mappings control how data are displayed in a direct and intuitive way, whereas the way that a display-design algorithm interprets the user's visualization goals may not be intuitively obvious. By making control over scalar mappings interactive, we enable users to explore a variety of different ways of displaying the data objects in their algorithms. We believe that this interactive exploration is likely to be more useful than displays generated by intelligent display generation algorithms.

**4.3 Visualizing Scientific Computations**

In this chapter and in Chapter 2 we have developed a visualization system approach based on the five goals listed in Section 1.1. Our visualization approach can be directly applied to visualize executing programs because it is interactive and integrated with a scientific programming language. This enables scientists to perform visual experiments with their computations. Any data object defined in a scientific computation can be visualized, and can be visualized in a wide variety of different ways. This enables scientists to find high-level problems with their algorithms in the same way that interactive debuggers enable them to find low-level bugs. Just as with a debugger, scientists can control execution and set breakpoints. However, VisAD enables scientists to visualize large and complex data objects and thus to understand high-level problems in their algorithms. This visualization does not interfere with scientific algorithms, since there is no need to embed calls to display functions in programs, and it does not distract scientists, since they do not need to write display programs. Thus the VisAD system is easy to use.

At the simplest level, visualization serves to make data objects visible. We can think of visualization like a microscope - making an invisible world visible. Further, the visualization of data objects provides understanding of computational processes involving those data objects. For example, consider a bubble sort algorithm written in the VisAD programming language:

```
type time = real;
type temperature = real;
type temperature_series = array [time] of temperature;

sort(temperature_series temperatures; time n;)
{
    time outer, inner;
    temperature swap;

    /* A bubble sort is organized as two nested loops */
    for (outer=n; outer>1; outer=outer-1) {
        for (inner=1; inner<outer; inner=inner+1) {
            /* compare adjacent elements */
            if (temperatures[inner-1] > temperatures[inner]) {
                /* adjacent elements are out of order, so exchange them */
                swap = temperatures[inner];
                temperatures[inner] = temperatures[inner-1];
                temperatures[inner-1] = swap;
            }
        }
```

```
    }
}
```

Five data objects are declared in this program. The array being sorted is named *temperatures* and has type *temperature_series*. It is an array of *temperatures* indexed by *time*. The *inner* and *outer* loop indices into this array have type *time*, as does the size *n* of the array. The *swap* variable of type *temperature* is used to exchange elements of the array. Figure 4.8 shows this program running under VisAD, and four of these data objects are displayed in the window on the right (the size *n* is not displayed since it does not change as the program runs). They are displayed using the mappings:

```
map time to x_axis;
map temperature to y_axis;
```

Figure 4.8. Visualizing the computations of a bubble sort algorithm. (color original)

The text that defines these mappings can be seen in the small window at the top of the screen. The *temperatures* array is displayed as a graph (the set of white points) of *temperature* versus *time*. The *outer* index is displayed as a small green sphere on the lower horizontal axis. Note that the white points to the right of the green sphere are

sorted. The *inner* index is displayed as a small red sphere. It marks the horizontal position of the current maximum value bubbling up through the *temperatures* array. The small blue sphere on the left hand vertical axis depicts the *swap* variable. This display changes as the algorithm runs, providing a clear depiction of how the bubble sort works. This is sometimes called *algorithm animation* (Brown and Sedgewick, 1984). VisAD's displays are generally asynchronous with computations, but may be synchronized with calls to the built-in function *sync*.

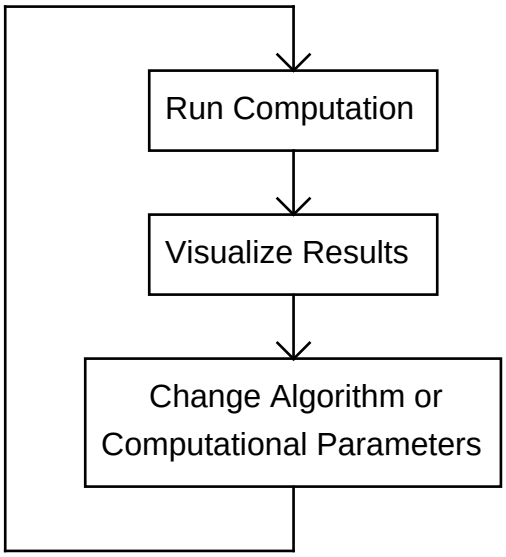


Figure 4.9. Visually experimenting with algorithms (this is a copy of Figure 1.3).

The ability to make computations visible can be used to find problems with algorithms, to experiment with different algorithms, and to tune algorithm parameters. Each of these places a slightly

different emphasis on the system-user feedback loop shown in Figure 4.9. The time around the feedback loop in Figure 4.9 may be less than a second when the user is tuning an algorithm, whereas minutes may be required for the user to edit a program to experiment with algorithm structure. Figure 4.10 illustrates the system-user feedback loop for finding the causes of problems with algorithms.

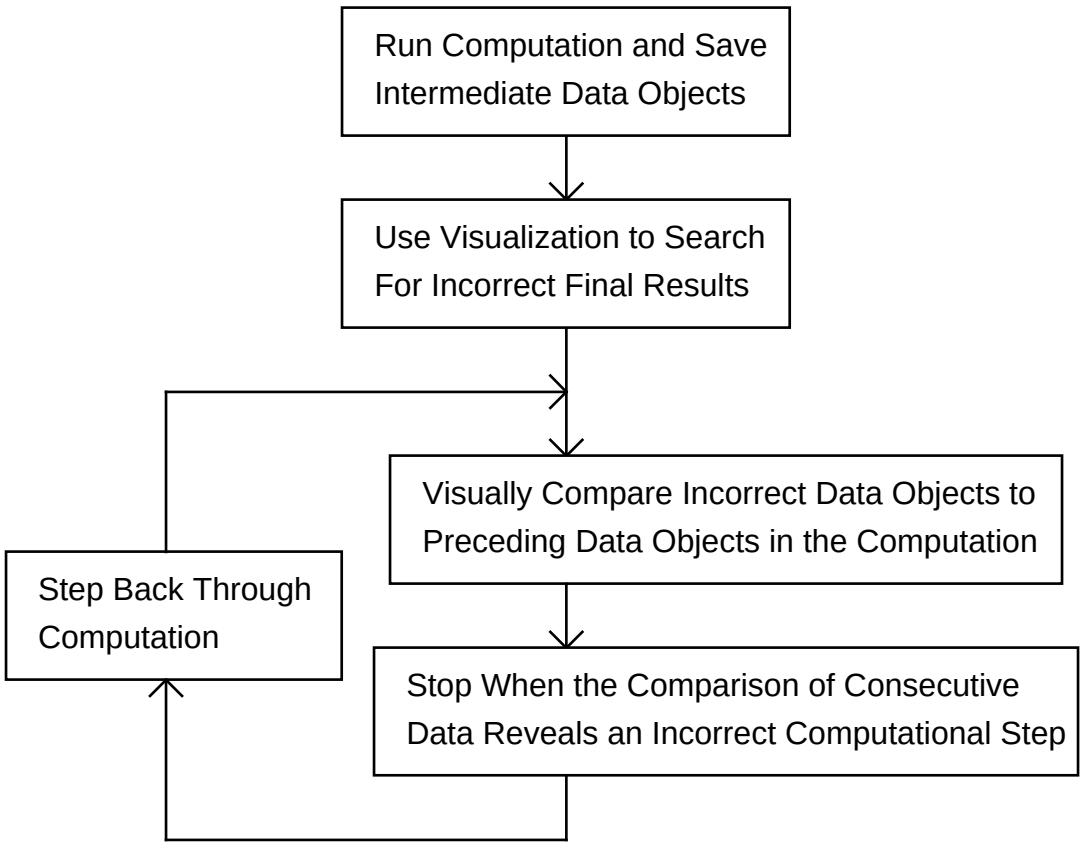


Figure 4.10. Visually tracing back to the causes of computational errors.

An algorithm for detecting clouds in GOES images provides a good example of using VisAD for finding high-level problems with algorithms. Some of the data types defined for this algorithm are:

```
type earth_location = real2d;
```

```
type ir_radiance = real;
type vis_radiance = real;
type ir_image = array [earth_location] of ir_radiance;
type image =
    array [earth_location] of
        structure {
            ir_radiance;
            vis_radiance;
        }

type image_region = integer;
type ir_image_partition = array [image_region] of ir_image;
type image_partition = array [image_region] of image;

type count = integer;
type histogram = array [ir_radiance] of count;
```

The input to the algorithm is a data object of type *image_partition*; Figure 4.11 shows an input data object displayed using the mappings:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map image_region to selector;
map count to selector;
```

The algorithm partitions images into rectangular regions and processes each region independently. Two regions are selected in Figure 4.11. The small bump straddling the two image regions on the left is a cloud. The output of the algorithm is another data object of type *image_partition* where the values of *non-cloud* pixels are set to *missing*. Figure 4.12 shows the output generated from Figure 4.11 with the same two image regions selected. The small cloud in Figure 4.11 is not seen, so its pixels have been marked as *non-cloud*. This is clearly an error.

We can find the cause of this error by visually comparing data objects at different stages of the algorithm's computations. Figure 4.13 shows three data objects of type *ir_image_partition*. Each data object is displayed in a different color: white, red and green. The white *ir_image_partition* data object includes all pixels but is overlaid by the red and green data objects. The algorithm selects *cloud* pixels as subsets of the non-*missing* pixels in the red and green *ir_image_partition* data objects. Since the bump on the left is white rather than red or green, the error in the computation must have been made before the calculation of the *ir_image_partition* data objects colored red and green. Pixels are selected for these two data objects according to whether their *ir_radiance* values lie in clusters of certain histograms. Three data objects of type *histogram* are shown in Figure 4.14 displayed using the mappings:

```
map earth_location to selector;
map ir_radiance to x_axis;
map vis_radiance to selector;
map image_region to selector;
map count to y_axis;
```

The white *histogram* data object includes all *ir_radiance* values but again these are overlaid by the red and green *histogram* data objects. The red and green *histogram* objects include only those *ir_radiance* values

lying in clusters. The ranges of *ir_radiance* defined by these red and green *histogram* objects are used to select pixels for the red and green *ir_image_partition* objects seen in Figure 4.13. The white *histogram* object is generated from the population of pixels within one image region pictured in Figure 4.11. Thus Figure 4.14 makes it clear that the little bump cloud on the left in Figure 4.11 is not large enough to generate a detectable cluster in the *histogram* object in Figure 4.14, possibly because this population is evenly divided between two image regions. Thus we have found the ultimate cause of the error in this computation.

Figure 4.11. A close-up view of two regions of a *goes_sequence* object displayed as a terrain. Note the small bump, undoubtedly a cloud, straddling the regions on the left. (color original)

Figure 4.12. A close-up view restricted to the "cloudy" pixels in two regions of a *goes_sequence* object displayed as a terrain. The small cloud seen on the left in Figure 4.11 is not detected as a cloud in this figure. (color original)

Figure 4.13. Three *goes_sequence* objects displayed as terrains, with *ir* radiance mapped to terrain height (the *y* axis) but without *vis* radiance mapped to *color*. (color original)

Figure 4.14. Three *histogram* objects displayed as graphs. The algorithm judges red and green points to lie in clusters - these define ranges of ir_radiance values that define the red and green pixels seen in Figure 4.13. (color original)

An algorithm for detecting valid observations of interstellar X-rays provides a good example of using the VisAD system for experimenting with algorithms. The Diffuse X-ray Spectrometer sensed several million distinct events during its January 1993 flight on the Space Shuttle (Sanders et al., 1993), each potentially an observation of an X-ray emanating from interstellar gas. However, most of these events were not valid, so Wilton Sanders and Richard Edgar of the University of Wisconsin-Madison needed to develop an algorithm for detecting valid events. Some of the data types defined for this algorithm are:

```
type time = real;
type wavelength = real;
type longitude = real;
type pulse_height = real;
type position_bin = real;
type goodness_of_fit = real;
type occulted flag = int;
type xray_event =
    structure {
        time;
        wavelength;
        longitude;
        pulse_height;
        position_bin;
        goodness_of_fit;
        occulted flag;
    }
type event_number = int;
```

```
type count = int;
type count2 = int;
type event_list = array [event_number] of xray_event;
type histogram_2d = array [longitude] of
    array [wavelength] of
        structure {
            count;
            count2;
        }
```

Figure 4.4 shows a data object of type *event_list* displayed using the following scalar mappings:

```
map longitude to x_axis;
map wavelength to y_axis;
map time to z_axis;
map pulse_height to color;
map position_bin to selector;
map goodness_of_fit to selector;
map occulted_flag to selector;
map event_number to selector;
map count to selector;
map count2 to selector;
```

In Figure 4.4 each X-ray event is displayed as a colored dot. Slider icons in the upper-right corner were used to select a range of values for each event field mapped to *selector*, and only those events whose

field values fall in the selected ranges are displayed. This provides an easy way to experiment with event selection criteria. During the development of the event selection algorithm, a large number of different sets of mappings were defined in order to experiment with selections based on different combinations of event fields and thus to help Sanders and Edgar to understand the mechanisms that produced invalid events.

Figure 4.15 shows a data object of type *histogram_2d* in a frame of reference defined by:

map longitude to y_axis;

map wavelength to x_axis;

map count to z_axis;

map count2 to color;

map time to selector;

map pulse_height to selector;

map position_bin to selector;

map goodness_of_fit to selector;

map occulted_flag to selector;

map event_number to selector;

This *histogram_2d* object contains frequency *counts* of X-ray events in bins of *wavelength* and *longitude*. The *count2* values are redundant with the *count* values. Both are included so that one may be mapped to the *x_axis* and the other mapped to *color*. The display of this object is seen from an oblique angle so that it appears as a series of short colored graphs, one for each *longitude* bin. Each colored graph shows *count* as a function of *wavelength*, and thus provides a spectrum of X-rays in a *longitude* bin. Some types of spurious events showed up as spikes in one-dimensional and two-dimensional histograms (i.e., these spurious events had similar values in one or two event fields) and this provided insight into how to remove

these events. Displays of histograms of populations of events selected by various algorithms provided insight into what further selection criteria were needed.

Figure 4.15. A two-dimensional histogram of X-ray events, with 10 degree *longitude* bins along the vertical axis and small *wavelength* bins along the horizontal axis. Viewed from an oblique angle, this object appears as a series of short graphs showing the X-ray spectrum in each *longitude* bin. (color original)

An algorithm for detecting cumulus clouds in GOES images provides a good example of using VisAD for tuning parameters of algorithms. Robert Rabin (Rabin et. al., 1990) of the National Severe Storms Laboratory, working at the University of Wisconsin-Madison, developed an algorithm for detecting cumulus clouds based on infrared radiance, visible radiance, and contrast (a quantity derived from visible radiance). Some of the data types defined for this algorithm are:

```
type earth_location = real2d;
type ir_radiance = real;
type vis_radiance = real;
type contrast = real;
type ir_image = array [earth_location] of ir_radiance;
type vis_image = array [earth_location] of vis_radiance;
type contrast_image = array [earth_location] of contrast;
```

Separate selection criteria were defined for each of *ir_radiance*, *vis_radiance* and *contrast*, and Figure 4.16 shows data objects of types *ir_image*, *vis_image* and *contrast_image* displayed according to the mappings:

```
map earth_location to xy_plane;
map ir_radiance to color;
map vis_radiance to color;
map contrast to color;
```

The visualization in Figure 4.16 was used to tune the cumulus cloud selection algorithm. In the displayed data objects, *ir_radiance*, *vis_radiance* and *contrast* values that do not satisfy the selection criteria have

been set to *missing* and are invisible. The color maps have been adjusted so that any non-*missing* *ir_radiance* is displayed as red, any non-*missing vis_radiance* is displayed as blue, and any non-*missing* *contrast* is displayed as green. Thus each pixel in the image takes one of eight colors, indicating the two × two × two combinations of selections by these three criteria. Only those pixels colored white are selected by all three criteria as cumulus cloud pixels (because white = red + blue + green). We were able to interactively adjust these selection criteria using slider icons (similar to those seen in Figure 2.2), to see how the selection of cumulus cloud pixels changed in response to those adjustments, and to understand from their colors which criteria cause pixels to fail to be selected.

Figure 4.16. Visualizing the three criteria used to select cumulus clouds. Pixels satisfying the infrared criterion are colored red, pixels satisfying the visible criterion are colored blue, and pixels satisfying the contrast criterion are colored green. Combinations of these colors indicate pixels satisfying more than one of the criteria. Pixels selected as cumulus clouds are colored white. (color original)

**4.4 System Organization**

We have described our system design in stages, explaining how it is motivated by the goals of Section 1.1 and the principles of Chapter 3. In this section we present on overview of the way the system integrates scientific data, computation and display.

Figure 4.17 illustrates the overall organization of the VisAD system. The system's computing components occupy the left side of this diagram and its display components occupy the right side, linked only through the data component. Furthermore, information from the system's display component does not flow into its data or computation components, emphasizing that the system's display functions do not intrude on a user's science programs.

Figure 4.17 also shows how the user interface is divided into five different components, two relating to computation and three relating to display. The computational user interface divides into

1. An editor for defining and editing programs. This editor is also used for defining data types, since they are part of the text of programs.

2. Controls over program execution. These include controls for starting and stopping execution, for executing single program statements, and for setting values on *slider* icons that are read by calls to the intrinsic function *slider* (as illustrated in Figure 2.2). Execution breakpoints are set (and cleared) by graphically picking program statements in the program text editor, and are indicated by highlighting statements in the program text.

The display user interface divides into

1. An editor for defining mappings from data scalar types to display scalar types. These mappings control the transformation of data into logical displays. Data objects are selected (and de-selected)

for display by graphically picking their names in the program text editor, and are indicated by highlighting their names in the program text.

2. Controls over the rendering transformation from logical to physical displays (i.e., the *RENDER* function). These include controls over animation, over color maps, over selecting ranges of values (for scalars mapped to selector), over contour levels, and over the projection from three to two dimensions (i.e., rotate, pan and zoom).

3. Physical displays visible to the user.

Note that there are two deviations from the clean separation of user interface functions and that both involve graphically picking and highlighting text segments in the program text editor. Specifically, program statements are selected as breakpoints and data objects are selected for display in this way. While we have not used a graphical user interface for designing the data and control flow of programs in our system, we have adopted these two graphical picking functions because they can be naturally integrated with a text based programming interface.

The overall system organization shown in Figure 4.17 is consistent with a variety of possible future system extensions. In particular, the display model could be extended by adding more display scalars, and a module could be added to design default scalar mappings appropriate for various aggregate data types. These would require changes to the system source code but would not be particularly difficult. However, based on the goals developed in Section 1.1, the system is designed to make it easy for users to define their own data types, displays and programs. By building such generality into our system's user interface we seek to reduce the need for changes to the system itself.

The system diagram shows the connection to external functions through a socket interface. This allows VisAD programs to link to functions written in C or Fortran and possibly running remotely (i.e., on

another computer connected via a network). The ability to define such links to compiled functions is important for the robustness of scientific computing environments. Mature scientific programming environments typically include hundreds of user-defined functions.

The ways that scalar values can sample one-, two- and three-dimensional real values is also extensible. The system supports a variety of built-in samplings for two-dimensional map projections and for geographically registering common meteorological satellites. While it is easy to define new built-in sampling functions, the system also provides a way for users to define one-, two- and three-dimensional samplings within the programming language.

Our system design defines a few simple capabilities that users can flexibly combine to produce complex applications. Users can define complex data types as hierarchies of scalars, tuples and arrays, they can express complex metadata by samplings and missing data, they can define complex algorithms in a general scientific programming language, and they can define a complete set of data displays by mappings from data primitives to display primitives.

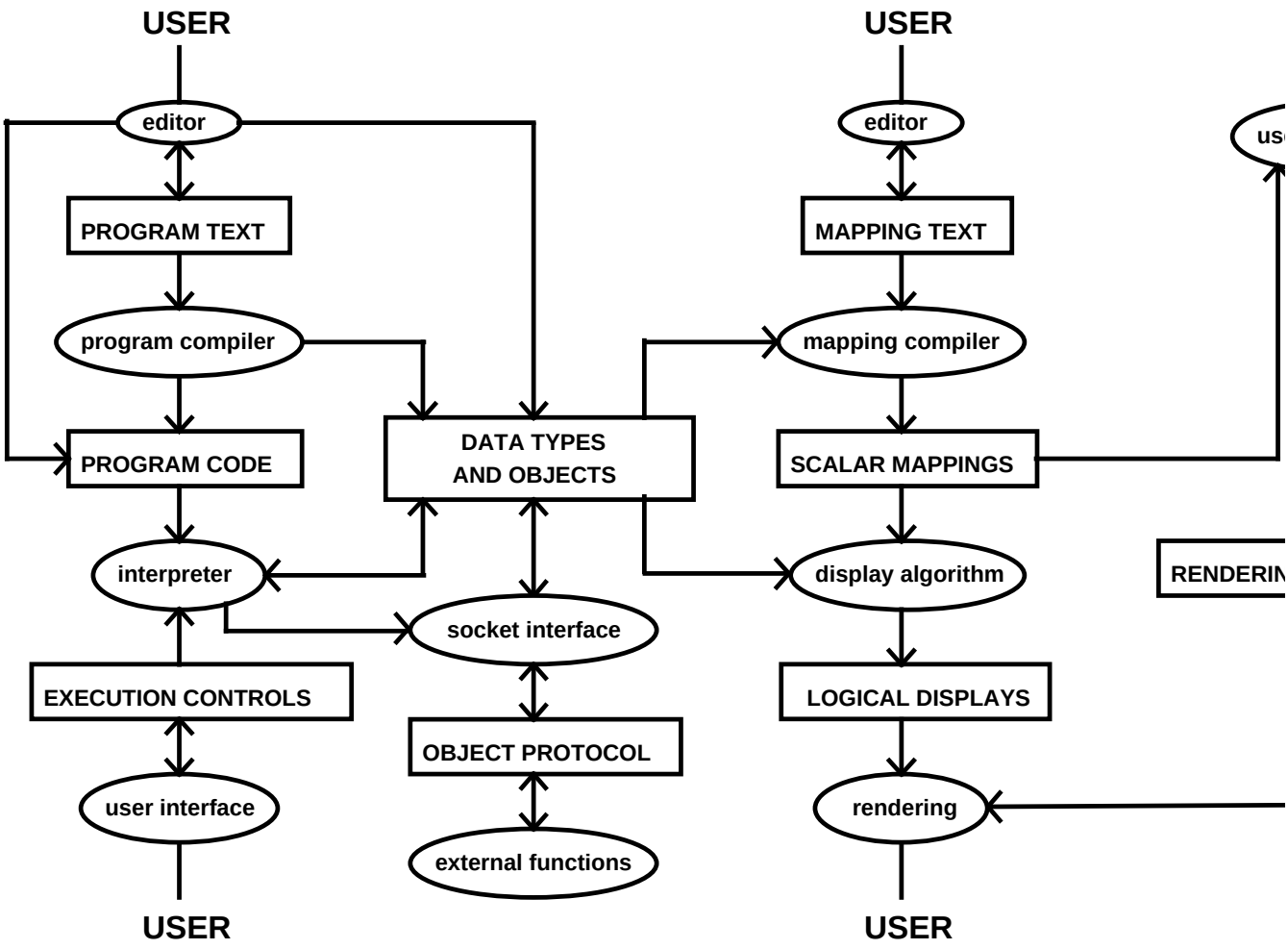


Figure 4.17. VisAD system organization.

# Chapter 5

## Applying the Lattice Model to Recursive Data Type Definitions

In Section 3.1 we showed that a function $D : U \rightarrow V$ satisfying the expressiveness conditions must be a lattice isomorphism. In Section 3.4 we applied this result to specific lattice structures defined for scientific data and display models. However, this result can be applied to any complete lattices and it is

natural to apply this result to other lattice structures for data and display models. The motive for new lattice structures must be new data models, since display models are themselves motivated by the need to visualize data. The data model defined in Section 3.2 includes tuples and arrays as ways of aggregating data, but does not include linked list structures defined in terms of pointers. In this chapter we describe several issues in extending our lattice theory to data types appropriate for handling objects with pointers.

**5.1 Recursive Data Types Definitions**

The denotational semantics of programming languages provides techniques for defining ordered sets whose members are the values of programming language expressions (Gunter and Scott, 1990; Schmidt, 1986; Scott, 1971; Scott, 1982). An important topic of denotational semantics is the study of *recursive domain equations*, which define *cpos* recursively (*cpo* is defined in Appendix A).

Consider the following example of a recursive domain equation from (Schmidt, 1986). A data type for a binary tree may be defined by

(5.1) $Bintree = (Data + (Data \times Bintree \times Bintree))_{\perp}$

Here "+", "×" and "$(.)_{\perp}$" are type construction operators similar to the tuple and array operators defined in Section 3.2.3. The "+" operator denotes a type that is a choice between two other types (this is similar to "union" in the C language), "×" denotes a type that is a cross product of other types (this is essentially the same as our tuple operator, so that ($Data \times Bintree \times Bintree$) is a 3-tuple), and the "⊥" subscript indicates a type that adds a new least element, ⊥, to the set of values of another type. Eq. (5.1) defines a data type called *Bintree*, and says that a *Bintree* object is either ⊥, a data object of type *Data*, or a 3-tuple consisting of a data object of type *Data* and two data objects of type *Bintree*. Intuitively, a data object of type *Bintree* is either missing, a leaf node with a data value, or a non-leaf node with a data value and two child nodes.

The obvious way to implement binary trees is to define a record or structure for a node of the tree, and to include two pointers to other nodes in that record or structure. In general, self references in recursive type definitions are implemented using pointers.

**5.2 The Inverse Limit Construction**

The equality in a recursive domain equation is really an isomorphism. As explained by Schmidt, these equation may be solved by the *inverse limit construction*. For the *Bintree* example this construction starts with $Bintree_0 = \{\perp\}$, and then applies Eq. (5.1) repeatedly to get:

(5.2) $Bintree_1 = (Data + (Data \times Bintree_0 \times Bintree_0))_\perp$

$Bintree_2 = (Data + (Data \times Bintree_1 \times Bintree_1))_\perp$

etc.

The construction also specifies a retraction pair $(g_i, f_i)$:$Bintree_i \leftrightarrow Bintree_{i+1}$ for all $i$, such that $g_i$ embeds $Bintree_i$ into $Bintree_{i+1}$ and $f_i$ projects $Bintree_{i+1}$ onto $Bintree_i$ (*retraction pair* is defined in Appendix A). Then *Bintree* is the set of all infinite tuples of the form $(t_0, t_1, t_2, ...)$ such that $t_i = f_i(t_{i+1})$ for all $i$. It can be shown that *Bintree* is isomorphic with $(Data + (Data \times Bintree \times Bintree))_\perp$, and thus "solves" the recursive domain equation. The order relation on the infinite tuples in *Bintree* is defined element-wise, just like the order relation on finite tuples defined in Section 3.2, and *Bintree* is a *cpo*. We note that the inverse limit construction can also be applied to solve sets of simultaneous domain equations.

The *cpos* defined by the inverse limit construction are generally not lattices. In order to apply Prop. C.3 to these *cpos* they must be embedded in complete lattices. However, the Dedekind-MacNeille completion shows that for any partially ordered set *A*, there is always a complete lattice *U* such that there is an order embedding of *A* into *U* (Davey and Priestley, 1990).

The set of *Bintree* objects defined by the inverse limit construction includes infinite trees. Denotational semantics must include values for non-terminating computations, and non-terminating computations may produce infinite trees as their values. Since our result that display functions are lattice isomorphisms depends on the assumption that data and display lattices are complete, it is likely that any data lattice we define that includes solutions of recursive domain equations must include infinite data objects.

The inverse limit construction defines the set of data objects of a particular data type that solves a particular recursive domain equation. However, our approach in Section 3.2 was to define a large lattice that contained data objects of many different data types. It would be useful to continue this approach, by defining a lattice that includes all data types that can be constructed from scalar types as tuples, arrays, and solutions of recursive domain equations. This is the subject of Section 5.3.

**5.3 Universal Domains**

A fundamental result of the theory of ordered sets is the *fixed point theorem*, which says that, for any *cpo D* and any continuous function $f{:}D \rightarrow D$, there is $fix(f) \in D$ such that $f(fix(f)) = fix(f)$ (that is, $fix(f)$ is a fixed point of $f$) and such that $fix(f)$ is less than any other fixed point of $f$.

Scott developed an elegant way to solve recursive domain equations by applying the fixed point theorem (Scott, 1976; Gunter and Scott, 1990). The idea is that the solution of a recursive domain equation is just a fixed point of a function that operates on *cpos*. Scott first defined a *universal domain U* and a set *W* of retracts of *U*. *W* may be the set of all retracts on *U*, the set of projections, the set of finitary projections, the set of closures, or the set of finitary closures (these terms are defined in Appendix A). Then he showed that a set *OP* of type construction operators (these operators build *cpo*'s from other *cpo*'s) can be represented by continuous functions over *W*, in the sense that for $op \in OP$ there is a continuous function *f* on *W* that makes the diagram in Figure 5.1 commute.

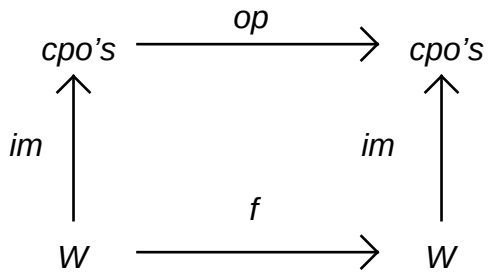


Figure 5.1. The type construction operator *op* is represented by function *f*.

Note that $im(w) = \{w(u) \mid u \in U\}$. For unary $op \in OP$ this is $im(f(w)) = op(im(w))$. Similar commuting expressions hold for multiary operators in *OP*. Then, for any recursive domain equation $D = O(D)$ where $O$ is composed from operators in *OP*, there is a continuous function $F{:}W \to W$ that represents $O$. By the fixed point theorem, $F$ will have a least fixed point $fix(F)$, and $O(im(fix(F))) = im(F(fix(F))) = im(fix(F))$, so $im(fix(F))$ is a *cpo* satisfying the recursive domain equation $D = O(D)$. The solution of any domain equation (or any set of simultaneous domain equations) involving the type construction operators in *OP* will be a *cpo* that is a subset of the universal domain *U*. Thus this approach is similar to the way that we embedded data types in a complete lattice (coincidentally denoted by *U*) in Section 3.2.3. Universal domains and representations have been defined for sets *OP* that include most of the type constructors used in denotational semantics, including "+", "×", "→", and "$(.)_\perp$".

A common example of a universal domain is the complete lattice *POWER*(**N**), which is just the set of all subsets of the natural numbers **N**. In general, the embeddings of data types into universal domains, as defined by papers in denotational semantics, are not suitable for our display theory. For example, a single integer (that is, an object of type **N**), and a function from the integers to the integers (that is, an object of type $\mathbf{N} \to \mathbf{N}$), may be embedded to the same member of *POWER*(**N**). A display function applied to the lattice *POWER*(**N**), with these embeddings, would produce the same display for the integer and the function from the integers to integers. Such displays cannot effectively communicate information about data objects, so other embeddings of types into universal domains must be developed.

**5.4 Display of Recursively Defined Data Types**

Since the goal of visualization is to communicate information about data to people, an extension of our theory must focus on the data lattice $U$. However, since a display function $D$ is a lattice isomorphism of $U$ onto a sublattice $V$, we should be able to say something about the structure of $V$. If a subset $A \subseteq U$ is the solution of a recursive domain equation (that is, $A$ is the set of data objects of some recursively defined data type), then $D(A) \subseteq V$ is isomorphic to $A$ and must itself be the solution of the recursive domain equation.

For example, if the set $A$ is the solution of Eq. (5.1) for *Bintree*, then the set $D(A)$ must also solve this equation. The isomorphism $D$ provides a definition of the operators "+", "$\times$" and "$(.)_{\perp}$" in $D(A)$ and thus also defines a relation between "tree" objects and their "subtree" objects in $D(A)$. The isomorphism does not tell us how to interpret these operators and relations in a graphical display, but it does tell us that such a logical structure exists. Given the complexity of this structure, it is seems likely that display objects in $D(A)$ will be interpreted using some graphical equivalent of the pointers that are used to implement data objects in $A$.

Two graphical analogs of pointers are commonly used in displays:

1. Diagrams. Here icons represent nodes in data objects, and lines between icons represent pointers.

2. Hypertext links. Here the visible contents of a display screen represents one or more nodes in a data object, and an icon embedded in that display screen represents an interactive link to another node or set of nodes. That is, if the user selects the icon (say by a mouse point and click), new display screen contents appear depicting the display object (and possibly other objects) referenced by the icon.

In order to extend our display theory to data types defined with recursive domain equations, we need to extend our display lattice *V* to include these graphical interpretations of pointers. A difficult open problem is to find a way to do this that produces a display lattice complex enough to be isomorphic to a universal domain as described in Section 5.3.

# Chapter 6

# Conclusions

This thesis was motivated by physical scientists' need for visualization techniques that can be applied to the data of a wide variety of scientific applications, that can produce a wide variety of different visualizations of data appropriate for different needs, that are as interactive as possible, that require minimal effort by scientists to use, and that can be integrated with a scientific programming environment. Our approach has been to achieve generality and simplicity by developing appropriate abstractions for scientific data, for scientific displays, and for the visualization mapping from data to displays.

## 6.1 Main Contributions and Limitations

The main contributions of this thesis can be summarized as follows:

1. Development of a system for scientific visualization that enables a wide variety of visual experiments with scientific computations. This system integrates visualization with a scientific programming language that can be used to express scientific computations. This programming language supports a wide variety of scientific data types and integrates common forms of scientific metadata into the computational and display semantics of data. Any data object defined in a program in this language can be visualized in a wide variety of ways during and after program execution. Displays are controlled by a set of simple mappings rather than program logic. These mappings are independent of data type and separate from a user's scientific programs, which is a clear distinction from previous visualization systems that require scientists to embed calls to visualization functions in their programs. Furthermore, computation and visualization are highly interactive. In particular, the selection of data objects for display and the controls for how they are displayed are treated like any

other interactive display control (e.g., interactive rotation). Previous visualization systems require a user to alter his program in order to make such changes. The generality, integration, interactivity and ease-of-use of this system all enhance the user's ability to perform visual experiments with their algorithms.

2. Introduction of a systematic approach to analyzing visualization based on lattices. We defined a set *U* of data objects and a set *V* of displays and showed how a lattice structure on *U* and *V* expresses a fundamental property of scientific data and displays (namely that they are approximations to the physical world). The visualization repertoire of a system can be defined as a set of mappings of the form $D : U \rightarrow V$. It is common to define a system's visualization repertoire by enumerating such a set of functions. However, an enumerated repertoire is justified only by the tastes and experience of the people who decide what functions to include in the set. In contrast, we interpreted certain well-known expressiveness conditions on the visualization mapping $D : U \rightarrow V$ in terms of a lattice structure, and defined a visualization repertoire as the set of functions that satisfy those conditions. Such a repertoire is justified by the generality of the expressiveness conditions. We showed that visualization mappings satisfy these conditions if and only if they are lattice isomorphisms. Lattice structures can be defined for a wide variety of data and display models, so this result can be applied to analyze visualization repertoires in a wide variety of situations.

3. Demonstration of a specific lattice structure that unifies data objects of many different scientific types in a data model *U*, and demonstration that the same lattice structure can express interactive, animated, three-dimensional displays in a display model *V*. These models integrate certain kinds of scientific metadata into the computational and display semantics of data. In the context of these scientific data and display models, we showed that the expressiveness conditions imply that mappings of data aggregates to display aggregates can always be factored into mappings of data

primitives to display primitives. We showed that our display mappings are complete, in the sense that we characterized all mappings satisfying the expressiveness conditions.

These results have several limitations. Foremost, they do not include data objects with pointers. Thus our visualization techniques are not applicable to the data objects of general programming languages. This thesis developed a single lattice-structured scientific data model in which real numbers are approximated by intervals and functions are approximated by finite sets of samples of their values. However, there are other ways to approximate numbers and functions based on Eq. (3.2) and these may serve as the basis for other lattice-structured models for scientific data. The display model developed in this thesis models the ways that computers generate displays, but does not model the ways that people perceive displays. Finally, this thesis only considered conditions on visualization mappings based on lattice structures, and did not consider conditions based on other kinds of mathematical structures.

**6.2 Future Directions**

The work presented in this thesis can be extended to other lattice-structured models for data and displays, and to analytic conditions on visualization functions based on types of mathematical structures other than lattices. Specific future directions include:

1. Extend the VisAD system's display model to include more display scalars, such as transparency, reflectivity and flow vectors. These would be interpreted by including volume and flow rendering techniques in the mapping $RENDER : V \rightarrow V'$.

2. Extending the VisAD system to supply default mappings for controlling the displays of data objects. This could be accomplished by integrating VisAD with others' work on automating the design of displays (Robertson, 1991; Senay and Ignatius, 1991; Senay and Ignatius, 1994; Beshers and Feiner, 1992).

3. Extending the lattice results to data objects with pointers (i.e., data objects of recursively-defined data types). In Chapter 3 we showed how to embed scientific data objects of many different data types in a lattice. In Chapter 5 we showed how this might be extended by describing Scott's technique for embedding data objects of many different recursively-defined data types in a lattice. We also described graphical analogs of data objects with pointers. However, we described why Scott's embeddings are not suitable for visualization. Thus, finding ways to extend Scott's embeddings to a form suitable for visualization is an important next step. This would enable us to extend the VisAD system to a general programming language rather than a scientific programming language.

4. Defining lattice structures based on forms of approximations to numbers and functions other than intervals and finite samplings. Whenever data objects can be identified with sets of mathematical objects we can apply Eq. (3.2) (i.e., $u \leq u' \Leftrightarrow math(u') \subseteq math(u)$) to define a lattice structure on a data model. For example, functions may be approximated by finite sets of Fourier coefficients rather than finite sets of function values.

5. The analytic approach has great potential for making the study of visualization more rigorous and systematic. It is difficult to explicitly identify all of the assumptions of a synthetic approach to visualization, whereas assumptions must be explicit in an analytic approach. Analytic conditions on visualization mappings must be based on some mathematical structures defined on data and display models. In this thesis we have explored the consequences of a single set of conditions defined in terms of lattice structures. However, the full potential of the analytic approach can only be realized by exploring a much wider set of conditions based on a variety of mathematical structures. Data and display models may also have topological structures, metric structures, symmetry structures, structures based on arithmetic operations, and type hierarchy structures. Each of these kinds of structures can be used to define conditions on visualization mappings. Such conditions may be able to express a wide range of visualization goals, and mathematical analysis of visualization mappings satisfying various conditions may provide a rigorous foundation for visualization.

6. Defining structures on display models that express principles of human perception. For example, a metric can be defined for the perceived distance between displays (as measured by psychology experiments or predicted by psychological models). Alternatively, perception of displays may be invariant to certain operations (e.g., time translation or spatial translation), which may be expressed by defining symmetry groups on sets of displays.

# Appendix A

## Definitions for Ordered Sets

The appendices contain all the formal definitions, propositions and proofs for developing a model of the display process based on lattices. Here we list some basic definitions from the theory of ordered sets.

**Def.** A *partially ordered set* (*poset*) is a set $D$ with a binary relation $\leq$ on $D$ such that, $\forall x, y, z \in D$

(a) $x \leq x$ "reflexive"

(b) $x \leq y \,\&\, y \leq x \Rightarrow x = y$ "anti-symmetric"

(c) $x \leq y \,\&\, y \leq z \Rightarrow x \leq z$ "transitive"

**Def.** An *upper bound* for a set $M \subseteq D$ is an element $x \in D$ such that $\forall y \in M.\ y \leq x$.

**Def.** The *least upper bound* of a set $M \subseteq D$, if it exists, is an upper bound $x$ for $M$ such that if $y$ is another upper bound for $M$, then $x \leq y$. The least upper bound of $M$ is denoted *sup* $M$ or $\bigvee M$. *sup*$\{x,y\}$ is written $x \vee y$.

**Def.** A *lower bound* for a set $M \subseteq D$ is an element $x \in D$ such that $\forall y \in M.\ x \leq y$.

**Def.** The *greatest lower bound* of a set $M \subseteq D$, if it exists, is a lower bound $x$ for $M$ such that if $y$ is another lower bound for $M$, then $y \leq x$. The greatest lower bound of $M$ is denoted *inf* $M$ or $\bigwedge M$. *inf*$\{x,y\}$ is written $x \wedge y$.

**Def.** A subset $M \subseteq D$ is a *down set* if $\forall x \in M. \forall y \in D.\ y \leq x \Rightarrow y \in M$. Given $M \subseteq D$, define $\downarrow M = \{y \in D \mid \exists x \in M.\ y \leq x\}$, and given $x \in D$, define $\downarrow x = \{y \in D \mid y \leq x\}$.

**Def.** A subset $M \subseteq D$ is an *up set* if $\forall x \in M.\ \forall y \in D.\ x \leq y \Rightarrow y \in M$. Given $M \subseteq D$, define $\uparrow M = \{y \in D \mid \exists x \in M.\ x \leq y\}$, and given $x \in D$, define $\uparrow x = \{y \in D \mid x \leq y\}$.

**Def.** A subset $M \subseteq D$ is a *chain* if, for all $x, y \in M$, either $y \leq x$ or $x \leq y$.

**Def.** A subset $M \subseteq D$ is *directed* if, for every finite subset $A \subseteq M$, there is an $x \in M$ such that $\forall y \in A.\ y \leq x$.

**Def.** A *poset* $D$ is *complete* (and called a *cpo*) if every directed subset $M \subseteq D$ has a least upper bound $\bigvee M$ and if there is a least element $\perp \in D$ (that is, $\forall y \in D.\ \perp \leq y$).

**Def.** If $D$ and $E$ are *posets*, we use the notation $(D \to E)$ to denote the set of all functions from $D$ to $E$.

**Def.** If $D$ and $E$ are *posets*, a function $f{:}D \to E$ is *strict* if $f(\perp) = \perp$.

**Def.** If $D$ and $E$ are *posets*, a function $f{:}D \to E$ is *monotone* if $\forall x, y \in D.\ x \leq y \Rightarrow f(x) \leq f(y)$. We use the notation $MON(D \to E)$ to denote the set of all monotone functions from $D$ to $E$.

**Def.** If $D$ and $E$ are *posets*, a function $f{:}D \to E$ is an *order embedding* if $\forall x, y \in D.\ x \leq y \Leftrightarrow f(x) \leq f(y)$.

**Def.** Given *posets* $D$ and $E$, a function $f{:}D \to E$, and a set $M \subseteq D$, we use the notation $f(M)$ to denote $\{f(d) \mid d \in M\}$.

**Def.** If $D$ and $E$ are *cpos*, a function $f{:}D \to E$ is *continuous* if it is monotone and if $f(\bigvee M) = \bigvee f(M)$ for directed $M \subseteq D$.

**Def.** If $D$ is a *cpo*, then $x \in D$ is *compact* if, for all directed $M \subseteq D$, $x \leq \bigvee M \Rightarrow \exists y \in M.\ x \leq y$.

**Def.** A *cpo* $D$ is *algebraic* if for all $x \in D$, $M = \{y \in D \mid y \leq x \ \&\ y \text{ compact}\}$ is directed and $x = \bigvee M$.

**Def.** A *cpo* $D$ is a *domain* if $D$ is algebraic and if $D$ contains a countable number of compact elements.

Most of the ordered sets used in programming language semantics are domains.

**Def.** A *poset* $D$ is a *lattice* if for all $x, y \in D$, $x \vee y$ and $x \wedge y$ exist in $D$.

**Def.** A *poset* $D$ is a *complete lattice* if for all $M \subseteq D$, $\bigvee M$ and $\bigwedge M$ exist in $D$.

**Def.** If $D$ and $E$ are lattices, a function $f:D\to E$ is a *lattice homomorphism* if for all $x, y \in D$, $f(x \wedge y) = f(x) \wedge f(y)$ and $f(x \vee y) = f(x) \vee f(y)$. If $f:D\to E$ is also a bijection then it is a *lattice isomorphism*.

**Def.** A binary relation $\equiv$ on a set D is an *equivalence relation* if $\forall x, y, z \in D$

(a) $x \equiv x$ "reflexive"

(b) $x \equiv y \Leftrightarrow y \equiv x$ "symmetric"

(c) $x \equiv y \ \& \ y \equiv z \Rightarrow x \equiv z$ "transitive"

**Def.** $id_D$ denotes the identity function on $D$. Given a function $f:D\to D$, $im(f) = \{f(d) \mid d \in D\}$.

**Def.** If $D$ is a *cpo*, a continuous function $f:D\to D$ is a *retraction* of $D$ if $f = f \circ f$. A retraction $f:D\to D$ is a *projection* if $f \leq id_D$ and a *finitetary projection* if in addition $im(f)$ is a domain. A retraction $f:D\to D$ is a *closure* if $f \geq id_D$ and a *finitetary closure* if in addition $im(f)$ is a domain.

**Def.** If $D$ and $E$ are *cpos*, a pair of continuous functions $f:D\to E$ and $g:E\to D$ are a *retraction pair* if $g \circ f \leq id_D$ and $f \circ g = id_E$. The function $g$ is called an *embedding*, and $f$ is called a *projection*.

# Appendix B

## Proofs for Section 3.1.4

Here we present the technical details for Section 3.1.4. We can interpret Mackinlay's expressiveness conditions as follows:

**Condition 1.** $\forall P \in MON(U \rightarrow \{\perp, 1\}). \exists Q \in MON(V \rightarrow \{\perp, 1\}).$

$\forall u \in U. P(u) = Q(D(u)).$

**Condition 2.** $\forall Q \in MON(V \rightarrow \{\perp, 1\}). \exists P \in MON(U \rightarrow \{\perp, 1\}).$

$\forall v \in V. Q(v) = P(D^{-1}(v)).$

**Prop. B.1.** If $D{:}U \rightarrow V$ satisfies Condition 2 then $D$ is a monotone bijection from $U$ onto $V$.

**Proof.** $D$ is a function from $U$ to $V$, and Condition 2 requires that $D^{-1}$ is a functon from $V$ to $U$, so Conditon 2 requires that $D$ is a bijection from $U$ onto $V$. Next, assume that $x \leq y$, and let $Q_X = \lambda v \in V$. (if ($v \geq D(x)$) then 1 else $\perp$). Then by Condition 2 there is a monotone function $P_X$ such that $\forall v \in V. Q_X(v) = P_X(D^{-1}(v))$. Since $D$ is a bijection, this is equivalent to $\forall u \in U. Q_X(D(u)) = P_X(u)$. Hence, $Q_X(D(y)) = P_X(y) \geq P_X(x) = Q_X(D(x)) = 1$ so $Q_X(D(y)) = 1$ and $D(y) \geq D(x)$. Thus $D$ is monotone. ■

By Prop. B.1, Conditon 2 is too strong since it requires that every display in $V$ is the display of some data object under $D$. Since $U$ is a complete lattice it contains a maximal data object $X$ (the least upper bound of all members of $U$). For all data objects

$u \in U, u \leq X$. Since $D$ is monotone this implies $D(u) \leq D(X)$. We use the notation $\downarrow D(X)$ for the set of all displays less than $D(X)$. $\downarrow D(X)$ is a complete lattice and for all data objects $u \in U, D(u) \in \downarrow D(X)$. Hence

we can replace $V$ by $\downarrow D(X)$ in Condition 2 in order to not require that every $v \in V$ is the display of some data object. We modify Condition 2 as follows:

**Condition 2'.** $\forall Q \in MON(\downarrow D(X) \to \{\bot, 1\}).\ \exists P \in MON(U \to \{\bot, 1\}).$

$$\forall v \in \downarrow D(X).\ Q(v) = P(D^{-1}(v)).$$

**Def.** A function $D{:}U \to V$ is a *display function* if it satisfies Conditions 1 and 2'.

The next two propositions demonstrate the consequences of this definition.

**Prop. B.2.** If $D{:}U \to V$ is a display function then:

(a) $D$ is a bijective order embedding from $U$ onto $\downarrow D(X)$

(b) $\forall v \in V.\ (\exists u' \in U.\ v \le D(u') \Rightarrow \exists u \in U.\ v = D(u))$

(c) $\forall M \subseteq U.\ \bigvee D(M) = D(\bigvee M)$ and $\forall M \subseteq U.\ \bigwedge D(M) = D(\bigwedge M)$.

**Proof.** For part (1), $D$ is a function from $U$ to $V$, and Condition 2' requires that $D^{-1}$ is a functon from $\downarrow D(X)$ to $U$, so $D$ is a bijection from $U$ onto $\downarrow D(X)$.

To show that $D$ is an order embedding, assume that $D(x) \le D(y)$, and let $P_x = \lambda u \in U.$ (if $(u \ge x)$ then 1 else $\bot$). Then by Condition 1 there is a monotone function $Q_x$ such that $\forall u \in U.\ Q_x(D(u)) = P_x(u)$. Hence, $P_x(y) = Q_x(D(y)) \ge Q_x(D(x)) = P_x(x) = 1$ so $P_x(y) = 1$ and $y \ge x$. Now assume that $x \le y$, and let

$Q_X = \lambda v \in V.$ (if $(v \geq D(x))$ then 1 else $\perp$). Then by Condition 2' there is a monotone function $P_X$ such that $\forall v \in V.\ Q_X(v) = P_X(D^{-1}(v))$. Since $D$ is a bijection, this is equivalent to $\forall u \in U.\ Q_X(D(u)) = P_X(u)$. Hence, $Q_X(D(y)) = P_X(y) \geq P_X(x) = Q_X(D(x)) = 1$ so $Q_X(D(y)) = 1$ and $D(y) \geq D(x)$. Thus $D$ is an order embedding.

For part (b), note that if $\exists u' \in U.\ v \leq D(u')$, then $v \leq D(X)$ and $v \in \downarrow D(X)$ so $\exists u \in U.\ v = D(u)$.

For part (c), $\forall m \in M.\ m \leq \bigvee M$ so $\forall m \in M.\ D(m) \leq D(\bigvee M)$ and so $\bigvee D(M) \leq D(\bigvee M)$. Thus, by part (2), $\exists u \in U.\ D(u) = \bigvee D(M)$, and $\forall m \in M.\ D(m) \leq D(u)$ so $\forall m \in M.\ m \leq u$ and thus $\bigvee M \leq u$. Therefore $D(\bigvee M) \leq D(u) = \bigvee D(M)$, and thus $D(\bigvee M) = \bigvee D(M)$.

Next, $\forall m \in M.\ \bigwedge M \leq m$ so $\forall m \in M.\ D(\bigwedge M) \leq D(m)$ and so $D(\bigwedge M) \leq \bigwedge D(M)$. For any $m \in M$, $\bigwedge D(M) \leq D(m)$, so, by part (2), $\exists u \in U.\ D(u) = \bigwedge D(M)$, and $\forall m \in M.\ D(u) \leq D(m)$ so $\forall m \in M.\ u \leq m$ and thus $u \leq \bigwedge M$. Therefore $\bigwedge D(M) = D(u) \leq D(\bigwedge M)$, and thus $D(\bigwedge M) = \bigwedge D(M)$. ■

As a corollary of Prop. B.2, next we show that display functions are lattice isomorphisms, and are continuous in the sense defined by Scott.

**Prop. B.3.** $D{:}U \to V$ is a display function if and only if it is a lattice isomorphism of $U$ onto $\downarrow D(X)$, which is a sublattice of $V$. Furthermore, a display function $D$ is continuous.

**Proof.** Assume $D{:}U \to V$ is a display function. For any $x, y \in U$, let $M = \{x, y\}$. Then, by Prop. B.2, $D(x \vee y) = D(x) \vee D(y)$ and $D(x \wedge y) = D(x) \wedge D(y)$, so $D$ is a lattice homomorphism. Next, $a, b \in \downarrow D(X) \Rightarrow a, b \leq D(X) \Rightarrow a \vee b,\ a \wedge b \leq D(X) \Rightarrow D(a \vee b), D(a \wedge b) \in \downarrow D(X)$, so $\downarrow D(X)$ is a sublattice of $V$. By Prop. B.2, $D$ is bijective, so it is a lattice isomorphism.

Assume $D{:}U \to \downarrow D(X)$ is a lattice isomorphism. If $x \le y$ then $D(y) = D(x \vee y) = D(x) \vee D(y) \ge D(x)$. If $D(x) \le D(y)$ then $y = D^{-1}(D(y)) = D^{-1}(D(x) \vee D(y)) = D^{-1}(D(x \vee y)) = x \vee y \ge x$. Thus $D$ is an order embedding. Hence it is injective [that is $D(x) = D(y) \Rightarrow D(x) \le D(y) \Rightarrow x \le y$ and $D(x) = D(y) \Rightarrow D(y) \le D(x) \Rightarrow y \le x$, so $D(x) = D(y) \Rightarrow x = y$]

so $D^{-1}$ is defined on $D(U) \subseteq V$. Given $P \in MON(U \to \{\bot, 1\})$, define $Q = \lambda v \in V. \bigvee\{P(D^{-1}(v')) \mid v' \le v \;\&\; v' \in D(U)\}$. The set of $v'$ such that $v' \le v$ and $v' \in D(U)$ always includes $D(\bot)$, so $Q$ is defined for all $v \in V$. $Q$ is a function from $V$ to $\{\bot, 1\}$, and $Q$ is monotone since $v_1 \le v_2 \Rightarrow \{v' \mid v' \le v_1 \;\&\; v' \in D(U)\} \subseteq \{v' \mid v' \le v_2 \;\&\; v' \in D(U)\}$. Then, for all $u \in U$,

$Q(D(u)) = \bigvee\{P(D^{-1}(v')) \mid v' \le D(u) \;\&\; v' \in D(U)\} =$

$P(D^{-1}(D(u))) \vee \bigvee\{P(D^{-1}(v')) \mid v' < D(u) \;\&\; v' \in D(U)\} =$

[since $P$ and $D^{-1}$ are both monotone, $v' < D(u) \Rightarrow P(D^{-1}(v')) \le P(D^{-1}(D(u)))$]

$P(D^{-1}(D(u))) = P(u)$.

This is equivalent to $P = Q \circ D$. Thus $D$ satisfies Condition 1.

Given $Q \in MON(V \to \{\bot, 1\})$, define $P = \lambda u \in U. Q(D(u))$. $P$ is a function from $U$ to $\{\bot, 1\}$, and $P$ is monotone since $Q$ and $D$ are monotone. Clearly $\forall u \in U. Q(D(u)) = P(u)$. Since $D$ is a lattice isomorphism it is a bijection from $U$ onto $\downarrow D(X)$ so this is equivalent to $\forall v \in \downarrow D(X). Q(v) = P(D^{-1}(v))$. Thus $D$ satisfies Condition 2' and is a display function.

A display function $D$ is an order embedding and thus monotone. For any directed set $M \subseteq U$, $\bigvee D(M) = D(\bigvee M)$ by Prop. B.2, so $D$ is continuous. ■

# Appendix C

## Proofs for Section 3.2.2

Here we present the technical details for Section 3.2.2. Our lattices of data objects and of displays are defined in terms of scalar types. Each scalar type defines a value set, which may be either discrete or continuous, and which includes the undefined value $\perp$. We use the symbol **R** to denote the set of real numbers.

**Def.** A *discrete scalar s* defines a countable value set $I_S$ that includes a least element $\perp$ and has discrete order. That is, $\forall x, y \in I_S. (x < y \Rightarrow (x = \perp \,\&\, y \neq \perp))$.

**Def.** A *continuous scalar s* defines a value set $I_S = \{\perp\} \cup \{[x, y] \mid x, y \in \mathbf{R} \,\&\, x \leq y\}$ (that is, the set of closed real intervals, plus $\perp$) with the order defined by: $\perp < [x, y]$ and $[x, y] \leq [u, v] \Leftrightarrow [u, v] \subseteq [x, y]$.

**Prop. C.1.** Discrete and continuous scalars are *cpos*. Discrete scalars are domains. However, a continuous scalar is not algebraic because its only compact element is $\perp$, and hence it is not a domain.

**Proof.** A discrete scalar *s* is clearly complete. To show that a continuous scalar *s* is complete, let *M* be a directed set in $I_S$. We need to show that $\bigvee M = \bigcap\{[u, v] \mid [u, v] \in M\}$ is an interval in $I_S$. Set $x = \max\{u \mid [u, v] \in M\}$ and $y = \min\{v \mid [u, v] \in M\}$. If $y < x$, set $a = x - y$, $y' = y + a / 3$ and $x' = x - a / 3$ so $y' < x'$. Then $\exists [u_1, v_1] \in M$. $v_1 \leq y'$ and $\exists [u_2, v_2] \in M$. $u_2 \geq x'$, so $[u_1, v_1] \cap [u_2, v_2] = \phi$. But *M* directed implies that $\exists [u_3, v_3] \in M$. $[u_3, v_3] \subseteq [u_1, v_1] \cap [u_2, v_2]$. This is a contradiction, so x ≤ y and $[x, y] = \bigvee M$.

Let *s* be continuous and pick $[x, y] \in I_S$. To see if $[x, y]$ is compact, set

$A_n = [x - 2^{-n}, y + 2^{-n}]$. Then $[x, y] = \bigvee_n A_n$ and $\{A_n\}$ is a directed set, but $\neg\exists A_n.\ [x, y] \leq A_n$ (that is, there is no interval $A_n$ contained in $[x, y]$. Thus $\perp$ is the only compact element in $I_S$ (for $s$ continuous). ■

We define a tuple space as the cross product of a set of scalar value sets, and define a data lattice whose members are sets of tuples. Note that we use the notation $\mathbf{X}A$ for the cross product of members of a set $A$.

**Def.** Let $S$ be a finite set of scalars, and let $X = \mathbf{X}\{I_S \mid s \in S\}$ be the set of tuples with an element from each $I_S$. Let $a_S$ denote the $s$ component of a tuple $a \in X$. Define an order relation on $X$ by: for $a, b \in X$, $a \leq b$ if $\forall s \in S.\ a_S \leq b_S$.

**Prop. C.2.** Let $A \subseteq X$. If $b_S = \bigvee\{a_S \mid a \in A\}$ is defined for all $s \in S$ then $b = \bigvee A$. If $c_S = \bigwedge\{a_S \mid a \in A\}$ for all $s \in S$ then $c = \bigwedge A$ (that is, *sups* and *infs* over $X$ are taken componentwise). Thus, $X$ is a *cpo*.

**Proof.** $\forall s \in S.\ \forall a \in A.\ a_S \leq b_S$, so $b$ is an upper bound for $A$. If $e$ is another upper bound for $A$, then $\forall s \in S.\ b_S \leq e_S$ (since $b_S$ is the least upper bound of $\{a_S \mid a \in A\}$). Thus, $b \leq e$, so $b$ is the least upper bound of $A$. The argument that $c = \bigwedge A$ is similar.

Let $A \subseteq X$ be a directed set, and let $A_S = \{a_S \mid a \in A\}$. If $\{a_{iS} \mid i\}$ is a finite subset of $A_S$, then $\{a_i \mid i\}$ is a finite subset of $A$, so $\exists e \in A.\ \forall i.\ a_i \leq e$. Then for each $s \in S$, $e_S \in A_S$ and $\forall i.\ a_{iS} \leq e_S$, so $A_S$ is a directed set, and thus $b_S = \bigvee A_S \in I_S$. As we just showed, $b = \bigvee A \in X$, so $X$ is complete. ■

**Def.** We use $POWER(X) = \{A \mid A \subseteq X\}$ to denote the set of all subsets of $X$.

As explained in Section 3.2.2, *POWER*(*X*) is not appropriate for a lattice structure, so we define equivalence classes on *POWER*(*X*) using the Scott topology. The Scott topology defines open and closed sets as follows.

**Def.** A set $A \subseteq X$ is *open* if $\uparrow A \subseteq A$ and, for all directed subsets $C \subseteq X$, $\bigvee C \in A \Rightarrow C \cap A \neq \phi$.

**Def.** A set $A \subseteq X$ is *closed* if $\downarrow A \subseteq A$ and, for all directed subsets $C \subseteq A$, $\bigvee C \in A$. We use $CL(X)$ to denote the set of all closed subsets of $X$.

**Def.** Define a relation $\leq_R$ on *POWER*(*X*) as follows: $A \leq_R B$ if for all open $C \subseteq X$, $A \cap C \neq \phi \Rightarrow B \cap C \neq \phi$. Also define a relation $\equiv_R$ on *POWER*(*X*) as follows: $A \equiv_R B$ if $A \leq_R B$ and $B \leq_R A$.

**Prop. C.3.** The relation $\equiv_R$ is an equivalence relation.

**Proof.** Clearly $\forall A.A \leq_R A$ and thus $\forall A.A \equiv_R A$. And $A \equiv_R B \Leftrightarrow A \leq_R B \,\&\, B \leq_R A \Leftrightarrow B \equiv_R A$. If $A \leq_R B$ and $B \leq_R C$ then for all open $E \subseteq X$, $A \cap E \neq \phi \Rightarrow B \cap E \neq \phi$ and $B \cap E \neq \phi \Rightarrow C \cap E \neq \phi$, so $A \cap E \neq \phi \Rightarrow C \cap E \neq \phi$, and thus $A \leq_R C$. So $\equiv_R$ is reflexive, symmetric and transitive, and therefore an equivalence relation. ■

If $A \equiv_R B$ and $C \equiv_R D$, then $A \leq_R C \Leftrightarrow B \leq_R D$. Thus the equivalence classes of $\equiv_R$ are ordered by $\leq_R$. Now we show that the closed sets of the Scott topology can be used in place of the equivalence classes.

**Def.** Given an equivalence class $E$ of the $\equiv_R$ relation, let $M_E = \bigcup E$.

**Prop. C.4.** Given an equivalence class $E$ of the $\equiv_R$ relation, then $M_E \in E$.

**Proof.** Pick some $A \in E$. Then $A \subseteq M_E$ so $A \leq_R M_E$. For all open $C \subseteq X$, we have $M_E \cap C \neq \phi \Rightarrow \exists B \in E.\ B \cap C \neq \phi$ (since $M_E = \bigcup E$), but $B \cap C \neq \phi \Rightarrow A \cap C \neq \phi$ (since $B \leq_R A$). Thus $M_E \leq_R A$ and $M_E \equiv_R A$ so $M_E \in E$. ■

**Prop. C.5.** Given an equivalence class $E$ of the $\equiv_R$ relation, then $M_E \in CL(X)$.

**Proof.** Given $a \in M_E$ and $b \leq a$, we need to show that $M_E \equiv_R M_E \cup \{b\}$ and hence that $b \in M_E$. Clearly $M_E \leq_R M_E \cup \{b\}$. For all open $C \subseteq X$, if $b \in C$ then $a \in C$ (since $b \leq a$) so $M_E \cap C \neq \phi$. Thus $M_E \cup \{b\} \leq_R M_E$ and $b \in M_E$.

Next, given a directed set $D \subseteq M_E$, let $b = \bigvee D$. Clearly $M_E \leq_R M_E \cup \{b\}$. For all open $C \subseteq X$, if $b \in C$ then $\exists c \in D.\ c \in C$ so $c \in M_E \cap C$. Thus $M_E \cup \{b\} \leq_R M_E$ and $b \in M_E$.

This shows that $M_E$ is closed. ■

**Prop. C.6.** Given equivalence classes $E$ and $E'$ of the $\equiv_R$ relation, then $E \leq_R E' \Leftrightarrow M_E \subseteq M_{E'}$, and $E = E' \Leftrightarrow M_E = M_{E'}$. If $A \subseteq X$ is a closed set, then for some equivalence class $E$, $A = M_E$.

**Proof.** Note that $E \leq_R E' \Leftrightarrow M_E \leq_R M_{E'}$. If $M_E \subseteq M_{E'}$, then for all $C \subseteq X$ (whether $C$ is open or not), $M_E \cap C \neq \phi \Rightarrow M_{E'} \cap C \neq \phi$ and thus $M_E \leq_R M_{E'}$. If

$\neg M_E \subseteq M_{E'}$, then there is $a \in M_E$ such that $a \notin M_{E'}$. The complement of $M_{E'}$, denoted $X \setminus M_{E'}$, is open, and $a \in M_E \cap (X \setminus M_{E'})$ but $M_{E'} \cap (X \setminus M_{E'}) = \phi$, so $\neg M_E \leq_R M_{E'}$.

$E = E' \Rightarrow M_E = \bigcup E = \bigcup E' = M_{E'}$. Conversely,

$M_E = M_{E'} \Rightarrow M_E \leq_R M_{E'} \,\&\, M_{E'} \leq_R M_E \Rightarrow E \leq_R E' \,\&\, E \leq_R E' \Rightarrow E = E'$. Thus $E \leftrightarrow M_E$ is a one-to-one correspondence between closed sets and equivalence classes of $\equiv_R$.

If $A \subseteq X$ is a closed set, then $A$ belongs to some equivalence class $E$ so $A \subseteq M_E$ and $A \equiv_R M_E$. If $A \neq M_E$ then there is $a \in M_E$ such that $a \notin A$. $X \setminus A$ is open and $a \in M_E \cap (X \setminus A)$ but $A \cap (X \setminus A) = \phi$, so $\neg M_E \leq_R A$. This contradicts $A \equiv_R M_E$ so $A = M_E$. ■

The last proposition showed that there is a one to one correspondence between the equivalence classes of $\equiv_R$ and $CL(X)$. Next, we show that these closed sets obey the usual laws governing intersections and unions of closed sets in a topology.

**Prop. C.7.** If $L$ is a set of closed subsets of $X$, then $\bigcap L$ is closed. If $L$ is finite, then $\bigcup L$ is closed. Furthermore, for all $x \in X$, $\downarrow x \in CL(X)$.

**Proof.** If $x \in \bigcap L$ and $y \leq x$, then for all $A \in L$, $x \in A$ and $\downarrow A \subseteq A$, so $y \in A$ and so $y \in \bigcap L$. Thus $\downarrow \bigcap L \subseteq \bigcap L$. If $C$ is a directed subset $C \subseteq \bigcap L$, then for all $A \in L$, $C \subseteq A$ and $\bigvee C \in A$. Thus $\bigvee C \in \bigcap L$ and $\bigcap L$ is closed.

Now assume $L$ is finite. If $x \in \bigcup L$ and $y \leq x$, then for some $A \in L$, $x \in A$ and $\downarrow A \subseteq A$, so $y \in A$ and so $y \in \bigcup L$. Thus $\downarrow \bigcup L \subseteq \bigcup L$. Let $C$ be a directed subset $C \subseteq \bigcup L$ and assume that $\bigvee C \notin \bigcup L$. Then $\forall A \in L. \bigvee C \notin A$ so, since all $A \in L$ are closed, $\forall A \in L. \neg C \subseteq A$. Thus $\forall A \in L. \exists c_A \in C. c_A \notin A$. Now, $\{c_A \mid A \in L\}$ is finite, so $\exists c \in C. \forall A \in L. c_A \leq c$. But $\forall A \in L. c_A \notin A \Rightarrow c \notin A$ (since $A \in L$ are down sets), so $c \notin \bigcup L$. This contradicts $C \subseteq \bigcup L$ so we must have $\bigvee C \in \bigcup L$. Thus $\bigcup L$ is closed.

Clearly $\downarrow(\downarrow x) \subseteq \downarrow x$. If $C \subseteq \downarrow x$ is a directed set (or any subset of $\downarrow x$), then $\forall c \in C.\ c \leq x$ so $\bigvee C \leq x$ and thus $\bigvee C \in \downarrow x$. Therefore $\downarrow x$ is closed. ■

Now we show that the equivalence classes of the $\equiv_R$ relation, and equivalently $CL(X)$, form a complete lattice.

**Prop. C.8.** If $W$ is a set of equivalence classes of the $\equiv_R$ relation, and then $\bigwedge W$ is defined and $\bigwedge W = E$ such that $M_E = \bigcap\{M_w \mid w \in W\}$. Similarly, $\bigvee W$ is defined and $\bigvee W = E$ such that $M_E$ is the smallest closed set containing $\bigcup\{M_w \mid w \in W\}$. Thus the equivalence classes of the $\equiv_R$ relation form a complete lattice, and equivalently $CL(X)$ is a complete lattice. If $W$ is finite and $E = \bigvee W$, then $M_E = \bigcup\{M_w \mid w \in W\}$.

**Proof.** By Prop. C.7, $\bigcap\{M_w \mid w \in W\}$ is closed and, by Prop. C.6, must be $M_E$ for some equivalence class $E$. Now, $\forall w \in W.\ M_E \subseteq M_w$ so $\forall w \in W.\ M_E \leq_R M_w$ and $\forall w \in W.\ E \leq_R w$. If $E'$ is an equivalence class such that $\forall w \in W.\ E' \leq_R w$, then $\forall w \in W.\ M_{E'} \subseteq M_w$, so $M_{E'} \subseteq M_E$ and $E' \leq_R E$. Thus $E = \bigvee W$.

By Prop. C.7, the intersection of all closed sets containing $\bigcup\{M_w \mid w \in W\}$ must be a closed set and, by Prop. C.6, must be $M_E$ for some equivalence class $E$. Now, $\forall w \in W.\ M_w \subseteq M_E$ so $\forall w \in W.\ M_w \leq_R M_E$ and $\forall w \in W.\ w \leq_R E$.
If $E'$ is an equivalence class such that $\forall w \in W.\ w \leq_R E'$, then $\forall w \in W.\ M_w \subseteq M_{E'}$, so $M_{E'}$ contains $\bigcup\{M_w \mid w \in W\}$. Thus $M_E \subseteq M_{E'}$ and $E \leq_R E'$. Therefore $E = \bigvee W$.

If $W$ is finite, then $\bigcup\{M_w \mid w \in W\}$ is closed and equal to $M_E$, where $E = \bigvee W$. ■

Now we prove two propositions that will be useful for determining when sets of tuples are closed.

**Prop. C.9.** If $a \in X$, $B \subseteq X$ and $a \leq \bigvee B$ then $a = \bigvee\{a \wedge b \mid b \in B\}$.

**Proof.** Let $a_S$ and $b_S$ denote the tuple components of $a$ and $b$. The order relation, *sups* and *infs* of a cross product are taken componentwise, so it is sufficient to prove the proposition for each tuple component. That is, we will show that

$\forall s \in S.\ a_S \leq \bigvee\{a_S \wedge b_S \mid b \in B\}$.

For discrete $s$, $I_S$ has the discrete order. If $\bigvee\{b_S \mid b \in B\} = \perp$ then $a_S = \perp$ and $\forall s \in S.\ b_S = \perp$, and the conclusion is clearly true. Otherwise, let $c_S = \bigvee\{b_S \mid b \in B\}$. Then $\forall b \in B.\ (b_S = \perp$ or $b_S = c_S)$. If $a_S = \perp$ then $\forall b \in B.\ a_S \wedge b_S = \perp$ and $a_S = \perp = \bigvee\{a_S \wedge b_S \mid b \in B\}$. Otherwise $a_S = c_S$ and $\forall b \in B.\ a_S \wedge b_S = b_S$ and $a_S = c_S = \bigvee\{b_S \mid b \in B\} = \bigvee\{a_S \wedge b_S \mid b \in B\}$.

For continuous $s$, the members of $I_S$ are real intervals, or are $\perp$. Let $a_S = [x_S, y_S]$ and $b_S = [x(b_S), y(b_S)]$, where we use $x = -\infty$ and $y = +\infty$ for $a_S = \perp$ or $b_S = \perp$. The order relation on $I_S$ corresponds to the inverse of interval containment, *sup* corresponds to intersection of intervals, and *inf* corresponds to the smallest interval containing the union of intervals. First, note that $\forall b \in B.\ a \wedge b \leq a$ and thus $\bigvee\{a \wedge b \mid b \in B\} \leq a$. So, it is only necessary to show that $a \leq \bigvee\{a \wedge b \mid b \in B\}$, or, in other words, that the intersection of the intervals $[\min\{x_S, x(b_S)\}, \max\{y_S, y(b_S)\}]$ for all $b \in B$ is contained in the interval $[x_S, y_S]$. This intersection of intervals is $[c, d] = [\max\{\min\{x_S, x(b_S)\} \mid b \in B\}, \min\{\max\{y_S, y(b_S)\} \mid b \in B\}]$. Now, $a_S \leq \bigvee\{b_S \mid b \in B\}$ says that $x_S \leq \max\{x(b_S) \mid b \in B\}$ and $\min\{x(b_S) \mid b \in B\} \leq y_S$. So for at least one $b \in B$, $x_S \leq x(b_S)$ and $\min\{x_S, x(b_S)\} = x_S$, and thus $c = \max\{\min\{x_S, x(b_S)\} \mid b \in B\} \geq x_S$. Similarly $d \leq y_S$, and so $[c, d] \subseteq [x_S, y_S]$, showing the needed containment. ■

**Prop. C.10.** If $Y \subseteq CL(X)$ then $B = \{\bigvee M \mid M \subseteq \bigcup Y \,\&\, M \text{ directed}\}$ is closed.

**Proof.** First, we show that $B$ is a down set. By Prop. C.9, $a \leq \bigvee M \Rightarrow a = \bigvee\{a \wedge m \mid m \in M\}$, so we need to show that $\bigvee\{a \wedge m \mid m \in M\}$ is directed when $M$ is. Given a finite set $\{a \wedge b_i \mid b_i \in M\}$ there is $c$ in $M$ such that $\forall i.\ b_i \leq c$, and thus $\bigvee_i b_i \leq c$. Now $\forall i.\ b_i \leq \bigvee_i b_i \Rightarrow \forall i.\ a \wedge b_i \leq a \wedge \bigvee_i b_i \Rightarrow \vee_i(a \wedge b_i) \leq a \wedge \bigvee_i b_i \leq a \wedge c$. However $a \wedge c \in \bigvee\{a \wedge m \mid m \in M\}$, so $\{a \wedge m \mid m \in M\}$ is directed, $a \in B$ and $B$ is a down set.

Next, we show that $B$ is closed under *sups*. Let $M$ be a directed subset of $B$ and we will show that $a = \bigvee M \in B$. For each $m \in M$ there is a directed set $Q(m) \subseteq \bigcup Y$ such that $m = \bigvee Q(m)$. Define $Q' = \bigcup \{Q(m) \mid m \in M\}$ and $Q = \{\bigvee C \mid C \subseteq Q' \,\&\, C \text{ finite}\}$. Note that $\bigvee Q'$ exists (and $= a$) so $\bigvee C$ exists. For each finite $C \subseteq Q'$, each $c \in C$ belongs to a member of $Y$. Thus $C$ is a subset of a finite union of members of $Y$, which is a closed set, so $\bigvee C$ must belong to this same closed set and therefore belongs to $\bigcup Y$. Thus $Q \subseteq \bigcup Y$. Pick a finite set $\{q_i\} \subseteq Q$. Each $q_i$ is the *sup* of a finite subset $C_i \subseteq Q'$, and $\bigvee_i q_i$ is the *sup* of the finite subset $\bigcup_i C_i$ of $Q'$. Thus $\bigvee_i q_i \in Q$ so $Q$ is a directed subset of $\bigcup Y$ with $a = \bigvee Q = \bigvee Q'$, so $a$ is a member of $B$. Thus $B$ is closed under sups, and is a closed set. ■

# Appendix D

## Proofs for Section 3.2.3

Here we present the technical details for Section 3.2.3.

**Def.** A set $T$ of *data types* can be defined from the set $S$ of scalars. Two functions, $SC$ and $DOM$ are defined with $T$, such that $\forall t \in T.\ SC(t) \subseteq S \ \&\ DOM(t) \subseteq S$.

$T$, $SC$ and $DOM$ are defined as follows:

(D.1) $s \in S \Rightarrow s \in T$ (that is, $S \subset T$)

$SC(s) = \{s\}$

$DOM(s) = \phi$.

(D.2) (for $i = 1,...,n.\ t_i \in T$) & $(i \neq j \Rightarrow SC(t_i) \cap SC(t_j) = \phi) \Rightarrow struct\{t_1;...;t_n\} \in T$

$SC(struct\{t_1;...;t_n\}) = \bigcup_i SC(t_i)$

$DOM(struct\{t_1;...;t_n\}) = \bigcup_i DOM(t_i)$

(D.3) $w \in S \ \&\ r \in T \ \&\ w \notin SC(r) \Rightarrow (array\ [w]\ of\ r) \in T$

$SC((array\ [w]\ of\ r)) = \{w\} \cup SC(r)$

$DOM((array\ [w]\ of\ r)) = \{w\} \cup DOM(r)$

The type $struct\{t_1;...;t_n\}$ is a *tuple* with *element* types $t_i$, and the type (*array* [*w*] *of r*) is an *array* with *domain* type *w* and *range* type *r*. *SC*(*t*) is the set of scalars occurring in *t*, and *DOM*(*t*) is the set of scalars occurring as array domains in *t*. Note that each scalar in *S* may occur at most once in a type in *T*.

**Def.** For each scalar $s \in S$, define a countable set $H_S \subseteq I_S$ such that for all $a, b \in H_S$, $a \wedge b \in H_S$, $a \vee b \in I_S \Rightarrow a \vee b \in H_S$, and such that $\forall a \in I_S.\ \exists A \subseteq H_S.\ a = \bigvee A$ (that is, $H_S$ is closed under *infs* and *sups*, and any member of $I_S$ is a *sup* of a set of members of $H_S$). For discrete *s* this implies that $H_S = I_S$ (recall that we defined discrete scalars as having countable value sets). Also note that, for continuous *s*, $H_S$ cannot be a *cpo*.

**Def.** Given a scalar *w*, let $FIN(H_w) = \{A \subseteq H_w \backslash \{\bot\} \mid A \text{ finite } \& \ \forall a, b \in A.\ \neg(a \leq b)\}$.

**Def.** Extend the definition of $H_t$ to $t \in T$ by:

(D.4) $t = struct\{t_1;...;t_n\} \Rightarrow H_t = H_{t_1} \times ... \times H_{t_n}$

(D.5) $t = (array\ [w]\ of\ r) \Rightarrow H_t = \bigcup\{(A \rightarrow H_r) \mid A \in FIN(H_w)\}$

**Def.** Define an embedding $E_t: H_t \rightarrow U$ by:

(D.6) $t \in S \Rightarrow E_t(a) = \downarrow(\bot,...,a,...,\bot)$

(D.7) $t = struct\{t_1;...;t_n\} \Rightarrow E_t((a_1,...,a_n)) = \{b_1 \vee ... \vee b_n \mid \forall i.\ b_i \in E_{t_i}(a_i)\}$

(D.8) $t = (array\ [w]\ of\ r) \Rightarrow$

$$[a \in (A \rightarrow H_r) \Rightarrow E_t(a) = \{b \vee c \mid x \in A \ \&\ b \in E_w(x) \ \&\ c \in E_r(a(x))\}]$$

The notation $\downarrow(\perp,...,a,...,\perp)$ in Eq. (D.6) indicates the closed set of all tuples less than $(\perp,...,a,...,\perp)$. As we will show in Prop. D.1, for all $a \in H_t$ and for all $b \in E_t(a)$, $b_s = \perp$ unless $s \in SC(t)$. Thus $b_1 \vee ... \vee b_n$ in Eq. (D.7) is the tuple that merges the non-$\perp$ components of the tuples $b_1, ..., b_n$., since the types $t_i$ in Eq. (D.7) are defined from disjoint sets of scalars. Similarly, $b \vee c$ in Eq. (D.8) is the tuple that merges the non-$\perp$ components of the tuples $b$ and $c$, since the scalar $w$ does not occur in the type $r$. Prop. D.2 will show that $E_t$ does indeed map members of $H_t$ to members of $U$.

**Def.** For $t \in T$ define $F_t = E_t(H_t)$.

**Prop. D.1.** Given $t \in T$ and $A \in F_t$, for all tuples $b \in A$,

$\forall s \in S.\ (s \notin SC(t) \Rightarrow b_s = \perp)$.

**Proof.** We prove this by induction on the structure of $t$. This is clearly true for $t \in S$. For $t = struct\{t_1;...;t_n\}$ pick $b = b_1 \vee ... \vee b_n \in A \in F_t$, where $b_i \in B_i \in F_{t_i}$. Then $b_s = b_{1s} \vee ... \vee b_{ns}$. By induction, $\forall i.\ \forall s.\ s \notin SC(t_i) \Rightarrow b_{is} = \perp$, so $\forall s.\ (\forall i.\ s \notin SC(t_i)) \Rightarrow b_s = \perp$, and so $\forall s.\ s \notin \bigcup_i SC(t_i) \Rightarrow b_s = \perp$. But $SC(t) = \bigcup_i SC(t_i)$.

For $t = (array\ [w]\ of\ r)$ pick $a = b \vee c \in A \in F_t$, where $b \in B \in F_w$ and $c \in C \in F_r$. Then $a_s = b_s \vee c_s$. By induction, $s \neq w \Rightarrow b_s = \perp$ and $s \notin SC(r)) \Rightarrow c_s = \perp$, so $\forall s.\ s \notin \{w\} \cup SC(r) \Rightarrow b_s = \perp$. But $SC(t) = \{w\} \cup SC(r)$. ■

The following propositions show that $E_t$ maps members of $H_t$ to closed sets, and that this mapping is injective.

**Prop. D.2.** For all $a \in H_t$, $E_t(a)$ is a closed set.

**Proof.** We prove this by induction on the structure of $t$. For $t \in S$, $E_t(a) = \downarrow(\bot,...,a,...,\bot)$ is closed, by Prop. C.7. For $t = struct\{t_1;...;t_n\}$, we need to show that $E_t(a) = \{b_1 \vee ... \vee b_n \mid \forall i.\ b_i \in E_{t_i}(a_i)\}$ is closed, where $a = (a_1,...,a_n)$. To show that $E_t(a)$ is a down-set, pick $b \leq b_1 \vee ... \vee b_n \in E_t(a)$. Then $\forall i.\ b \wedge b_i \leq b_i$ and hence $\forall i.\ b \wedge b_i \in E_{t_i}(a_i)$ (since these are down sets). Thus, by Prop. C.9, $b = (b \wedge b_1) \vee ... \vee (b \wedge b_n) \in E_t(a)$. To show that $E_t(a)$ is closed under *sups* of directed sets, pick a directed set $C \subseteq E_t(a)$ and for all $c \in C$ let $c = b_1(c) \vee ... \vee b_n(c)$ where $\forall i.\ b_i(c) \in E_{t_i}(a_i)$. We need to show that $C_i = \{b_i(c) \mid c \in C\}$ is a directed set. Pick a finite subset $\{b_i(c_j) \mid j\} \subseteq C_i$. Since $C$ is directed, there is $m \in C$ such that $\forall j.\ c_j \leq m$. Note that $m = b_1(m) \vee ... \vee b_n(m)$ where $\forall i.\ b_i(m) \in C_i$. Since the $t_i$ have disjoint sets of non-$\bot$ components, $\forall i.\ \forall j.\ b_i(c_j) \leq b_i(m)$. Thus $C_i$ is directed, and $\bigvee C_i \in E_{t_i}(a_i)$. Hence $\bigvee C = \bigvee C_1 \vee ... \vee \bigvee C_n \in E_t(a)$, and thus $E_t(a)$ is closed under *sups* of directed sets.

For $t = (array\ [w]\ of\ r)$, we need to show that $E_t(a) = \{b \vee c \mid x \in A\ \&\ b \in E_w(x)\ \&\ c \in E_r(a(x))\}$ is closed, where $a \in (A \to H_r)$. Define $E_t(a)_x = \{b \vee c \mid b \in E_w(x)\ \&\ c \in E_r(a(x))\}$. Note that $E_t(a)_x = E_{struct\{w;r\}}((a, a(x)))$ [where $struct\{w; r\}$ is a tuple type and $(a, a(x)) \in H_{struct\{w;r\}}$] and thus, by the argument above for tuple types, $E_t(a)_x$ is closed. Also note that $E_t(a) = \bigcup\{E_t(a)_x \mid x \in A\}$. However, A is finite, so $E_t(a)$ is a union of a finite number of closed sets, and thus is itself closed. ■

**Prop. D.3.** The embedding $E_t : H_t \to U$ is injective.

**Proof.** We prove this by induction on the structure of *t*.

Let *t* be a scalar and $a \neq b$. Then $\neg(a \leq b)$ or $\neg(b \leq a)$. Assume without loss of generality that $\neg(a \leq b)$. Then $(\bot,...,a,...,\bot) \in \downarrow(\bot,...,a,...,\bot) = E_t(a)$ but $(\bot,...,a,...,\bot) \notin \downarrow(\bot,...,b,...,\bot) = E_t(b)$, so $E_t(a) \neq E_t(b)$.

Let $t = struct\{t_1;...;t_n\}$ and $a = (a_1,...,a_n) \neq (b_1,...,b_n) = b$. Then $\exists k.\ a_k \neq b_k$ and, by the inductive hypothesis, $E_{t_k}(a_k) \neq E_{t_k}(b_k)$. Assume without loss of generality that $\exists c_k \in E_{t_k}(a_k).\ c_k \notin E_{t_k}(b_k)$, and for all $i \neq k$ pick $c_i \in E_{t_i}(a_i)$. Then $c_1 \vee ... \vee c_n \in E_t((a_1,...,a_n))$, but, since $c_k \notin E_{t_k}(b_k)$ and since $\forall s \in S. \forall i \neq k.\ c_{ks} \neq \bot \Rightarrow c_{is} = \bot$, $c_1 \vee ... \vee c_n \notin E_t((b_1,...,b_n))$. Thus $E_t((a_1,...,a_n)) \neq E_t((b_1,...,b_n))$.

Let $t = (array\ [w]\ of\ r)$ and $a \neq b$ where $a \in (A \to H_r)$ and $b \in (B \to H_r)$. Then either $A \neq B$ or $A = B\ \&\ \exists x \in A.\ a(x) \neq b(x)$. In the first case (that is, $A \neq B$), assume without loss of generality that $\exists x \in A.\ x \notin B$. If $\exists y \in B.\ x \leq y$ then $\neg\exists z \in A.\ y \leq z$ (otherwise $x \in A\ \&\ z \in A\ \&\ x \leq z$). Thus either $\exists x \in A.\ \neg(\exists y \in B.\ x \leq y)$ or $\exists y \in B.\ \neg(\exists z \in A.\ y \leq z)$. Assume without loss of generality that $\exists x \in A.\ \neg(\exists y \in B.\ x \leq y)$. Then $e = (\bot,...,x,...,\bot) \in E_w(x)$ and $\neg(\exists y \in B.\ e \in E_w(y))$. Pick $f \in E_r(a(x))$. Then $e \vee f \in E_t(a)$ but $e \vee f \notin E_t(b)$, so $E_t(a) \neq E_t(b)$. In the second case (that is, $A = B\ \&\ \exists x \in A.\ a(x) \neq b(x)$), by the inductive hypothesis, $E_r(a(x)) \neq E_r(b(x))$. Assume without loss of generality that $\exists x \in A.\ \exists f \in E_r(a(x)).\ f \notin E_r(b(x))$. Pick $e \in E_w(x)$. Then $e \vee f \in E_t(a)$ but $e \vee f \notin E_t(b)$, so $E_t(a) \neq E_t(b)$. ■

Because $E_t : H_t \to U$ is injective, we can define an order relation between the members of $H_t$ simply by assuming that $E_t$ is an order embedding. If $E_t$ were not injective, it would map a pair of members of $H_t$ to the same member of *U*, and induce an anti-symmetric relation on $H_t$.

**Def.** Given $a, b \in H_t$, we say that $a \leq b$ if and only if $E_t(a) \leq E_t(b)$.

The order that $E_t$ induces on $H_t$ has a simple and intuitive structure, as the following proposition shows.

**Prop. D.4.** If $t$ is a scalar and $a, b \in H_t$ then $E_t(a) \leq E_t(b)$ if and only if $a \leq b$ in $I_t$.

If $t = struct\{t_1;...;t_n\}$ then $E_t((a_1,...,a_n)) \leq E_t((b_1,...,b_n))$ if and only if $\forall i.\ E_{t_i}(a_i) \leq E_{t_i}(b_i)$ (that is, the order relation between tuples is defined element-wise).

If $t = (array\ [w]\ of\ r)$, if $a, b \in H_t$ and if $a \in (A \to H_r)$ and $b \in (B \to H_r)$, then $E_t(a) \leq E_t(b)$ if and only if $\forall x \in A.\ E_r(a(x)) \leq \bigvee\{E_r(b(y)) \mid y \in B\ \&\ E_w(x) \leq E_w(y)\}$ (that is, an array $a$ is less than an array $b$ if the embedding of the value of $a$ at any sample $x$ is less than the *sup* of the embeddings of the set of values of $b$ at its samples greater than $x$).

**Proof.** Recall that members of $U$ are closed sets ordered by set inclusion, so $E_t(a) \leq E_t(b) \Leftrightarrow E_t(a) \subseteq E_t(b)$. Let $t$ be a scalar. If $a \leq b$ in $I_t$ then

$$E_t(a) = \downarrow(\bot,...,a,...,\bot) = \{(\bot,...,c,...,\bot) \mid c \leq a\} \subseteq$$

$$\{(\bot,...,c,...,\bot) \mid c \leq b\} = \downarrow(\bot,...,b,...,\bot) = E_t(b).$$

Now assume that $E_t(a) \leq E_t(b)$. Then

$$E_t(a) = \downarrow(\bot,...,a,...,\bot) = \{(\bot,...,c,...,\bot) \mid c \leq a\} \subseteq$$

$$\{(\bot,...,c,...,\bot) \mid c \leq b\} = \downarrow(\bot,...,b,...,\bot) = E_t(b)$$

so $(\bot,...,a,...,\bot) \in \{(\bot,...,c,...,\bot) \mid c \leq b\}$ so $a \leq b$ in $I_t$.

Let $t = struct\{t_1;...;t_n\}$. If $\forall i. E_{t_i}(a_i) \subseteq E_{t_i}(b_i)$ then

$E_t((a_1,...,a_n)) = \{c_1 \vee ... \vee c_n \mid \forall i.\ c_i \in E_{t_i}(a_i)\} \subseteq$

$\{c_1 \vee ... \vee c_n \mid \forall i.\ c_i \in E_{t_i}(b_i)\} = E_t((b_1,...,b_n))$.

Now assume that $E_t((a_1,...,a_n)) \leq E_t((b_1,...,b_n))$. Then

$E_t((a_1,...,a_n)) = \{c_1 \vee ... \vee c_n \mid \forall i.\ c_i \in E_{t_i}(a_i)\} \subseteq$

$\{c_1 \vee ... \vee c_n \mid \forall i.\ c_i \in E_{t_i}(b_i)\} = E_t((b_1,...,b_n))$.

[Parenthetical argument: assume that $c_k \notin E_{t_k}(b_k)$, $c_i \in E_{t_i}(a_i)$ for $i \neq k$, and $c_1 \vee ... \vee c_n \in E_t((b_1,...,b_n))$. Then there are $d_i \in E_{t_i}(b_i)$ such that $c_1 \vee ... \vee c_n = d_1 \vee ... \vee d_n$. However, $i \neq j \Rightarrow SC(t_i) \cap SC(t_j) = \phi$ so, by Prop. D.1, $d_{is} = \perp$ for $s \in SC(t_k)$ and $i \neq k$. Thus $c_{ks} = d_{ks}$ for $s \in SC(t_k)$ and so $c_k = d_k$. This is impossible, so $c_1 \vee ... \vee c_n \in E_t((b_1,...,b_n))$ and $c_i \in E_{t_i}(a_i)$ for $i \neq k \Rightarrow c_k \in E_{t_k}(b_k)$.]

Thus $\forall i.\ (c_i \in E_{t_i}(a_i) \Rightarrow c_i \in E_{t_i}(b_i))$, or in other words, $\forall i. E_{t_i}(a_i) \subseteq E_{t_i}(b_i)$.

Let $t = (array\ [w]\ of\ r)$, $a, b \in H_t$ and $a \in (A \to H_r)$ and $b \in (B \to H_r)$. Assume that $\forall x \in A.$ $E_r(a(x)) \leq \bigvee\{E_r(b(y)) \mid y \in B \ \&\ E_w(x) \leq E_w(y)\}$. Then

$\forall x \in A.\ E_r(a(x)) \subseteq \bigcup\{E_r(b(y)) \mid y \in B \ \&\ E_w(x) \leq E_w(y)\}$.

$E_t(a) = \{e \vee f \mid x \in A \ \&\ e \in E_w(x) \ \&\ f \in E_r(a(x))\} =$

$\bigcup\{\{e \vee f \mid e \in E_w(x) \ \&\ f \in E_r(a(x))\} \mid x \in A\}$.

Now, $f \in E_r(a(x)) \Rightarrow \exists y \in B.\ E_w(x) \leq E_w(y)\ \&\ f \in E_r(b(y))$ and

$e \in E_w(x)\ \&\ E_w(x) \leq E_w(y) \Rightarrow e \in E_w(y)$, so (continuing the chain)

$$\bigcup\{\{e \vee f \mid e \in E_w(x) \;\&\; f \in E_r(a(x))\} \mid x \in A\} \subseteq$$

$$\bigcup\{\{e \vee f \mid e \in E_w(y) \;\&\; f \in E_r(b(y)) \;\&\; E_w(x) \leq E_w(y) \;\&\; y \in B\} \mid x \in A\} \subseteq$$

$$\bigcup\{\{e \vee f \mid e \in E_w(y) \;\&\; f \in E_r(b(y))\} \mid y \in B\} =$$

$$\{e \vee f \mid y \in B \;\&\; e \in E_w(y) \;\&\; f \in E_r(b(y))\} = E_t(b).$$

Thus $E_t(a) \leq E_t(b)$.

Now assume that $E_t(a) \leq E_t(b)$. That is,

$$E_t(a) = \{e \vee f \mid x \in A \;\&\; e \in E_w(x) \;\&\; f \in E_r(a(x))\} \subseteq$$

$$\bigcup\{\{e \vee f \mid e \in E_w(y) \;\&\; f \in E_r(b(y))\} \mid y \in B\} = E_t(b).$$

Since $w \notin SC(r)$, $e \in E_w(x) \;\&\; f \in E_r(a(x)) \;\&\; e \vee f \in E_t(b) \Rightarrow \exists y \in B.\ e \in E_w(y) \;\&\; f \in E_r(b(y))$

[this is a result of the parenthetical argument in the tuple case of this proof]. Pick $x \in A$ and $f \in E_r(a(x))$, and define $e = (\bot,...,x,...,\bot)$. Then $\exists y \in B.\ e \in E_w(y) \;\&\; f \in E_r(b(y))$. Now $e \in E_w(y) \Rightarrow x \leq y \Rightarrow E_w(x) \leq E_w(y)$ so $f \in \bigcup\{E_r(b(y)) \mid y \in B \;\&\; E_w(x) \leq E_w(y)\} =$

$\bigvee\{E_r(b(y)) \mid y \in B \;\&\; E_w(x) \leq E_w(y)\}$. Thus

$\forall x \in A.\ E_r(a(x)) \leq \bigvee\{E_r(b(y)) \mid y \in B \;\&\; E_w(x) \leq E_w(y)\}$. ■

# Appendix E

## Proofs for Section 3.2.4

Here we present the technical details for Section 3.2.4.

**Def.** Given $A \in U$, define $MAX(A) = \{a \in A \mid \forall b \in A.\ \neg(a < b)\}$. That is, $MAX(A)$ consists of the maximal elements of $A$.

**Zorn's Lemma.** Let $P$ be a non-empty ordered set in which every chain has an upper bound. Then $P$ has a maximal element.

**Prop. E.1.** $\forall A \in U.\ A \subseteq \downarrow MAX(A)$, and hence $A = \downarrow MAX(A)$.

**Proof.** Pick $A \in U$ and $a \in A$ and define $P_a = \{x \in A \mid a \leq x\}$. For all chains $C \subseteq P_a$, $C$ is a directed set and $C \subseteq A$, so $b = \bigvee C \in A$ (since $A$ is closed). If $C$ is not empty, then $a \leq b$ so $b \in P_a$. Thus, every chain in $P_a$ has an upper bound in $P_a$, so by Zorn's Lemma, $P_a$ has a maximal element $d$. If there is any $c \in A$ such that $d < c$ then $a < c$ so $c \in P_a$, contradicting the maximality of $d$ in $P_a$. Thus $d \in MAX(A)$ and $a \in \downarrow MAX(A)$. Therefore $A \subseteq \downarrow MAX(A)$. Clearly $MAX(A) \subseteq A$, and, since $A$ is closed, $\downarrow MAX(A) \subseteq \downarrow A \subseteq A$ and so $A = \downarrow MAX(A)$. ■

**Prop. E.2.** $\forall A, B \in U.\ A = B \Leftrightarrow MAX(A) = MAX(B)$.

**Proof.** Assume $A$ and $B$ are in $U$. Clearly, $A = B \Rightarrow MAX(A) = MAX(B)$. To show the converse, assume $A \neq B$ and, without loss of generality, that $a \in A$ & $a \notin B$. Since $A \subseteq \downarrow MAX(A)$, there must be $c \in MAX(A)$ with $a \leq c$. However, since $B$ is a down-set, $c \notin B$, and hence $c \notin MAX(B)$. Thus $MAX(A) \neq MAX(B)$. ■

**Prop. E.3.** $\forall A \in U.\ A \equiv_R MAX(A)$.

**Proof.** First, $MAX(A) \leq_R A$, since $MAX(A) \subseteq A$. Now, if $A \cap C \neq \phi$ for $C \subseteq X$ open then $\exists a \in A \cap C$. Now, $A \subseteq \downarrow MAX(A)$ so $\exists b \in MAX(A).\ a \leq b$. However, since $C$ is open $b \in C$ so $b \in A \cap C$ and $MAX(A) \cap C \neq \phi$. Thus $A \leq_R \mathrm{MAX}(A)$ and

$A \equiv_R MAX(A)$. ■

**Prop. E.4.** Given a tuple type $t = struct\{t_1;...;t_n\} \in T$, $A \in F_t$ and $a = a_1 \vee ... \vee a_n \in A$, where $\forall i\ .\ a_i \in A_i \in F_{t_i}$, then $a \in MAX(A) \Leftrightarrow \forall i.\ a_i \in MAX(A_i)$.

**Proof.** Note that $a$ and the $a_i$ are tuples, and the *sup* of tuples is taken componentwise, so $\forall s \in S.\ a_s = a_{1s} \vee ... \vee a_{ns}$. Also note that $i \neq j \Rightarrow SC(t_i) \cap SC(t_i) = \phi$. If there is some $i$ such that $a_i \notin MAX(A_i)$, then $\exists b_i \in A_i.\ a_i < b_i$ so $b = a_1 \vee ... \vee b_i \vee ... \vee a_n \in A$. Now, $a_i < b_i \Rightarrow \exists s \in S.\ a_{is} < b_{is}$ and (since $j \neq i \Rightarrow a_{js} = \perp = b_{js}$) $a_s = a_{is}$ and $b_s = b_{is}$, so $a < b$. Thus $a \notin MAX(A)$. Conversely, if $a \notin MAX(A)$ then $\exists b \in A.\ a < b$ with $a = a_1 \vee ... \vee a_n$, $b = b_1 \vee ... \vee b_n$, and $\forall i.\ a_i, b_i \in A_i$. For some $s \in S$, $a_s < b_s$. Thus $b_s > \perp$ so $\exists j.\ s \in SC(t_j)$, and so $a_s < b_s \Rightarrow a_j < b_j$ (since $a_s = a_{js}$ and $b_s = b_{js}$). Thus $a_j \notin MAX(A_j)$. ■

**Prop. E.5.** For all types $t \in T$ and all $A \in F_t$, $MAX(A)$ is finite. If $t \in S$ and $A = {\downarrow}(\perp,...,a,...,\perp) \in F_t$ then $MAX(A) = \{(\perp,...,a,...,\perp)\}$. If $t = struct\{t_1;...;t_n\} \in T$ and $A = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} \in F_t$ then $MAX(A) = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in MAX(A_i)\}$. If $t = (array\ [w]\ of\ r) \in T$ and $A = \{a_1 \vee a_2 \mid g \in G\ \&\ a_1 \in E_w(g)\ \&\ a_2 \in E_r(a(g))\} \in F_t$ then $MAX(A) = \{a_1 \vee a_2 \mid g \in G\ \&\ a_1 \in MAX(E_w(g))\ \&\ a_2 \in MAX(E_r(a(g)))\}$.

**Proof.** We will demonstrate this proposition by induction on the structure of $t$. Let $t \in S$ and let $A \in F_t$. Then $\exists a \in I_S.\ A = {\downarrow}(\perp,...,a,...,\perp)$, so $MAX(A) = \{(\perp,...,a,...,\perp)\}$. $MAX(A)$ has a single member and is thus finite.

Let $t = struct\{t_1;...;t_n\} \in T$ and let $A \in F_t$. By Prop. E.4, $MAX(A) = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in MAX(A_i)\}$. By the inductive hypothesis, the $MAX(A_i)$ are finite, so $MAX(A)$ is finite.

Let $t = (array\ [w]\ of\ r) \in T$ and let $A \in F_t$. There is a finite set $G \in FIN(H_w)$ and a function $a \in (G \to H_r)$ such that

$$A = \{a_1 \vee a_2 \mid g \in G \ \&\ a_1 \in E_w(g) \ \&\ a_2 \in E_r(a(g))\} =$$
$$\bigcup\{\{a_1 \vee a_2 \mid a_1 \in E_w(g) \ \&\ a_2 \in E_r(a(g))\} \mid g \in G\} = \bigcup\{A_g \mid g \in G\}$$

where we define $A_g = \{a_1 \vee a_2 \mid a_1 \in E_w(g) \ \&\ a_2 \in E_r(a(g))\}$. Each $A_g$ is an object in $F_{struct\{w;\ r\}}$ for the tuple type $struct\{w;\ r\}$. By Prop. E.4,

$$MAX(A_g) = \{a_1 \vee a_2 \mid a_1 \in MAX(E_w(g)) \ \&\ a_2 \in MAX(E_r(a(g)))\} =$$
$$\{(\bot,...,g,...,\bot) \vee a_2 \mid a_2 \in MAX(E_r(a(g)))\}$$

Pick $g \neq g'$ in $G$, and $b \in MAX(A_g)$ and $b' \in MAX(A_{g'})$. Then there are $b_2 \in MAX(E_r(a(g)))$ and $b_2' \in MAX(E_r(a(g')))$ such that $b = (\bot,...,g,...,\bot) \vee b_2$ and $b' = (\bot,...,g',...,\bot) \vee b_2'$. If $b > b'$ then $g > g'$ since $b_{2w} = b_{2w}' = \bot$. However, this contradicts the defintion of $FIN(H_w)$. Thus no $b \in MAX(A_g)$ is larger than any $b' \in MAX(A_{g'})$ for $g \neq g'$ in $G$. Thus

$$MAX(A) = MAX(\bigcup\{A_g \mid g \in G\}) = \bigcup\{MAX(A_g) \mid g \in G\} =$$

$$\bigcup\{\{a_1 \vee a_2 \mid a_1 \in MAX(E_w(g)) \,\&\, a_2 \in MAX(E_r(a(g)))\} \mid g \in G\} =$$

$$\{a_1 \vee a_2 \mid g \in G \,\&\, a_1 \in MAX(E_w(g)) \,\&\, a_2 \in MAX(E_r(a(g)))\}.$$

$G$ is finite, and by the inductive hypothesis, $MAX(E_w(g))$ and $MAX(E_r(a(g)))$ are finite, so $MAX(A)$ is finite.

■

# Appendix F

## Proofs for Section 3.4.1

Here we present the technical details for Section 3.4.1. First, two definitions are given to provide the context for the work in this and subsequent appendices.

**Def.** Let $S$ denote a finite set of scalars, let $X = \mathbf{X}\{I_S \mid s \in S\}$ denote a set of tuples, and let $U = CL(X)$ denote the lattice of data objects consisting of closed sets of tuples whose primitive values are taken from the scalars in $S$.

**Def.** Let $DS$ denote a finite set of display scalars, let $Y = \mathbf{X}\{I_d \mid d \in DS\}$ denote a set of tuples, and let $V = CL(Y)$ denote the lattice of displays consisting of closed sets of tuples whose primitive values are taken from the display scalars in $DS$.

Now we prove four propositions that we will use as lemmas in other proofs.

**Prop. F.1.** For all $A, B \in U$, $\downarrow A \wedge \downarrow B = \downarrow(A \wedge B)$.

**Proof.** $\downarrow A \wedge \downarrow B = \downarrow A \cap \downarrow B = \{C \mid C \leq A\} \cap \{C \mid C \leq B\} = \{C \mid C \leq A \,\&\, C \leq B\} = \{C \mid C \leq A \wedge B\} = \downarrow(A \wedge B)$. ■

**Prop. F.2.** $D(\phi) = \phi$ and $D(\{(\perp,...,\perp)\}) = \{(\perp,...,\perp)\}$.

**Proof.** First, note that $\forall u \in U.\ \phi \leq u$ and $\forall u \in U.\ u \neq \phi \Rightarrow \{(\perp,...,\perp)\} \leq u$. That is, $\phi$ is the least element in $U$, and $\{(\perp,...,\perp)\}$ is the next largest element in $U$. If

$D(\phi) = v > \phi$, then $\exists u \in U.\ D(u) = \phi$ and $u < \phi$, which is impossible. Thus $D(\phi) = \phi$. Similarly, if $D(\{(\bot,..., \bot)\}) = v > \{(\bot,...,\bot)\}$, then $\exists u \in U.\ D(u) = \{(\bot,...,\bot)\}$ and $u < \{(\bot,...,\bot)\}$. However, the only $u < \{(\bot,...,\bot)\}$ is $\phi$, and $D(\phi) = \phi$, so $D(\{(\bot,...,\bot)\}) = \{(\bot,...,\bot)\}$. ■

**Prop. F.3.** If $D{:}U \to V$ is a display function, then its inverse $D^{-1}$ is a continuous function from $D(U)$ to $U$.

**Proof.** First, $D^{-1}$ is a function since $D$ is injective, and $D^{-1}$ is monotone since $D$ is an order embedding. $D^{-1}$ is continuous if for all directed $M \subseteq D(U)$, $\bigvee D^{-1}(M) = D^{-1}(\bigvee M)$. However, since $D$ is a homomorphism, $D^{-1}(M)$ is a directed set in $U$. Thus, since $D$ is continuous, $\bigvee D(D^{-1}(M)) = D(\bigvee D^{-1}(M))$, and so $D^{-1}(\bigvee D(D^{-1}(M))) = D^{-1}(D(\bigvee D^{-1}(M)))$. This simplifies to $D^{-1}(\bigvee M) = \bigvee D^{-1}(M)$, showing that $D^{-1}$ is continuous. ■

**Prop. F.4.** If $D{:}U \to V$ is a display function, then $\forall M \subseteq D(U).\ \bigvee D^{-1}(M) = D^{-1}(\bigvee M)$.

**Proof.** Given $M \subseteq D(U)$ let $N = D^{-1}(M) \subseteq U$. By Prop. B.2, $\bigvee D(N) = D(\bigvee N)$, which is equivalent to $\bigvee M = D(\bigvee D^{-1}(M))$, and applying $D^{-1}$ to both sides of this, we get $D^{-1}(\bigvee M) = D^{-1}(D(\bigvee D^{-1}(M))) = \bigvee D^{-1}(M)$. ■

Now we define an open neighborhood of a tuple in $X$, and prove two more lemmas. Note that in the following we will use the notation $a_S$ to indicate the $s$ component of a tuple $a \in \mathbf{X}\{I_S \mid s \in S\}$.

**Def.** Given a tuple $a \in \mathbf{X}\{I_S \mid s \in S\}$ such that $a_S \neq [x, x]$ for continuous $s$, define *neighbor*($a$) as the set of tuples $b$ such that:

$s$ discrete $\Rightarrow b_S \geq a_S$

$s$ continuous and $a_S = \perp \Rightarrow b_S \geq a_S$

$s$ continuous and $a_S \neq \perp \Rightarrow b_S > a_S$

(that is $a_S = [x, y]$ and $b_S = [u, v] \Rightarrow x < u$ and $v < y$).

**Prop. F.5.** For $a \in \bigtimes\{I_S \mid s \in S\}$, the set *neighbor*($a$) is open (in the Scott topology).

**Proof.** Clearly *neighbor*($a$) is an up set. Let $C$ be a directed set in $\bigtimes\{I_S \mid s \in S\}$ such that $d = \bigvee C$ belongs to *neighbor*($a$). The *sup* is taken componentwise, so $d_S = \bigvee\{c_S \mid c \in C\}$ for each $s$. If $s$ is discrete, then $\exists c^S \in C.\ c^S{}_S = d_S > a_S$. If $s$ is continuous and $a_S = \perp$, then for any $c \in C$, $c_S \geq a_S$. If $s$ is continuous and $a_S \neq \perp$, then $a_S$ and $d_S$ are intervals such that $d_S = [u, v] \subset [x, y] = a_S$, with $x < u$ and $v < y$. Here $u = \max\{p \mid \exists c \in C.\ [p, q] = c_S\}$ and $v = \min\{q \mid \exists c \in C.\ [p, q] = c_S\}$ so there exist $c^S{}_1 , c^S{}_2 \in C$ such that $c^S{}_{1S} = [p_1, q_1]$ and $c^S{}_{2S} = [p_2, q_2]$ with $x < p_1$ and $q_2 < y$. Since $C$ is directed, there must be $c^S \in C$ such that $c^S \geq c^S{}_1 \vee c^S{}_2$ , so $c^S{}_S > a_S$. For each $s \in S$ we have shown that there is $c^S \in C$ such that $c^S{}_S \geq a_S$. Since $S$ is finite, and $C$ is directed, there is $c \in C$ such that $c \geq \bigvee\{c^S \mid s \in S\} \geq a$ and $c \in$ *neighbor*($a$). Thus *neighbor*($a$) is an open set. ■

**Prop. F.6.** Given a set $C \subseteq U$, $B = \bigvee C$ and an open set $A$ in $\bigtimes\{I_S \mid s \in S\}$, then $A \cap B \neq \phi \Rightarrow \exists c \in C.\ A \cap c \neq \phi$.

**Proof.** $B$ and all $c \in C$ are closed, so $B$ is the smallest closed set containing $\bigcup C$. All the $c \in C$ are down sets, so $\bigcup C$ is also a down set. Thus, by Prop. C.10, $\{\bigvee M \mid M \subseteq \bigcup C$ & $M$ directed$\}$ is closed and hence equal to $B$. We are given that there is a $y \in A \cap B$, so there must be a directed set $M$ in $\bigcup C$ such that $y = \bigvee M$. However, since $A$ is open, there must be $m \in M \cap A$, and since $M \subseteq \bigcup C$, there is $c \in C$ such that $m \in c \cap A$. ■

Now we define the embeddings of scalar objects and display scalar objects in the lattices $U$ and $V$.

**Def.** For each scalar $s \in S$, define an embedding $E_S{:}I_S \to U$ by:

$\forall b \in I_S.\ E_S(b) = \downarrow(\bot,...,b,...,\bot)$ (this notation indicates that all elements of the tuple are $\bot$ except $b$). Also define $U_S = E_S(I_S) \subseteq U$.

**Def.** For each display scalar $d \in DS$, define an embedding $E_d{:}I_d \to V$ by:

$\forall b \in I_d.\ E_d(b) = \downarrow(\bot,...,b,...,\bot)$. Also define $V_d = E_d(I_d) \subseteq V$.

Next, we use an argument involving open neighborhoods to show that a display function maps embedded scalar objects to displays of the form $\downarrow x$, where $x$ is a display tuple. Prop. F.8 will show that these $\downarrow x$ must be embedded display scalar objects.

**Prop. F.7.** If $D{:}U \to V$ is a display function, then for all $s \in S$,

$\forall b \in I_S.\ \exists x \in \bigtimes\{I_d \mid d \in DS\}.\ D(\downarrow(\bot,...,b,...,\bot)) = \downarrow x$.

**Proof.** Given $s \in S$ and $b \in I_S$, let $a = (\bot,...,b,...,\bot)$ and let $z = D(\downarrow a)$. Then

$z = \bigvee\{\downarrow y \mid y \in z\}$, and by Prop. F.4, $\downarrow a = D^{-1}(z) = \bigvee\{D^{-1}(\downarrow y) \mid y \in z\}$ (note $\downarrow y \leq z$ so $D^{-1}(\downarrow y)$ exists).

Now we know that $a \in \bigvee\{D^{-1}(\downarrow y) \mid y \in z\}$. If we could show that $\bigvee\{D^{-1}(\downarrow y) \mid y \in z\} = \bigcup\{D^{-1}(\downarrow y) \mid y \in z\}$ then there must be $x \in z$ such that $a \in D^{-1}(\downarrow x)$. However, the $D^{-1}(\downarrow y)$ are closed sets, and, by Prop. C.8, we can only show that $\bigvee\{D^{-1}(\downarrow y) \mid y \in z\} = \bigcup\{D^{-1}(\downarrow y) \mid y \in z\}$ if $z$ is finite. Thus we need a more complex argument to construct $x \in z$ such that $a \in D^{-1}(\downarrow x)$.

Define a sequence of tuples $a_n$ in $U$, for $n$ =1, 2, ..., by:

if $s$ is continuous and $b = a_s = [x, y]$ for some interval $[x, y]$, then

$a_{ns} = [x\text{-}1/n, y\text{+}1/n]$

if $s$ is continuous and $b = a_s = \perp$, then $a_{ns} = \perp$

if $s$ is discrete, then $a_{ns} = a_s$

for all $s' \in S$ such that $s' \neq s$, $a_{ns'} = \perp$

Also define $z_n = D(\downarrow a_n) \leq D(\downarrow a) = z$, and note that $\downarrow a_n = \bigvee\{D^{-1}(\downarrow x) \mid x \in z_n\}$. Now $neighbor(a_{n\text{-}1})$ is open and $\downarrow a_n \cap neighbor(a_{n\text{-}1}) \neq \phi$, so by Prop. F.6 there must be $x_n \in z_n$ such that $D^{-1}(\downarrow x_n) \cap neighbor(a_{n\text{-}1}) \neq \phi$. Say $y$ is in this intersection. Then $y \in neighbor(a_{n\text{-}1}) \Rightarrow a_{n\text{-}1} \leq y$ and $y \in D^{-1}(\downarrow x_n) \Rightarrow \downarrow y \leq D^{-1}(\downarrow x_n)$ so $\downarrow a_{n\text{-}1} \leq \downarrow y \leq D^{-1}(\downarrow x_n)$. Furthermore, $x_n \in z_n \Rightarrow D^{-1}(\downarrow x_n) \leq D^{-1}(z_n) = \downarrow a_n$, so we have $\downarrow a_{n\text{-}1} \leq D^{-1}(\downarrow x_n) \leq \downarrow a_n$, or equivalently $\downarrow x_{n\text{-}1} \leq D(\downarrow a_{n\text{-}1}) \leq \downarrow x_n$. Thus $x_{n\text{-}1} \leq x_n$ and the set $\{x_n\}$ is a chain and thus a directed set. Since $\bigtimes\{I_d \mid d \in DS\}$ is a *cpo*, $x = \bigvee\{x_n\} \in \bigtimes\{I_d \mid d \in DS\}$. Since $z \in U$, $z$ is a closed under sups and thus $x \in z$.

Now, $\forall n.\ x_n \leq x$ so $\forall n.\ \downarrow a_n \leq D^{-1}(\downarrow x_{n+1}) \leq D^{-1}(\downarrow x)$. Thus $\downarrow a = \bigvee_n \downarrow a_n \leq D^{-1}(\downarrow x)$ (note that $a \in D^{-1}(\downarrow x)$) and $D(\downarrow a) \leq \downarrow x$. On the other hand, $x \in z \Rightarrow \downarrow x \leq z = D(\downarrow a)$, and so $D(\downarrow a) = \downarrow x$. ■

Prop. F.7 showed that a display function maps embedded scalar objects to displays of the form $\downarrow x$, where $x$ is a display tuple. Now we show that these $\downarrow x$ must be embedded display scalar objects, and that embedded scalar objects are mapped to embedded display scalar objects of the same kind (that is, discrete or continuous).

**Prop. F.8.** If $D{:}U \to V$ is a display function, then

$\forall s \in S.\ \forall a \in U_S.\ \exists d \in DS.\ D(a) \in V_d$.

Furthermore, if $s$ is discrete, then $d$ is discrete, and if $s$ is continuous, then $d$ is continuous.

**Proof.** A value $u \in U_S$ has the form $u = \downarrow(\perp,...,a,...,\perp)$. If $a = \perp$ then $D(u) = \{(\perp,...,\perp)\}$ which belongs to $V_d$ for all $d \in DS$. Otherwise, by Prop. F.7, $\exists v \in \mathbf{X}\{I_d \mid d \in DS\}.\ D(u) = \downarrow v$ and by Prop. F.2, $\downarrow v > \{(\perp,...,\perp)\}$. If $\downarrow v$ is not in any $V_d$, then some $(...,e,...,f,...) \in \downarrow v$ with $e \neq \perp \neq f$. We consider the discrete and continuous cases separately.

First, consider $s$ discrete. We have $\downarrow(...,e,...,\perp,...) < \downarrow v$ and $\exists u' \in U$ such that $D(u') = \downarrow(...,e,...,\perp,...) < \downarrow v = D(u)$, so $u' < u$. But the only $u'$ less than $u$ are $\phi$ and $\{(\perp,...,\perp)\}$, and $D$ does not carry them into $\downarrow(...,e,...,\perp,...)$. Thus $\downarrow v$ must be in some $V_d$.

Second, consider $s$ continuous. Define $w_{ef} = (\perp,...,e,...,f,...,\perp)$ (that is, $e$ and $f$ are the only elements in this tuple that are not $\perp$). Also define $v_e = \downarrow(\perp,...,e,...,\perp,...,\perp)$ and $v_f = \downarrow(\perp,...,\perp,...,f,...,\perp)$. Then $v_e,\ v_f < \downarrow w_{ef} \leq \downarrow v = D(u)$ so $\exists u_e,\ u_f < u.\ (D(u_e) = v_e\ \&\ D(u_f) = v_f)$. Now, $v_e \neq \{(\perp,...,\perp)\}$ so $u_e \neq \{(\perp,...,\perp)\}$ and $\exists a_e \neq \perp.\ (\perp,...,a_e,...,\perp) \in u_e$ and hence $\downarrow(\perp,...,a_e,...,\perp) \leq u_e$. Similarly, $\exists a_f \neq \perp.\ \downarrow(\perp,...,a_f,...,\perp) \leq u_f$. By Prop. F.1, $\downarrow(\perp,...,a_e \wedge a_f,...,\perp) \leq u_e \wedge u_f$. However, $a_e$ and $a_f$ are real intervals (since they belong to a continuous scalar and are not $\perp$), so $a_e \wedge a_f$ is the smallest interval containing both $a_e$ and $a_f$. Let $a_g$ be this interval. Then $a_g = a_e \wedge a_f \neq \perp$, and $\downarrow(\perp,...,a_g,...,\perp) \leq u_e \wedge u_f$. Thus $u_e \wedge u_f \neq \{(\perp,...,\perp)\}$. On the other hand, $v_e \wedge v_f = \{(\perp,...,\perp)\}$. But this contradicts $D(u_e \wedge u_f) = v_e \wedge v_f$, so $\downarrow v$ must be in some $V_d$.

Next we show that discrete scalar values map to discrete scalar values, and that continuous scalar values map to continuous scalar values.

Let $u = \downarrow(\bot,...,a,...,\bot) \in U_s$ for discrete $s$ with $D(u) = v = \downarrow(\bot,...,b,...,\bot) \in V_d$ and $b \neq \bot$. If $d$ is continuous, then $\exists b'.\ \bot < b' < b$ such that

$\{(\bot,...,\bot)\} < \downarrow(\bot,...,b',...,\bot) = v' < v$. Thus $\exists u'.\ D(u') = v'$ where

$\{(\bot,...,\bot)\} < u' < u = \downarrow(\bot,...,a,...,\bot)$. Thus $u' = \downarrow(\bot,...,a',...,\bot)$ where $a' < a$, which is impossible for discrete $s$, so $d$ must be discrete.

Let $u = \downarrow(\bot,...,a,...,\bot) \in U_s$ for continuous $s$ with $D(u) = v = \downarrow(\bot,...,b,...,\bot) \in V_d$. Then $\exists a'.\ \bot < a' < a$ and $\{(\bot,...,\bot)\} < \downarrow(\bot,...,a',...,\bot) = u' < u$, so

$D(\{(\bot,...,\bot)\}) = \{(\bot,...,\bot)\} < D(u') = v' < v$. This is only possible if $V_d$ is continuous. ■

Next we show that embedded objects from different scalars are not mapped to the same display scalar embedding.

**Prop. F.9.** If $D{:}U \rightarrow V$ is a display function, then for all $s$ and $s'$ in $S$,

$(s \neq s' \,\&\, u_a \in U_s \,\&\, u_b \in U_{s'} \,\&\, u_a \neq \bot \neq u_b \,\&\, D(u_a) \in V_d \,\&\, D(u_b) \in V_{d'}) \Rightarrow d \neq d'$.

**Proof.** Let $v_a = \mathrm{D}(u_a)$ and $v_b = \mathrm{D}(u_b)$. Assume that $v_a$ and $v_b$ are in the same $V_d$, and let

$u_a = \downarrow(\bot,...,a,...,\bot,...,\bot)$,

$u_b = \downarrow(\bot,...,\bot,...,b,...,\bot)$,

$v_a = \downarrow(\bot,...,e,...,\bot)$ and

$v_b = \downarrow(\bot,...,f,...,\bot)$, where $a \neq \bot \neq b$ and $e \neq \bot \neq f$.

This notation indicates that $u_a$ and $u_b$ are in different $U_s$, and that $v_a$ and $v_b$ are in the same $V_d$.

First, we treat the continuous case. $u_a \wedge u_b = \{(\bot,...,\bot)\}$ and, by Prop. F.1,

$v_a \wedge v_b = \downarrow(\bot,...,e \wedge f,...,\bot)$. $e$ and $f$ are real intervals, and $e \wedge f$ is the smallest interval containing both $e$ and $f$. Thus $e \wedge f \neq \bot$ so $v_a \wedge v_b \neq \{(\bot,...,\bot)\}$, which contradicts

$D(u_a \wedge u_b) = v_a \wedge v_b$. Thus $v_a$ and $v_b$ must be in the same $V_d$.

Second, treat the discrete case. Note that

$u_a \vee u_b$ = {(⊥,...,*a*,...,⊥,...,⊥), (⊥,...,⊥,...,*b*,...,⊥), (⊥,...,⊥)} and

$D(u_a \vee u_b) = v_a \vee v_b$ = {(⊥,...,*e*,...,⊥), (⊥,...,*f*,...,⊥), (⊥,...,⊥)}.

Let $x$ = ↓(⊥,...,*a*,...,*b*,...,⊥) =

{(⊥,...,*a*,...,*b*,...,⊥), (⊥,...,*a*,...,⊥,...,⊥), (⊥,...,⊥,...,*b*,...,⊥), (⊥,...,⊥)} > $u_a \vee u_b$.

Set $y = D(x)$. Then $y > v_a \vee v_b$ so there is (⊥,...,*g*,...,⊥) ∈ $y$ (all elements of this tuple are ⊥ except *g*) such that (⊥,...,*e*,...,⊥) ≠ (⊥,...,*g*,...,⊥) ≠ (⊥,...,*f*,...,⊥). [In fact (⊥,...,*g*,...,⊥) may not even be in the same $V_d$ that (⊥,...,*e*,...,⊥) and (⊥,...,*f*,...,⊥) are in.] Now if ↓(⊥,...,*g*,...,⊥) = $y$ then $e \leq g$ and $f \leq g$ which is impossible in the discrete order of $I_d$. Thus ↓(⊥,...,*g*,...,⊥) < $y$ and so $\exists w < x$. $D(w)$ = ↓(⊥,...,*g*,...,⊥). However, the only $w$ less than $x$ are ϕ, {(⊥,...,⊥)}, $u_a$ , $u_b$ and $u_a \vee u_b$. This contradicts $g \neq e$ and $g \neq f$. Thus $v_a$ and $v_b$ must be in the same $V_d$. ■

As a corollary of Prop. F.9, we show that only embedded scalar objects are mapped to embedded display scalar objects (that is, non-scalar objects must be mapped to non-display scalar objects).

**Prop. F.10.** If $D$:$U \rightarrow V$ is a display function, then $\forall d \in DS$. ($D(u) \in V_d \Rightarrow \exists s \in S$. $u \in U_s$).

**Proof.** If $u \in U$ is not in any scalar embedding, then $\exists$(...,*e*,...,*f*,...) ∈ $u$. $e \neq \perp \neq f$. Assume $D(u) = v \in V_d$. Then (⊥,...,*e*,...,⊥,...,⊥) ∈ $u$ and (⊥,...,⊥,...,*f*,...,⊥) ∈ $u$, so ↓(⊥,...,*e*,...,⊥,...,⊥) ≤ $u$ and ↓(⊥,...,⊥,...,*f*,...,⊥) ≤ $u$, and thus $D$(↓(⊥,...,*e*,...,⊥,...,⊥)) ∈ $V_d$ and $D$(↓(⊥,...,⊥,...,*f*,...,⊥)) ∈ $V_d$. However ↓(⊥,...,*e*,...,⊥,...,⊥) and ↓(⊥,...,⊥,...,*f*,...,⊥) are in two different scalar embeddings and, by Prop. F.9, cannot both be mapped to $V_d$. Thus $D(u)$ cannot belong to any display scalar embedding. ■

Next, we show that all embedded objects from a continuous scalar are mapped to embedded objects from the same display scalar. Note, however, that embedded objects from the same discrete scalar may be mapped to embedded objects from different display scalars.

**Prop. F.11.** If $D{:}U \rightarrow V$ is a display function and if $s$ is a continuous scalar, then $\forall u_a, u_b \in U_s.\ ((D(u_a) \in V_d \,\&\, D(u_b) \in V_{d'} \,\&\, u_a \neq \perp \neq u_b) \Rightarrow d = d')$.

**Proof.** Let $v_a = \mathrm{D}(u_a)$ and $v_b = \mathrm{D}(u_b)$. Assume that $s$ is continuous and that $v_a$ and $v_b$ are in different $V_d$. Let

$u_a = \downarrow(\perp,...,a,...,\perp)$,

$u_b = \downarrow(\perp,...,b,...,\perp)$,

$v_a = \downarrow(\perp,...,e,...,\perp,...,\perp)$ and

$v_b = \downarrow(\perp,...,\perp,...,f,...,\perp)$, where $a \neq \perp \neq b$ and $e \neq \perp \neq f$.

This notation indicates that $u_a$ and $u_b$ are in the same $U_S$, and that $v_a$ and $v_b$ are in different $V_d$. Now $v_a \wedge v_b = \{(\perp,...,\perp)\}$ and, by Prop. F.1, $u_a \wedge u_b = \downarrow(\perp,...,a \wedge b,...,\perp)$. Since $a$ and $b$ are real intervals, $a \wedge b$ is the smallest interval containing both $a$ and $b$, so $a \wedge b \neq \perp$. However, this contradicts $D(u_a \wedge u_b) = v_a \wedge v_b$. Thus, $v_a$ and $v_b$ must be in the same $V_d$. ■

Now we show that a display function maps objects of the form $\downarrow a$, for $a \in \mathbf{X}\{I_s \mid s \in S\}$, to objects of the form $\downarrow x$, for $x \in \mathbf{X}\{I_d \mid d \in DS\}$, and conversely. Furthermore, the values of display functions on objects of the form $\downarrow a$ are determined by their values on embedded scalar objects. Given this, it is an easy step in Prop. F.13 to show that the values of display functions on all of $U$ are determined by their values on embedded scalar objects.

**Prop. F.12.** If $D{:}U \to V$ is a display function and if $a$ is a tuple in $\mathbf{X}\{I_s \mid s \in S\}$ then there exists a tuple $x$ in $\mathbf{X}\{I_d \mid d \in DS\}$ such that $D(\downarrow a) = \downarrow x$. Conversely, if $x$ is a tuple in $\mathbf{X}\{I_d \mid d \in DS\}$ such that $\exists A \in U.\ x \in D(A)$, then there exists a tuple $a$ in $\mathbf{X}\{I_s \mid s \in S\}$ such that $D(\downarrow a) = \downarrow x$. From Prop. F.8 we know that for all $s \in S$,

$a_s \neq \perp \Rightarrow \exists d \in DS.\ \exists y_d \in I_d.\ (y_d \neq \perp\ \&\ \downarrow(\perp,...,y_d,...,\perp) = D(\downarrow(\perp,...,a_s,...,\perp)))$,

and similarly, from Prop. D.3 we know that for all $d \in DS$,

$x_d \neq \perp \Rightarrow \exists s \in S.\ \exists b_s \in I_s.\ (b_s \neq \perp\ \&\ \downarrow(\perp,...,x_d,...,\perp) = D(\downarrow(\perp,...,b_s,...,\perp)))$,

Here we assert that for all $s \in S$, $a_s \neq \perp \Rightarrow a_s = b_s$, and for all $d \in DS$, $x_d \neq \perp \Rightarrow x_d = y_d$. That is, the tuple elements of $a$ determine the tuple elements of $x$, and vice versa, according to the values of $D$ on the scalar embeddings $U_s$.

**Proof.** This is similar to the proof of Prop F.7. Given $a \in \mathbf{X}\{I_s \mid s \in S\}$, let $z = D(\downarrow a)$. Then $z = \bigvee\{\downarrow y \mid y \in z\}$, and by Prop. F.4, $\downarrow a = D^{-1}(z) = \bigvee\{D^{-1}(\downarrow y) \mid y \in z\}$ (note $\downarrow y \leq z$ so $D^{-1}(\downarrow y)$ exists).

Define a sequence of tuples $a_n$ in $U$, for $n$ =1, 2, ..., by:

$s$ discrete $\Rightarrow a_{ns} = a_s$

$s$ continuous and $a_s = \perp \Rightarrow a_{ns} = a_s$

$s$ continuous and $a_s = [x, y] \Rightarrow a_{ns} = [x\text{-}1/n, y\text{+}1/n]$.

Also define $z_n = D(\downarrow a_n) \leq D(\downarrow a) = z$, and note that $\downarrow a_n = \bigvee\{D^{-1}(\downarrow x) \mid x \in z_n\}$. Now $neighbor(a_{n\text{-}1})$ is open and $\downarrow a_n \cap neighbor(a_{n\text{-}1}) \neq \phi$. By Prop. F.6 there must be $x_n \in z_n$ such that $D^{-1}(\downarrow x_n) \cap neighbor(a_{n\text{-}1}) \neq \phi$. Say $y$ is in this intersection. Then $y \in neighbor(a_{n\text{-}1}) \Rightarrow a_{n\text{-}1} \leq y$ and $y \in D^{-1}(\downarrow x_n) \Rightarrow \downarrow y \leq D^{-1}(\downarrow x_n)$ so $\downarrow a_{n\text{-}1} \leq \downarrow y \leq D^{-1}(\downarrow x_n)$. Furthermore, $x_n \in z_n \Rightarrow D^{-1}(\downarrow x_n) \leq D^{-1}(\downarrow z_n) = \downarrow a_n$, so we have $\downarrow a_{n\text{-}1} \leq D^{-1}(\downarrow x_n) \leq \downarrow a_n$.

Now consider the tuple components of $a_n$ and $x_n$. Define $x_n$' by $\downarrow(\perp,...,x_{nd}',...,\perp) = D(\downarrow(\perp,...,a_{ns},...,\perp))$, and set $x_{nd}' = \perp$ for those $d$ not corresponding to any $a_{ns} \neq \perp$. Also define $a_n$' by $\downarrow(\perp,...,x_{nd},...,\perp) = D(\downarrow(\perp,...,a_{ns}',...,\perp))$ for those $d$ such that $x_{nd} \neq \perp$, and set $a_{ns}' = \perp$ for those $s$ not corresponding to any $x_{nd} \neq \perp$. Note that $\downarrow(\perp,...,x_{nd},...,\perp) \leq \downarrow x_n$ so $\exists w \in U.\ \downarrow(\perp,...,x_{nd},...,\perp) = D(w)$, and, by Prop. D.3, $w$ must have the form $\downarrow(\perp,...,a_{ns}',...,\perp)$, so $a_{ns}'$ exists for $x_{nd} \neq \perp$. First, we use $D^{-1}(\downarrow x_n) \leq \downarrow a_n$ to show that:

(a) $\downarrow(\perp,...,x_{nd},...,\perp) \leq \downarrow x_n \Rightarrow$

$\downarrow(\perp,...,a_{ns}',...,\perp) = D^{-1}(\downarrow(\perp,...,x_{nd},...,\perp)) \leq D^{-1}(\downarrow x_n) \leq \downarrow a_n \Rightarrow$

$a_{ns}' \leq a_{ns} \Rightarrow$

$\downarrow(\perp,...,a_{ns}',...,\perp) \leq \downarrow(\perp,...,a_{ns},...,\perp) \Rightarrow$

$\downarrow(\perp,...,x_{nd},...,\perp) = D(\downarrow(\perp,...,a_{ns}',...,\perp)) \leq$

$D(\downarrow(\perp,...,a_{ns},...,\perp)) = \downarrow(\perp,...,x_{nd}',...,\perp) \Rightarrow$

$x_{nd} \leq x_{nd}'$

The transition from the fourth to the fifth line in (a) shows that if $a_{ns}$ and $a_{ns}$' are in the same scalar *s*, then $x_{nd}$ and $x_{nd}$' are in the same display scalar *d*. Next, we use $\downarrow a_n \leq D^{-1}(\downarrow x_{n+1})$ to show that:

(b) $\downarrow(\bot,...,a_{ns},...,\bot) \leq \downarrow a_n \Rightarrow$

$\downarrow(\bot,...,x_{nd}',...,\bot) = D(\downarrow(\bot,...,a_{ns},...,\bot)) \leq D(\downarrow a_n) \leq \downarrow x_{n+1} \Rightarrow$

$x_{nd}' \leq x_{(n+1)d} \Rightarrow$

$\downarrow(\bot,...,x_{nd}',...,\bot) \leq \downarrow(\bot,...,x_{(n+1)d},...,\bot) \Rightarrow$

$\downarrow(\bot,...,a_{ns},...,\bot) = D^{-1}(\downarrow(\bot,...,x_{nd}',...,\bot)) \leq$

$D^{-1}(\downarrow(\bot,...,x_{(n+1)d},...,\bot)) = \downarrow(\bot,...,a_{(n+1)s}',...,\bot) \Rightarrow$

$a_{ns} \leq a_{(n+1)s}'$

The transition from the fourth to the fifth line in (b) shows that if $x_{nd}$ and $x_{(n+1)d}$' are in the same display scalar *d*, then $a_{ns}$ and $a_{(n+1)s}$' are in the same scalar *s*.

Putting (a) and (b) together shows that $a_{ns}' \leq a_{ns} \leq a_{(n+1)s}'$ and $x_{nd} \leq x_{nd}' \leq x_{(n+1)d}$ for all *s* and *d*. If *d* is a discrete display scalar, then there is an *n* such that $\forall m \geq n.\ x_{md} = x_{nd}$, and define $x_d = x_{nd}$. If *d* is a continuous display scalar, then there either all the $x_{nd}$ are $\bot$ or there is an *n* such that $\forall i, j \geq n.\ i \geq j \Rightarrow x_{id} = [u_i, v_i] \subseteq [u_j, v_j] = x_{jd}$. In the first case, define $x_d = \bot$ and in the second case define $x_d = [u, v] = \bigcap\{[u_i, v_i] \mid i \geq n\}$. In any case, $x_d = \bigvee_n x_{nd}$, and defining *x* as the tuple with components $x_d$, $x = \bigvee_n x_n$. Since *z* is closed, $\{x_n\}$ is a directed set, and $\forall n.\ x_n \in z$, then $x \in z$.

By definition, $a = \bigvee_n a_n$. We have already shown that $\downarrow a_{n-1} \le D^{-1}(\downarrow x_n)$, so $a_{n-1} \in D^{-1}(\downarrow x_n) \subseteq D^{-1}(\downarrow x)$. Since $D^{-1}(\downarrow x)$ is closed, $a \in D^{-1}(\downarrow x)$ and thus $\downarrow a \le D^{-1}(\downarrow x)$. However, $x \in z$, so $D^{-1}(\downarrow x) \le \downarrow a$ and thus $\downarrow a = D^{-1}(\downarrow x)$. Define $x'$ and $a'$ by $\downarrow(\perp,...,x_{nd}',...,\perp) = D(\downarrow(\perp,...,a_{ns},...,\perp))$ and $\downarrow(\perp,...,x_{nd},...,\perp) = D(\downarrow(\perp,...,a_{ns}',...,\perp))$. Then we can apply the logic of (a) and (b) (using $\downarrow a \le D^{-1}(\downarrow x) \le \downarrow a$) to show that $a_s' \le a_s \le a_s'$ and $x_d \le x_d' \le x_d$, which is just $a_s = a_s'$ and $x_d = x_d'$. Thus $D$ takes the set of tuple components of $a$ into exactly the set of tuple components of $x$.

For the converse, we are given a tuple $x$ in $\mathbf{X}\{I_d \mid d \in DS\}$ such that $\exists A \in U.\ x \in D(A)$. Then $\downarrow x \le D(A)$ and $\exists z \le A.\ \downarrow x = D(z) = \bigvee\{D(\downarrow b) \mid b \in z\}$. After this, the argument for the converse is identical, relying on properties of $D$ that are shared by $D^{-1}$. $D^{-1}$ is a homomorphism from $D(U)$ to $U$, and Props. F.3 and F.4 show that $D^{-1}$ is continuous and preserves arbitrary *sups*. In the argument $D^{-1}$ is only applied to members of $V$ that are less than $\downarrow x$, where $D^{-1}$ is guaranteed to be defined. ■

Proposition F.13 will show that the values of display functions on all of $U$ are determined by their values on the scalar embeddings $U_S$, which is particularly interesting since most elements of $U$ cannot be expressed as *sups* of sets of elements of the scalar embeddings $U_S$.

**Prop. F.13.** If $D{:}U \to V$ is a display function, then its values on $U$ are determined by its values on the scalar embeddings $U_S$.

**Proof.** For all $u \in U$, $u = \bigvee\{\downarrow x \mid x \in u\}$. By Prop. B.2, $D(u) = \bigvee\{D(\downarrow x) \mid x \in u\}$. Now, each $x \in u$ is a tuple so by Prop. F.12, $D(\downarrow x)$ is determined by the values of $D$ applied to the tuple components of $x$. Thus $D(u)$ is determined by the values of $D$ on the scalar embeddings $U_S$. ■

The propositions in Appendix F are combined in the following definition and theorem about mappings from scalars to display scalars.

**Def.** Given a display function $D$, define a mapping $MAP_D$: $S \to POWER(DS)$ by $MAP_D(s) = \{d \in DS \mid \exists a \in U_S.\ D(a) \in V_d\}$.

**Theorem. F.14.** Every display function $D$:$U \to V$ is an injective lattice homomorphism whose values are determined by its values on the scalar embeddings $U_S$. $D$ maps values in the scalar embedding $U_S$ to values in the display scalar embeddings $V_d$ for $d \in MAP_D(s)$. Furthermore,

$s$ discrete and $d \in MAP_D(s) \Rightarrow d$ discrete,

s continuous and $d \in MAP_D(s) \Rightarrow d$ continuous,

$s \neq s' \Rightarrow MAP_D(s) \cap MAP_D(s') = \phi$,

$s$ continuous $\Rightarrow MAP_D(s)$ contains a single display scalar.

# Appendix G

# Proofs for Section 3.4.2

Here we present the technical details for Section 3.4.2. First, we prove three lemmas that explore the relation between closed real intervals in terms of the lattice structure.

**Prop. G.1.** Given a continuous scalar $s \in S$, and $[x, y] \in I_S$, then

$\downarrow(\perp,...,[x, y],...,\perp) = \bigcap\{\downarrow(\perp,...,[z, z],...,\perp) \mid x \le z \le y\}$.

**Proof.** $\downarrow(\perp,...,[z, z],...,\perp) = \{[u, v] \mid u \le z \le v\}$ so

$\bigcap\{\downarrow(\perp,...,[z, z],...,\perp) \mid x \le z \le y\} = \{[u, v] \mid \forall z.\ (x \le z \le y \Rightarrow u \le z \le v)\} =$

$\{[u, v] \mid u \le x \le y \le v\} = \downarrow(\bot,...,[x, y],...,\bot)$. ■

**Prop. G.2.** Given a continuous scalar $s \in S$, and a set $A \subseteq I_S \backslash \{\bot\}$ such that

$\exists u'.\ \forall [u, v] \in A.\ u' \le u$ and $\exists v'.\ \forall [u, v] \in A.\ v \le v'$, then

$\downarrow(\bot,...,[inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}],...,\bot) =$

$\bigcap\{\downarrow(\bot,...,[u, v],...,\bot) \mid [u, v] \in A\}$.

**Proof.** Let $x = inf\{u \mid [u, v] \in A\}$ and $y = sup\{v \mid [u, v] \in A\}$. This *inf* and *sup* exist since the lower and upper bounds $u'$ and $v'$ exist. Then

$(\bot,...,[a, b],...,\bot) \in \downarrow(\bot,...,[x, y],...,\bot) \Leftrightarrow$

$a \le x \le y \le b \Leftrightarrow$

$\forall [u, v] \in A.\ a \le u \le v \le b \Leftrightarrow$

$\forall [u, v] \in A.\ (\bot,...,[a, b],...,\bot) \in \downarrow(\bot,...,[u, v],...,\bot) \Leftrightarrow$

$(\bot,...,[a, b],...,\bot) \in \bigcap\{\downarrow(\bot,...,[u, v],...,\bot) \mid [u, v] \in A\}$.

Thus $\downarrow(\bot,...,[x, y],...,\bot) = \bigcap\{\downarrow(\bot,...,[u, v],...,\bot) \mid [u, v] \in A\}$. ■

**Prop. G.3.** Given a display function $D{:}U \to V$, a continuous scalar $s \in S$, and $[x, y] \in I_S$, then $D(\downarrow(\bot,...,[x, y],...,\bot)) = \bigcap\{D(\downarrow(\bot,...,[z, z],...,\bot)) \mid x \le z \le y\}$.

**Proof.** $x \le w \le y \Rightarrow \bigwedge\{D(\downarrow(\bot,...,[z, z],...,\bot)) \mid x \le z \le y\} \le D(\downarrow(\bot,...,[w, w],...,\bot))$,

so there is $A \in U$ such that $D(A) = \bigwedge\{D(\downarrow(\bot,...,[z, z],...,\bot)) \mid x \le z \le y\} =$

$\bigcap\{D(\downarrow(\bot,...,[z, z],...,\bot)) \mid x \le z \le y\}$ (by Prop. C.8) and such that

$x \le w \le y \Rightarrow A \le \downarrow(\bot,...,[w, w],...,\bot)$. Thus $A \le \bigwedge\{\downarrow(\bot,...,[w, w],...,\bot) \mid x \le w \le y\} =$

$\bigcap\{\downarrow(\bot,...,[w, w],...,\bot) \mid x \le w \le y\} = \downarrow(\bot,...,[x, y],...,\bot)$ (by Prop. G.1).

On the other hand, $x \le z \le y \Rightarrow \downarrow(\bot,...,[x, y],...,\bot) \le \downarrow(\bot,...,[z, z],...,\bot) \Rightarrow$

$D(\downarrow(\bot,...,[x, y],...,\bot)) \le D(\downarrow(\bot,...,[z, z],...,\bot))$, so $D(\downarrow(\bot,...,[x, y],...,\bot)) \le D(A)$ and thus

$\downarrow(\bot,...,[x, y],...,\bot) \le A$. Therefore $\downarrow(\bot,...,[x, y],...,\bot) = A$ so

$D(\downarrow(\perp,...,[x, y],...,\perp)) = D(A) = \bigcap\{D(\downarrow(\perp,...,[z, z],...,\perp)) \mid x \le z \le y\}$. ■

Now we define the values of display functions on embedded continuous scalar objects in terms of functions of real numbers.

**Def.** Given a display function $D{:}U \to V$ and a continuous scalar $s \in S$, by Prop. F.8 and Prop. F.11 there is a continuous $d \in DS$ such that values in $U_S$ are mapped to values in $V_d$. Define functions $g_S{:}\mathbf{R} \times \mathbf{R} \to \mathbf{R}$ and $h_S{:}\mathbf{R} \times \mathbf{R} \to \mathbf{R}$ by:

$\forall\downarrow(\perp,...,[x, y],...,\perp) \in U_S$, $D(\downarrow(\perp,...,[x, y],...,\perp)) = \downarrow(\perp,...,[g_S(x, y), h_S(x, y)],...,\perp) \in V_d$.

Since $D(\{(\perp,...,\perp)\}) = \{(\perp,...,\perp)\}$ and $D$ is injective, $D$ maps intervals in $I_S$ to intervals in $I_d$, so $g_S(x, y)$ and $h_S(x, y)$ are defined for all $z$. Also define functions $g'_S{:}\mathbf{R} \to \mathbf{R}$ and $h'_S{:}\mathbf{R} \to \mathbf{R}$ by $g'_S(z) = g_S(z, z)$ and $h'_S(z) = h_S(z, z)$.

In Prop. G.4 we show how the functions $g_S$ and $h_S$ can be defined in terms of the functions $g'_S$ and $h'_S$.

**Prop. G.4.** Given a display function $D{:}U \to V$, a continuous scalar $s \in S$, and $[x, y] \in I_S$, then $g_S(x, y) = inf\{g'_S(z) \mid x \le z \le y\}$ and $h_S(x, y) = sup\{h'_S(z) \mid x \le z \le y\}$.

**Proof.** By Prop. G.3, $D(\downarrow(\perp,...,[x, y],...,\perp)) = \bigcap\{D(\downarrow(\perp,...,[z, z],...,\perp)) \mid x \le z \le y\} = \bigcap\{\downarrow(\perp,...,[g'_S(z), h'_S(z)],...,\perp) \mid x \le z \le y\}$. By Prop. F.8 this is $\downarrow(\perp,...,[a, b],...,\perp)$ for some $a, b \in \mathbf{R}$. Define $A = \{[g'_S(z), h'_S(z)] \mid x \le z \le y\}$. Then $\forall[g'_S(z), h'_S(z)] \in A.\ a \le g'_S(z)$ and $\forall[g'_S(z), h'_S(z)] \in A.\ h'_S(z) \le b$, and, by Prop. G.2,

$D(\downarrow(\perp,...,[x, y],...,\perp)) = \downarrow(\perp,...,[a, b],...,\perp) = \downarrow(\perp,...,[inf\{g'_S(z) \mid x \le z \le y\}, sup\{h'_S(z) \mid x \le z \le y\}],...,\perp)$. ■

Next, we prove a two lemmas useful for studying the functions $g_S$ and $h_S$.

**Prop. G.5.** Given a display function $D{:}U \to V$, a continuous scalar $s \in S$, and a finite set $A \subseteq I_S \backslash \{\perp\}$, then

$g_S(inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}) = inf\{g_S(u, v) \mid [u, v] \in A\}$ and

$h_S(inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}) = sup\{h_S(u, v) \mid [u, v] \in A\}$.

**Proof.** Since $A$ is finite, $inf\{u \mid [u, v] \in A\}$ and $sup\{v \mid [u, v] \in A\}$ exist, so, by Prop. G.2, $\downarrow(\perp,...,[inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}],...,\perp) =$

$\bigcap\{\downarrow(\perp,...,[u, v],...,\perp) \mid [u, v] \in A\} = \bigwedge\{\downarrow(\perp,...,[u, v],...,\perp) \mid [u, v] \in A\}$. Let

$a = g_S(inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}$ and

$b = h_S(inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\})$. Then

$\downarrow(\perp,...,[a, b],...,\perp) =$

$D(\downarrow(\perp,...,[inf\{u \mid [u, v] \in A\}, sup\{v \mid [u, v] \in A\}],...,\perp)) =$

$\bigwedge\{D(\downarrow(\perp,...,[u, v],...,\perp)) \mid [u, v] \in A\} =$

$\bigcap\{\downarrow(\perp,...,[g_S(u, v), h_S(u, v)],...,\perp) \mid [u, v] \in A\} =$ (by Prop. G.2)

$\downarrow(\perp,...,[inf\{g_S(u, v) \mid [u, v] \in A\}, sup\{h_S(u, v) \mid [u, v] \in A\}],...,\perp)$, so

$a = inf\{g_S(u, v) \mid [u, v] \in A\}$ and $b = sup\{h_S(u, v) \mid [u, v] \in A\}$. ■

**Prop. G.6.** Given a display function $D{:}U \to V$ and a continuous scalar $s \in S$, then

$[a, b] \subset [x, y] \Leftrightarrow [g_S(a, b), h_S(a, b)] \subset [g_S(x, y), h_S(x, y)]$.

**Proof.** $[a, b] \subset [x, y] \Leftrightarrow \downarrow[a, b] > \downarrow[x, y] \Leftrightarrow$

$D(\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp)) > D(\downarrow(\perp,...,[g_S(x, y), h_S(x, y)],...,\perp)) \Leftrightarrow$

$[g_S(a, b), h_S(a, b)] \subset [g_S(x, y), h_S(x, y)]$. ■

Now we show that the overall behavior of a display function on a continuous scalar must fall into one of two categories.

**Prop. G.7.** Given a display function $D{:}U \to V$ and a continuous scalar $s \in S$, then either

(a) $\forall x, y, z \in \mathbf{R}.\ x < y < z$ implies that $g_S(x, z) = g_S(x, y)$ & $h_S(x, y) < h_S(x, z)$ and that

$g_S(x, z) < g_S(y, z)$ & $h_S(y, z) = h_S(x, z)$,

or

(b) $\forall x, y, z \in \mathbf{R}.\ x < y < z$ implies that $g_S(x, z) < g_S(x, y)$ & $h_S(x, y) = h_S(x, z)$ and that

$g_S(x, z) = g_S(y, z)$ & $h_S(y, z) < h_S(x, z)$.

**Proof.** Let $x < y < z$. Then, by Prop. G.5, $g_S(x, z) = \min\{g_S(x, y), g_S(y, z)\}$ and $h_S(x, z) = \max\{h_S(x, y), h_S(y, z)\}$. If $g_S(x, z) < g_S(x, y)$ and $h_S(x, y) < h_S(x, z)$ then $g_S(y, z) = g_S(x, z)$ and $h_S(y, z) = h_S(x, z)$, so $[g_S(x, y), h_S(x, y)] \subset [g_S(y, z), h_S(y, z)]$ and by Prop. G.6, $[x, y] \subset [y, z]$, which is impossible. Thus either $g_S(x, y) = g_S(x, z)$ or $h_S(x, y) = h_S(x, z)$. However, both equalities cannot hold, since $\downarrow(\perp,...,[g_S(x, y), h_S(x, y)],...,\perp) = \downarrow(\perp,...,[g_S(x, z), h_S(x, z)],...,\perp) \Rightarrow$ $\downarrow(\perp,...,[x, y],...,\perp) = \downarrow(\perp,...,[x, z],...,\perp)$, which is impossible. Thus $g_S(x, z) = g_S(x, y)$ & $h_S(x, y) < h_S(x, z)$ or $g_S(x, z) < g_S(x, y)$ & $h_S(x, y) = h_S(x, z)$. A similar argument applies to the relation between $[y, z]$ and $[x, z]$, so $g_S(x, z) = g_S(y, z)$ & $h_S(y, z) < h_S(x, z)$ or $g_S(x, z) < g_S(y, z)$ & $h_S(y, z) = h_S(x, z)$.

Since $g_S(x, z) = \min\{g_S(x, y), g_S(y, z)\}$ and $h_S(x, z) = \max\{h_S(x, y), h_S(y, z)\}$, if $g_S(x, z) = g_S(x, y)$ then $h_S(x, y) < h_S(x, z)$ so $h_S(x, z) = h_S(y, z)$, and if $g_S(x, z) = g_S(y, z)$ then $h_S(y, z) < h_S(x, z)$ so $h_S(x, z)=h_S(x, y)$. Thus, for all $x, y, z \in \mathbf{R}$, $x < y < z$ implies that

(c) $g_S(x, z) = g_S(x, y)$ & $h_S(x, y) < h_S(x, z)$ and $g_S(x, z) < g_S(y, z)$ & $h_S(y, z) = h_S(x, z)$,

or

(d) $g_S(x, z) < g_S(x, y)$ & $h_S(x, y) = h_S(x, z)$ and $g_S(x, z) = g_S(y, z)$ & $h_S(y, z) < h_S(x, z)$.

We need to show that either (c) is true for all $x < y < z$, or that (d) is true for all $x < y < z$.

Now let $x < y < z < w$. Apply (c) and (d) to $x < y < z$ and $x < z < w$, but assume that (c) applies in one case and that (d) applies in the other case. That is, assume that

$g_S(x, w) = g_S(x, z) < g_S(x, y)$ and $h_S(x, y) = h_S(x, z) < h_S(x, w)$, or that

$g_S(x, w) < g_S(x, z) = g_S(x, y)$ and $h_S(x, y) < h_S(x, z) = h_S(x, w)$. Under both of these assumptions, $g_S(x, w) < g_S(x, y)$ and $h_S(x, y) < h_S(x, w)$, which is impossible (applying the result of the previous paragraph to $x < y < w$). Thus either (c) applies to both $x < y < z$ and $x < z < w$, or (d) applies to both $x < y < z$ and $x < z < w$.

Similarly, apply (c) and (d) to $x < y < w$ and $y < z < w$, but assume that (c) applies in one case and that (d) applies in the other case. That is, assume that

$g_S(x, w) < g_S(y, w) = g_S(z, w)$ and $h_S(z, w) < h_S(y, w) = h_S(x, w)$, or that

$g_S(x, w) = g_S(y, w) < g_S(z, w)$ and $h_S(z, w) = h_S(y, w) < h_S(x, w)$. Under both of these assumptions, $g_S(x, w) < g_S(z, w)$ and $h_S(z, w) < h_S(x, w)$, which is impossible (applying the result of the previous paragraph to $x < z < w$). Thus either (c) applies to both $x < y < w$ and $y < z < w$, or (d) applies to both $x < y < w$ and $y < z < w$.

Now let $x < y < z < x' < y' < z'$. The results of the last two paragraphs can be applied to show that (c) and (d) are applied consistently to the following chain of triples:

$x < y < z$

$x < y < x'$

$x < x' < y'$

$x < y' < z'$

$y < y' < z'$

$z < y' < z'$

$x' < y' < z'$.

Thus either (c) applies to both $x < y < z$ and $x' < y' < z'$, or (d) applies to both $x < y < z$ and $x' < y' < z'$.

Given any two triples $x < y < z$ and $x' < y' < z'$, pick $x'' < y'' < z''$ with $z < x''$ and

$z' < x''$. Then $x < y < z < x'' < y'' < z''$ and $x' < y' < z' < x'' < y'' < z''$ so either (c) or (d) applies uniformly to the triples $x < y < z$, $x'' < y'' < z''$ and $x' < y' < z'$. Thus either (c) or (d) applies uniformly to all triples, proving the proposition. ■

Next we define names for the two categories established in Prop. G.7.

**Def.** Given a display function $D:U \to V$ and a continuous scalar $s \in S$, by Prop. G.7, either (a) or (b) is applies to all triples $x < y < z$. If (a) applies, say that $D$ is *increasing* on $s$, and if (b) applies, say that $D$ is *decreasing* on $s$.

Prop. G.8 is useful for showing how the categories established in Prop. G.7 apply to the functions $g'_S$ and $h'_S$.

**Prop. G.8.** Given a display function $D:U \to V$, a continuous scalar $s \in S$, $z \in \mathbf{R}$, and a set $A \subseteq I_S \backslash \{\perp\}$ such that $[z, z] = \bigcap A$, then

$g'_S(a) = sup\{g_S(a, b) \mid [a, b] \in A\}$ and

$h'_S(a) = inf\{h_S(a, b) \mid [a, b] \in A\}$.

**Proof.**

$\downarrow(\perp,...,[z, z],...,\perp) = \{(\perp,...,[u, v],...,\perp) \mid u \le z \le v\} =$

$\{(\perp,...,[u, v],...,\perp) \mid \exists [a, b] \in A.\ u \le a \le b \le v\} =$

$\bigcup\{\downarrow(\perp,...,[a, b],...,\perp) \mid [a, b] \in A\}$. This union of closed sets is closed (since it equals $\downarrow(\perp,...,[z, z],...,\perp)$), so, by Prop. C.8,

$\downarrow(\perp,...,[z, z],...,\perp) = \bigvee\{\downarrow(\perp,...,[a, b],...,\perp) \mid [a, b] \in A\}$. Then, by Prop. B.3,

$D(\downarrow(\perp,...,[z, z],...,\perp)) = \bigvee\{D(\downarrow(\perp,...,[a, b],...,\perp)) \mid [a, b] \in A\} =$

$\bigvee\{\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \mid [a, b] \in A\}$. Therefore

$\downarrow(\perp,...,[g'_S(a), h'_S(a)],...,\perp) = \bigvee\{\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \mid [a, b] \in A\}$. Thus

$\forall [a, b] \in A.\ \downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \leq \downarrow(\perp,...,[g'_S(a), h'_S(a)],...,\perp)$, so

$\forall [a, b] \in A.\ g_S(a, b) \leq g'_S(a) \leq h'_S(a) \leq h_S(a, b)$. Therefore

$sup\{g_S(a, b) \mid [a, b] \in A\} \leq g'_S(a)$ and $h'_S(a) \leq inf\{h_S(a, b) \mid [a, b] \in A\}$.

Now assume that $sup\{g_S(a, b) \mid [a, b] \in A\} < g'_S(a)$ and pick $u$ such that

$sup\{g_S(a, b) \mid [a, b] \in A\} < u < g'_S(a)$. Then for all $[a, b] \in A$, $g_S(a, b) < u$ so

$\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \leq \downarrow(\perp,...,[u, h'_S(a)],...,\perp)$. Therefore

$\bigvee\{\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \mid [a, b] \in A\} \leq$

$\downarrow(\perp,...,[u, h'_S(a)],...,\perp) < \downarrow(\perp,...,[g'_S(a), h'_S(a)],...,\perp)$,

which contradicts

$\bigvee\{\downarrow(\perp,...,[g_S(a, b), h_S(a, b)],...,\perp) \mid [a, b] \in A\} = \downarrow(\perp,...,[g'_S(a), h'_S(a)],...,\perp)$. Thus

$g'_S(a) = sup\{g_S(a, b) \mid [a, b] \in A\}$. A similar argument shows that

$h'_S(a) = inf\{h_S(a, b) \mid [a, b] \in A\}$. ■

Now we show how the categories of behavior established in Prop. G.7 apply to the functions $g'_S$ and $h'_S$.

**Prop. G.9.** Given a display function $D{:}U \rightarrow V$, a continuous scalar $s \in S$, and $z < z'$, if $D$ is increasing on $s$ then $g'_S(z) < g'_S(z')$ and $h'_S(z) < h'_S(z')$, and if $D$ is decreasing on $s$ then $g'_S(z) > g'_S(z')$ and $h'_S(z) > h'_S(z')$.

**Proof.** First assume that $D$ is increasing on $s$. Then, by Prop. G.8, $g'_S(z) = sup\{g_S(z, x) \mid z < x\}$. By Prop. G.7, $\forall x > z.\ \forall y > z.\ g_S(z, x) = g_S(z, y)$, so $\forall x > z.\ g'_S(z) = g_S(z, x)$. Similarly, $\forall x > z'.\ g'_S(z') = g_S(z', x)$. Pick $x > z' > z$. Then, by Prop. G.7, $g'_S(z) = g_S(z, x) < g_S(z', x) = g'_S(z')$.

By Prop. G.8, $h'_S(z) = inf\{h_S(x, z) \mid x < z\}$. By Prop. G.7,

$\forall x < z.\ \forall y < z.\ h_S(x, z) = h_S(y, z)$, so $\forall x < z.\ h'_S(z) = h_S(x, z)$. Similarly,

$\forall x < z'.\ h'_S(z') = h_S(x, z')$. Pick $x < z < z'$. Then, by Prop. G.7,

$h'_S(z) = h_S(x, z) < h_S(x, z') = h'_S(z')$.

Next assume that $D$ is decreasing on $s$. Then, by Prop. G.8,

$g'_S(z) = sup\{g_S(x, z) \mid x < z\}$. By Prop. G.7, $\forall x < z.\ \forall y < z.\ g_S(x, z) = g_S(y, z)$, so

$\forall x < z.\ g'_S(z) = g_S(x, z)$. Similarly, $\forall x < z'.\ g'_S(z') = g_S(x, z')$. Pick $x < z < z'$. Then, by Prop. G.7, $g'_S(z) = g_S(x, z) > g_S(x, z') = g'_S(z')$.

By Prop. G.8, $h'_S(z) = inf\{h_S(z, x) \mid z < x\}$. By Prop. G.7,

$\forall x > z.\ \forall y > z.\ h_S(z, x) = h_S(z, y)$, so $\forall x > z.\ h'_S(z) = h_S(z, x)$. Similarly,

$\forall x > z'.\ h'_S(z') = h_S(z', x)$. Pick $x > z' > z$. Then, by Prop. G.7,

$h'_S(z) = h_S(z, x) > h_S(z', x) = h'_S(z')$. ■

Next we show that the functions $g'_S$ and $h'_S$ must be continuous functions of real variables. The key idea is that $g'_S$ and $h'_S$ are either increasing or decreasing, so if they are discontinuous there must be a gap in their values, which contradicts Prop. B.2.

**Prop. G.10.** Given a display function $D{:}U \rightarrow V$ and a continuous scalar $s \in S$, the functions $g'_S$ and $h'_S$ are continuous (in the topological sense).

**Proof.** Assume that $D$ is increasing on $s$. Then, by Prop. G.9, $g'_S$ and $h'_S$ are monotone increasing. Now assume that $g'_S$ is discontinuous at $z$. Then

(a) $\exists \varepsilon > 0.\ \forall \delta > 0.\ \exists w.$

$z - \delta < w < z\ \&\ g'_S(w) \leq g'_S(z) - \varepsilon$ or

$z < w < z + \delta\ \&\ g'_S(z) + \varepsilon \leq g'_S(w)$

Fix $\varepsilon$ satisfying (5). If

(b) $\exists w_{_}.\ (w_{_} < z\ \&\ g'_S(z) - \varepsilon < g'_S(w_{_}))$

then

(c) $\forall w.\ w_- < w < z \Rightarrow g'_S(z) - \varepsilon < g'_S(w) < g'_S(z)$

and if

(d) $\exists w_+.\ (z < w_+ \ \& \ g'_S(w_+) < g'_S(z) + \varepsilon)$

then

(e) $\forall w.\ z < w < w_+ \Rightarrow g'_S(z) < g'_S(w) < g'_S(z) + \varepsilon.$

Now, ((c) & (e)) contradicts (a), so (¬(b) or ¬(d)).

$\neg$(b) ≡ $\forall w.\ w < z \Rightarrow g'_S(w) \leq g'_S(z) - \varepsilon$

and

$\neg$(d) ≡ $\forall w.\ z < w \Rightarrow g'_S(z) + \varepsilon \leq g'_S(w).$

In the ¬(b) case, since $z \leq w \Rightarrow g'_S(z) \leq g'_S(w)$, there is no $w \in \mathbf{R}$ such that $g'_S(z) - \varepsilon < g'_S(w) < g'_S(z)$. Now, $[g'_S(z), h'_S(z)] \subset [g'_S(z) - \varepsilon/2, h'_S(z)]$ so $\downarrow(\perp,...,[g'_S(z) - \varepsilon/2, h'_S(z)],...,\perp) \leq \downarrow(\perp,...,[g'_S(z), h'_S(z)],...,\perp)$. Thus, by Prop. B.2, there is $u \in U$ such that $D(u) = \downarrow(\perp,...,[g'_S(z) - \varepsilon/2, h'_S(z)],...,\perp)$, and by Prop. F.9 and Prop. F.10, $u \in U_S$. Let $u = \downarrow(\perp,...,[a, b],...,\perp)$. Then, by Prop. G.4, $g'_S(z) - \varepsilon/2 = g_S(a, b) = inf\{g'_S(w) \mid a \leq w \leq b\}$. However, since there is no $w$ such that $g'_S(z) - \varepsilon < g'_S(w) < g'_S(z)$, this is impossible. Thus $g'_S$ cannot be discontinuous at $z$.

In the ¬(d) case, since $w \leq z \Rightarrow g'_S(w) \leq g'_S(z)$, there is no $w \in \mathbf{R}$ such that $g'_S(z) < g'_S(w) < g'_S(z) + \varepsilon$, and furthermore, $z < z' \Rightarrow g'_S(z) < g'_S(z')$, so there is $z'$ such that $g'_S(z) + \varepsilon \leq g'_S(z')$. Now, $[g'_S(z'), h'_S(z')] \subset [g'_S(z) + \varepsilon/2, h'_S(z')]$ so $\downarrow(\perp,...,[g'_S(z) + \varepsilon/2, h'_S(z')],...,\perp) \leq \downarrow(\perp,...,[g'_S(z'), h'_S(z')],...,\perp)$. Thus, by Prop. B.2, there is $u \in U$ such that $D(u) = \downarrow(\perp,...,[g'_S(z) + \varepsilon/2, h'_S(z')],...,\perp)$, and by Prop. F.9 and Prop. F.10, $u \in U_S$. Let $u = \downarrow(\perp,...,[a, b],...,\perp)$. Then, by Prop. G.4, $g'_S(z) + \varepsilon/2 = g_S(a, b) = inf\{g'_S(w) \mid a \leq w \leq b\}$. However, since there is no $w$ such that $g'_S(z) < g'_S(w) < g'_S(z) + \varepsilon$, this is impossible. Thus $g'_S$ cannot be discontinuous at $z$.

The proof that $h'_S$ is continuous, and the proofs that $g'_S$ and $h'_S$ are continuous when $D$ is decreasing on $s$, are virtually identical to this. ■

Prop. G.11 completes the list of conditions on the functions $g'_S$ and $h'_S$ that will allow us to define necessary and sufficient conditions for display functions.

**Prop. G.11.** Given a display function $D{:}U \to V$ and a continuous scalar $s \in S$, then $g'_S$ has no lower bound and $h'_S$ has no upper bound. Furthermore,

$\forall z \in \mathbf{R}.\ g'_S(z) \le h'_S(z)$.

**Proof.** If $\exists a.\ \forall z.\ g'_S(z) > a$ then,

$D(\downarrow(\bot,...,[0, 0],...,\bot)) = \downarrow(\bot,...,[g'_S(0), h'_S(0)],...,\bot) \ge \downarrow(\bot,...,[a\text{-}1, h'_S(0)],...,\bot)$

[since $a\text{-}1 < a \le g'_S(0)$], so there must be $u \in U$ such that

$D(u) = \downarrow(\bot,...,[a\text{-}1, h'_S(0)],...,\bot)$. By Prop. F.9 and Prop. F.10, $u \in U_S$. However, by Prop. G.4, there is no $[x, y] \in I_S$ such that

$D(\downarrow(\bot,...,[x, y],...,\bot)) = \downarrow(\bot,...,[a\text{-}1, h'_S(0)],...,\bot)$. Thus $g'_S$ has no lower bound. The proof that $h'_S$ has no upper bound is virtually identical.

If $g'_S(z) > h'_S(z)$ then $[g'_S(z), h'_S(z)] \notin I_S$, which is impossible, so

$\forall z \in \mathbf{R}.\ g'_S(z) \le h'_S(z)$. ■

The results of this appendix can be summarized in the following definition.

**Def.** A pair of functions $g'_S{:}\mathbf{R} \to \mathbf{R}$ and $h'_S{:}\mathbf{R} \to \mathbf{R}$ are called a *continuous display pair* if:

(a) $g'_S$ has no lower bound and $h'_S$ has no upper bound,

(b) $\forall z \in \mathbf{R}.\ g'_S(z) \le h'_S(z)$, and

(c) $g'_S$ and $h'_S$ are continuous,

(d) either $g'_S$ and $h'_S$ are increasing:

$\forall z, z' \in \mathbf{R}.\ z < z' \Rightarrow g'_S(z) < g'_S(z') \ \&\ h'_S(z) < h'_S(z')$,

or $g'_S$ and $h'_S$ are decreasing:

$\forall z, z' \in \mathbf{R}.\ z < z' \Rightarrow g'_S(z) > g'_S(z') \ \&\ h'_S(z) > h'_S(z')$.

# Appendix H

## Proofs for Section 3.4.3

Here we present the technical details for Section 3.4.3.

**Def.** Given a finite set $S$ of scalars, a finite set $DS$ of display scalars, $X = \mathbf{X}\{I_s \mid s \in S\}$, $Y = \mathbf{X}\{I_d \mid d \in DS\}$, $U = CL(\mathrm{X})$, and $V = CL(\mathrm{Y})$, then a function $D{:}U \to V$ is a *scalar mapping function* if:

(a) there is a function $MAP_D{:}\ S \to POWER(DS)$ such that

$\forall s, s' \in S.\ MAP_D(s) \cap MAP_D(s') = \phi$,

(b) for all continuous $s \in S$, $MAP_D(s)$ contains a single continuous $d \in DS$,

(c) for all discrete $s \in S$, all $d \in MAP_D(s)$ are discrete,

(d) $D(\phi) = \phi$ and $D(\{(\perp,...,\perp)\}) = \{(\perp,...,\perp)\}$,

(e) for all continuous $s \in S$, $g'_S$ and $h'_S$ are a continuous display pair,

for all $[u, v] \in I_S$, $g_S(u, v) = inf\{g'_S(z) \mid u \le z \le v\}$ and

$h_S(u, v) = sup\{h'_S(z) \mid u \le z \le v\}$,

and, given $\{d\} = MAP_D(s)$, then for all $[u, v] \in I_S\backslash\{\perp\}$,

$D(\downarrow(\perp,...,[u, v],...,\perp)) = \downarrow(\perp,...,[g_S(u, v), h_S(u, v)],...,\perp) \in V_d$,

(f) for all discrete $s \in S$, for all $a \in I_S\backslash\{\perp\}$,

$D(\downarrow(\perp,...,a,...,\perp)) = \mathrm{b} \in V_d$ for some $d \in MAP_D(s)$, where $b \ne \{(\perp,...,\perp)\}$,

and, for all $a, a' \in I_S\backslash\{\perp\}$, $a \ne a' \Rightarrow D(\downarrow(\perp,...,a,...,\perp)) \ne D(\downarrow(\perp,...,a',...,\perp))$

(g) for all $x \in X$, $D(\downarrow x) = \downarrow\bigvee\{y \mid \exists s \in S.\ x_S \ne \perp\ \&\ \downarrow y = D(\downarrow(\perp,...,x_S,...,\perp))\}$,

where $x_S$ represents tuple components of $x$, and using the values for $D$ defined in (e) and (f),

(h) for all $u \in U$, $D(u) = \bigvee\{D(\downarrow x) \mid x \in u\}$, using the values for $D$ defined in (g).

This definition contains a variety of expressions for the value of $D$ on various subsets of $U$. The next proposition shows that these expressions are consistent where the subsets of $U$ overlap. This involves showing that $D$ is monotone.

**Prop. H.1.** In the definition of scalar mapping functions, the values defined for $D$ in (d), (e), (f), (g) and (h) are consistent. Furthermore, $D$ is monotone.

**Proof.** (e), (f), (g) and (h) do not apply to $\phi$ and thus do not conflict with the definition of $D(\phi)$ in (d). (e) and (f) do not apply to $\{(\perp,...,\perp)\}$ and thus do not conflict with the definition of $D(\{(\perp,...,\perp)\})$ in (d). The definition of $D(\{(\perp,...,\perp)\})$ in (d) is consistent with (g) and (h) if the *sup* of an empty set of objects is defined as $(\perp,...,\perp)$. (e) and (f) apply to disjoint sets and thus do not conflict. For all $s \in S$, (g) applies to objects
$x \in U_S$, but defines $D(\downarrow x)$ as the *sup* of the singleton set containing the value of $D(\downarrow x)$ defined by (e) or (f), and is thus consistent with that value. (h) applies to objects $x \in U_S$, and is consistent with (e) and (f) if it is consistent with (g) on these objects. Thus we need to show the consistency of (g) and (h).

If $u = \downarrow y$ then (h) defines $D(\downarrow y) = \bigvee\{D(\downarrow x) \mid x \in \downarrow y\} = \bigvee\{D(\downarrow x) \mid x \leq y\}$. To Show consistency with (g), it is necessary to show that $x \leq y \Rightarrow D(\downarrow x) \leq D(\downarrow y)$ for the definition of $D$ in (d), (e), (f) and (g) (that is, that $D$ is monotone). Clearly $D$ in (d) is monotone, in itself and in relation to $D$ in (e), (f) and (g). If $s \in S$ is discrete, then for all $a, a' \in I_S\backslash\{\perp\}$, $a \neq a' \Rightarrow \neg(a \leq a')$, so $D$ in (f) is monotone by default. If $s \in S$ is continuous then for all $[u, v], [u', v'] \in I_S\backslash\{\perp\}$,

$\downarrow(\perp,...,[u, v],...,\perp) \leq \downarrow(\perp,...,[u', v'],...,\perp) \Rightarrow$

$[u', v'] \subseteq [u, v] \Rightarrow$

$[inf\{g'_S(z) \mid u' \leq z \leq v'\}, sup\{h'_S(z) \mid u' \leq z \leq v'\}] \subseteq$

$[inf\{g'_S(z) \mid u \leq z \leq v\}, sup\{h'_S(z) \mid u \leq z \leq v\}] \Rightarrow$

$D(\downarrow(\bot,...,[u, v],...,\bot)) \leq D(\downarrow(\bot,...,[u', v'],...,\bot))$.

Thus $D$ in (e) is monotone. For all $x, x' \in X$,

$x \leq x' \Rightarrow$

$\forall s \in S.\ x_s \leq x_s' \Rightarrow$ (since $D$ in (e) and (f) is monotone)

$\forall s \in S.\ D(\downarrow(\bot,...,x_s,...,\bot)) \leq D(\downarrow(\bot,...,x'_s,...,\bot)) \Rightarrow$

$D(\downarrow x) \leq D(\downarrow x')$.

Thus $D$ in (g) is monotone, so $D$ is consistent in (g) and (h).

All that remains is to show that $D$ in (h) is monotone. For all $u, u' \in U$,

$u \leq u' \Rightarrow u \subseteq u'$ so $\bigvee\{D(\downarrow x) \mid x \in u\} \leq \bigvee\{D(\downarrow x) \mid x \in u'\}$. Thus $D$ is monotone. ■

As we will show in Prop. H.5, the values of a scalar mapping function $D$ can be decomposed into the values of an auxiliary function $D'$ from $X$ to $Y$. Now we define this auxiliary function, show that it is an order embedding, and prove two lemmas that will be useful in the proof of Prop. H.5.

**Def.** Given a scalar mapping function $D{:}U \to V$, define $D'{:}X \to Y$ by

$D'(x) = \bigvee\{(\bot,...,a_d,...,\bot) \mid s \in S \ \&\ x_s \neq \bot \ \&\ D(\downarrow(\bot,...,x_s,...,\bot)) = \downarrow(\bot,...,a_d,...,\bot)\}$.

**Prop. H.2.** Given a scalar mapping function $D{:}U \to V$, $D'$ is an order embedding.

**Proof.** Given $x, x' \in X$, $x \leq x' \Leftrightarrow \forall s \in S.\ x_s \leq x'_s$. Let

$D'((\bot,...,x_s,...,\bot)) = (\bot,...,a_d,...,\bot)$ and $D'((\bot,...,x'_s,...,\bot)) = (\bot,...,a'_d,...,\bot)$ where

$d \in MAP_D(s)$. Note that $x_s \leq x'_s \Rightarrow (\bot,...,x_s,...,\bot) \leq (\bot,...,x'_s,...,\bot) \Rightarrow$

$(\bot,...,a_d,...,\bot) \leq (\bot,...,a'_d,...,\bot)$ (since a $D$ is monotone) so $a_d$ and $a'_d$ are in the same $I_d$.

For all $s \in S$, $x_s \leq x'_s \Leftrightarrow (\bot,...,x_s,...,\bot) \leq (\bot,...,x'_s,...,\bot) \Leftrightarrow$

$\downarrow(\bot,...,a_d,...,\bot) = D(\downarrow(\bot,...,x_s,...,\bot)) \leq D(\downarrow(\bot,...,x'_s,...,\bot)) = \downarrow(\bot,...,a'_d,...,\bot) \Leftrightarrow$

$(\bot,...,a_d,...,\bot) \leq (\bot,...,a'_d,...,\bot) \Leftrightarrow a_d \leq a'_d$. Thus

$(\forall s \in S.\ x_S \le x'_S) \Leftrightarrow (\forall d \in DS.\ a_d \le a'_d)$. Since $\forall s, s' \in S.\ MAP_D(s) \cap MAP_D(s') = \phi$,

$c = D'(x) \Rightarrow (\forall d \in DS.\ c_d \ne \perp \Rightarrow \exists s \in S.\ D(\downarrow(\perp,...,x_S,...,\perp)) = \downarrow(\perp,...,c_d,...,\perp))$

(that is, $c_d = a_d$), and thus $(\forall d \in DS.\ a_d \le a'_d) \Leftrightarrow D'(x) \le D'(x')$. Therefore, by a chain of logical equivalences, $x \le x' \Leftrightarrow D'(x) \le D'(x')$. ■

**Prop. H.3.** Let $D{:}U \to V$ be a scalar mapping function. Then, for all $u \in U$, $x \in u$ and $b \le D'(x) = a$, there is $y \le x$ such that $b = D'(y)$.

**Proof.** For all $d \in DS$, $b_d \ne \perp$ implies that

$\exists s \in S.\ D'((\perp,...,x_S,...,\perp)) = (\perp,...,a_d,...,\perp)$ and $b_d \le a_d$. For discrete $s$,

$b_d \le a_d$ & $b_d \ne \perp \Rightarrow b_d = a_d$. Thus $D'((\perp,...,x_S,...,\perp)) = (\perp,...,b_d,...,\perp)$. Let $y_S = x_S$.

For continuous $s$, let $a_d = [inf\{g'_S(z) \mid u \le z \le v\}, sup\{h'_S(z) \mid u \le z \le v\}]$ where $x_S = [u, v]$. There are $e, f \in \mathbf{R}$ such that $b_d = [e, f]$ where

$e \le inf\{g'_S(z) \mid u \le z \le v\} \le sup\{h'_S(z) \mid u \le z \le v\} \le f$.

Since $g'_S$ is continuous and has no lower bound, $\exists u'.\ g'_S(u') = e$, and since $h'_S$ is continuous and has no upper bound, $\exists v'.\ h'_S(v') = f$. Now $g'_S$ and $h'_S$ are either increasing or decreasing.

If $g'_S$ and $h'_S$ are increasing then $u' \le u$ and $v \le v'$, so $e = inf\{g'_S(z) \mid u' \le z \le v'\}$ [since $u' \le z \Rightarrow g'_S(u') \le g'_S(z)$] and $f = sup\{h'_S(z) \mid u' \le z \le v'\}$ [since $z \le v' \Rightarrow h'_S(z) \le h'_S(v')$]. Then

$b_d = [e, f] = [inf\{g'_S(z) \mid u' \le z \le v'\}, sup\{h'_S(z) \mid u' \le z \le v'\}]$ and

$D'((\perp,...,[u', v'],...,\perp)) = (\perp,...,b_d,...,\perp)$. Let $y_S = [u', v']$.

If $g'_S$ and $h'_S$ are decreasing then $v' \le u$ and $v \le u'$, so $e = inf\{g'_S(z) \mid v' \le z \le u'\}$ [since $z \le u' \Rightarrow g'_S(u') \le g'_S(z)$] and $f = sup\{h'_S(z) \mid v' \le z \le u'\}$ [since $v' \le z \Rightarrow h'_S(z) \le h'_S(v')$]. Then

$b_d = [e, f] = [inf\{g'_S(z) \mid v' \le z \le u'\}, sup\{h'_S(z) \mid v' \le z \le u'\}]$ and

$D'((\perp,...,[v', u'],...,\perp)) = (\perp,...,b_d,...,\perp)$. Let $y_S = [v', u']$.

Thus for all $d \in DS$ such that $b_d \neq \perp$, there is $y_S \leq x_S$ such that $D\curlyvee((\perp,...,y_S,...,\perp)) = (\perp,...,b_d,...,\perp)$. For any $s \in S$ such that $y_S$ is not determined by any $b_d$, set $y_S = \perp$. Then $D\curlyvee(y) = b$. ■

**Prop. H.4.** Given a scalar mapping function $D{:}U \rightarrow V$, and a directed set $M \subseteq X$, $D\curlyvee(\bigvee M) = \bigvee D\curlyvee(M)$.

**Proof.** Given a directed set $M \subseteq X$, let $x = \bigvee M$ and y = $D\curlyvee(x)$. Since $D'$ is an order embedding, $D\curlyvee(M)$ is directed so $z = \bigvee D\curlyvee(M)$ exists. Also, $\forall m \in M.\ m \leq x$, so $\forall m \in M.\ D\curlyvee(m) \leq y$ and thus $z \leq y$. For all $d \in DS$, if $y_d \neq \perp$ then there is $s \in S$ such that $\downarrow(\perp,...,y_d,...,\perp) = D(\downarrow(\perp,...,x_S,...,\perp))$, and so $(\perp,...,y_d,...,\perp) = D\curlyvee((\perp,...,x_S,...,\perp))$. Since *sups* are taken componentwise in $X$, $x_S = \bigvee\{m_S \mid m \in M\}$.

If $s$ is discrete, then $\exists m \in M.\ x_S = m_S$ so $(\perp,...,y_d,...,\perp) = D\curlyvee((\perp,...,m_S,...,\perp)) \leq D\curlyvee(m) \leq z$, and thus $y_d \leq z_d$. Since $z \leq y$, and thus $z_d \leq y_d$, this gives $y_d = z_d$.

If $s$ is continuous, then $x_S = [u, v]$ and $m_S = [u_m, v_m]$ are real intervals (we adopt the convention that $u_m = -\infty$ and $v_m = \infty$ for $m_S = \perp$). Then $[u, v]$ is the intersection of the $[u_m, v_m]$, for all $m \in M$, so $u = sup\{u_m \mid m \in M\}$ and $v = inf\{v_m \mid m \in M\}$ and thus $y_d = [a, b] = [inf\{g'_S(z) \mid u \leq z \leq v\}, sup\{h'_S(z) \mid u \leq z \leq v\}]$. Also let $z_d = [e, f]$.

Then, since $MAP_D(s)$ contains only $d$,

$e = sup\{inf\{g'_S(z) \mid u_m \leq z \leq v_m\} \mid m \in M\}$ and

$f = inf\{sup\{h'_S(z) \mid u_m \leq z \leq v_m\} \mid m \in M\}$.

If $g'_S$ and $h'_S$ are increasing then, since they are continuous,

$a = inf\{g'_S(z) \mid sup\{u_m \mid m \in M\} \leq z \leq inf\{v_m \mid m \in M\}\} = g'_S(sup\{u_m \mid m \in M\}) =$

$sup\{g'_S(u_m) \mid m \in M\} = sup\{inf\{g'_S(z) \mid u_m \leq z \leq v_m\} \mid m \in M\} = e$ and

$b = sup\{h'_S(z) \mid sup\{u_m \mid m \in M\} \leq z \leq inf\{v_m \mid m \in M\}\} = h'_S(inf\{v_m \mid m \in M\}) =$

$inf\{h'_S(v_m) \mid m \in M\} = inf\{sup\{h'_S(z) \mid u_m \le z \le v_m\} \mid m \in M\} = f.$

If $g'_S$ and $h'_S$ are decreasing then, since they are continuous,

$a = inf\{g'_S(z) \mid sup\{u_m \mid m \in M\} \le z \le inf\{v_m \mid m \in M\}\} = g'_S(inf\{v_m \mid m \in M\}) =$

$sup\{g'_S(v_m) \mid m \in M\} = sup\{inf\{g'_S(z) \mid u_m \le z \le v_m\} \mid m \in M\} = e$ and

$b = sup\{h'_S(z) \mid sup\{u_m \mid m \in M\} \le z \le inf\{v_m \mid m \in M\}\} = h'_S(sup\{u_m \mid m \in M\}) =$

$inf\{h'_S(u_m) \mid m \in M\} = inf\{sup\{h'_S(z) \mid u_m \le z \le v_m\} \mid m \in M\} = f.$

In either case, $y_d = [a, b] = [e, f] = z_d$.

Thus $y_d = z_d$ for all $d \in DS$ such that $y_d \neq \perp$. However, we also have $z \le y$ so $z_d = \perp$ whenever $y_d = \perp$, so $y_d = z_d$ for all $d \in DS$ and thus $y = z$. ■

Now we show how a scalar mapping function $D$ can be defined in terms of the auxiliary function $D'$.

**Prop. H.5.** Given a scalar mapping function $D{:}U \to V$, for all $u \in U$, $D(u) = \{D'(x) \mid x \in u\}$.

**Proof.** First, we show that for all $u \in U$, $u$ is closed $\Rightarrow \{D'(x) \mid x \in u\}$ is closed.

Assume $x \in u$ and $b \le D'(x)$. Then, by Prop. H.3, $\exists y \le x.\ b = D'(y)$. Further, $y \le x \Rightarrow y \in u$ so $b \in \{D'(x) \mid x \in u\}$. Now assume $N \subseteq \{D'(x) \mid x \in u\}$ and $N$ is directed. Then there is $M \subseteq u$ such that $N = D'(M)$, and, since $D'$ is an order embedding, $M$ is directed. Thus $\bigvee M \in u$ and, by Prop. H.4, $\bigvee N = D'(\bigvee M) \in \{D'(x) \mid x \in u\}$. Thus $\{D'(x) \mid x \in u\}$ is closed.

Second, we show that for all $x \in X$, $D(\downarrow x) = \{D'(y) \mid y \le x\}$. By (g) in the definition of scalar mapping functions, $\forall y \in X.\ \exists b \in Y.\ D(\downarrow y) = \downarrow b$. Furthermore, comparing (g) with the definition of $D'$, $\forall y \in X.\ D(\downarrow y) = \downarrow b \Leftrightarrow D'(y) = b$. Then, given $D(\downarrow x) = \downarrow a$, $b \le a \Leftrightarrow \downarrow b \le \downarrow a \Leftrightarrow \exists y \le x.\ D(\downarrow y) = \downarrow b \Leftrightarrow \exists y \le x.\ D'(y) = b$. Thus

$D(\downarrow x) = \downarrow a = \{b \mid b \leq a\} = \{D'(y) \mid y \leq x\}$.

By Prop. C.8, $\bigvee\{D(\downarrow x) \mid x \in u\}$ is the smallest closed set containing $\bigcup\{D(\downarrow x) \mid x \in u\}$. However,

$\bigcup\{D(\downarrow x) \mid x \in u\} = \bigcup\{\{D'(y) \mid y \leq x\} \mid x \in u\} = \{D'(x) \mid x \in u\}$, which is closed, so

$\bigvee\{D(\downarrow x) \mid x \in u\} = \bigcup\{D(\downarrow x) \mid x \in u\}$. Thus, for all $u \in U$,

$D(u) = \bigvee\{D(\downarrow x) \mid x \in u\} = \{D'(x) \mid x \in u\}$. ■

The next two propositions show that a scalar mapping function satisfies the conditions of a display function.

**Prop. H.6.** A scalar mapping function $D:U \rightarrow V$ is an order embedding (and thus injective).

**Proof.** By Prop. H.5, for all $u \in U$, $D(u) = \{D'(x) \mid x \in u\}$. Members of $U$ are ordered by set inclusion, so

$u \leq u' \Rightarrow u \subseteq u' \Rightarrow D(u) = \{D'(x) \mid x \in u\} \subseteq \{D'(x) \mid x \in u'\} = D(u') \Rightarrow D(u) \leq D(u')$.

By Prop. H.2, $D'$ is an order embedding, and thus injective, so $u = \{(D')^{-1}(x) \mid x \in D(u)\}$.

Therefore $D(u) \leq D(u') \Rightarrow D(u) \subseteq D(u') \Rightarrow$

$u = \{(D')^{-1}(x) \mid x \in D(u)\} \subseteq \{(D')^{-1}(x) \mid x \in D(u')\} = u' \Rightarrow u \leq u'$.

Thus $D$ is an order embedding. ■

**Prop. H.7.** A scalar mapping function $D:U \rightarrow V$ is a surjective function onto $\downarrow D(X)$.

**Proof.** Assume that $v' < v = D(X)$. We need to show that there is $u' \in U$ such that $v' = D(u')$. As we saw in the proof of Prop. H.6, if there is such a $u'$, then $u' = \{(D')^{-1}(x) \mid x \in v'\}$. Thus let $u' = \{(D')^{-1}(x) \mid x \in v'\}$, and we will show that this is a closed set, and thus a member of $U$.

Assume that $y \in u'$ and $b \leq y$. Then $D'(b) \leq D'(y)$, and since $D'(y) \in v'$ and $v'$ is closed, $D'(b) \in v'$ so $b \in u'$. Now assume that $N \subseteq u'$ and $N$ is directed. Then $M = D'(N) \subseteq v'$ is directed (since $D'$ is an order embedding), so $\bigvee M \in v'$ and $(D')^{-1}(\bigvee M) \in u'$. By Prop. H.4, $\bigvee M = D'(\bigvee N)$ so $\bigvee N = (D')^{-1}(\bigvee M) \in u'$. Thus $u'$ is closed. ■

The results of the last three sections show that display functions are completely characterized as scalar mapping functions. This is summarized by the following theorem.

**Theorem H.8.** $D: U \to V$ is a display function if and only if it is a scalar mapping function.

**Proof.** If $D: U \to V$ is a display function then Theorem F.14 shows that $D$ satisfies conditions (a), (b), (c) and (f) of the definition of scalar mapping functions. Theorem F.14, along with Props. G.4, G.9, G.10 and G.11 show that $D$ satisfies condition (e). Prop. F.2 shows that $D$ satisfies condition (d). Prop. F.12 shows that $D$ satisfies condition (g), and the proof of Prop. F.13 shows that $D$ satisfies condition (h). Thus $D$ is a scalar mapping function.

If $D: U \to V$ is a scalar mapping function then Props. H.6 and H.7 show that $D$ is a display function. ■

# Appendix I

## Proofs for Section 3.4.4

Here we present the technical details for Section 3.4.4. Define a set of display scalars as follows:

$DS = \{red, green, blue, transparency, reflectivity, vector_x, vector_y, vector_z,$
$contour_1, ..., contour_n, x, y, z, animation, selector_1, ..., selector_m\}$

Also define a subset of display scalars

$DOMDS = \{x, y, z, animation, selector_1, ..., selector_m\}$ and define

$Y_{DOMDS} = \mathbf{X}\{I_d \mid d \in DOMDS\}$ and $Y = \mathbf{X}\{I_d \mid d \in DS\}$. Let

$P_{DOMDS} : Y \rightarrow Y_{DOMDS}$ be the natural projection from $Y$ onto $Y_D$ (that is, if $a \in Y$ and $b = P_{DOMDS}(a)$, then for all $d \in DOMDS$, $b_d = a_d$). Then we can define $V_{display}$ as follows.

**Def.** $V_{display} = \{A \in V \mid \forall b, c \in MAX(A).\ P_{DOMDS}(b) = P_{DOMDS}(c) \Rightarrow b = c\}$. That is, if $A$ is an object in $V_{display}$, then different tuples in $A$ cannot have the same set of values for all display scalars in *DOMDS*.

In Prop. I.4 we will define conditions under which the displays of data objects are members of $V_{display}$. First, we prove three lemmas. Note that we use the notation $a_d$ for the $d$ component of a tuple $a \in \mathbf{X}\{I_d \mid d \in DS\}$.

**Prop. I.1.** Given a type $t \in T$ and $A \in D(F_t)$, then, for all tuples $a \in A$, $\forall d \in DS.\ (d \notin MAP_D(SC(t)) \Rightarrow a_d = \perp)$.

**Proof.** There is $B \in F_t$ such that $A = D(B)$. By Prop. F.12 for any $a \in A$ there is $b \in U$ such that $\downarrow a = D(\downarrow b)$. Since $\downarrow a \leq A$, $\downarrow b \leq B$ so $b \in B$. Furthermore, by Prop. F.12, if $a_d \neq \perp$ then there is $s \in S$ and $b_s \neq \perp$ such that $\downarrow(\perp,...,a_d,...,\perp) = D(\downarrow(\perp,...,b_s,...,\perp))$ and $d \in MAP_D(s)$. By Prop. D.1, $\forall s \in S.\ (b_s \neq \perp \Rightarrow s \in SC(t))$. Thus $a_d \neq \perp \Rightarrow d \in MAP_D(SC(t))$. ■

**Prop. I.2.** Given a tuple type $t = struct\{t_1;...;t_n\} \in T$, $A \in D(F_t)$ and $a = a_1 \vee ... \vee a_n \in A$, where $\forall i\ .\ a_i \in A_i \in D(F_{t_i})$, then $a \in MAX(A) \Leftrightarrow \forall i.\ a_i \in MAX(A_i)$.

**Proof.** Note that $a$ and the $a_i$ are tuples, and the *sup* of tuples is taken componentwise, so $\forall d \in DS.\ a_d = a_{1d} \vee ... \vee a_{nd}$. Also note that

$i \neq j \Rightarrow SC(t_i) \cap SC(t_i) = \phi$, and, by Prop. F.9,

$i \neq j \Rightarrow MAP_D(SC(t_i)) \cap MAP_D(SC(t_i)) = \phi$. If there is some $i$ such that $a_i \notin MAX(A_i)$, then $\exists b_i \in A_i.\ a_i < b_i$ so $b = a_1 \vee ... \vee b_i \vee ... \vee a_n \in A$. Now, $a_i < b_i \Rightarrow \exists d \in DS.\ a_{id} < b_{id}$ and (since $j \neq i \Rightarrow a_{jd} = \perp = b_{jd}$) $a_d = a_{id}$ and $b_d = b_{id}$, so $a < b$. Thus $a \notin MAX(A)$. Conversely, if $a \notin MAX(A)$ then $\exists b \in A.\ a < b$ with $a = a_1 \vee ... \vee a_n$, $b = b_1 \vee ... \vee b_n$, and

$\forall i.\ a_i, b_i \in A_i$. For some $d \in DS$, $a_d < b_d$. Thus $b_d > \perp$ so $\exists j.\ d \in MAP_D(SC(t_j))$, and so $a_d < b_d \Rightarrow a_j < b_j$ (since $a_d = a_{jd}$ and $b_d = b_{jd}$). Thus $a_j \notin MAX(A_j)$. ■

**Prop. I.3.** Given a tuple type $t = struct\{t_1;...;t_n\} \in T$, and given $B_i \in F_{t_i}$ and $A_i = D(B_i)$ for $i$=1,..,$n$, then:

(a) if $b_i \in B_i$ and $\downarrow a_i = D(\downarrow b_i)$ for $i$=1,..,$n$, then $\downarrow(a_1 \vee ... \vee a_n) = D(\downarrow(b_1 \vee ... \vee b_n))$

(b) $A_i = \{a_i \mid \exists b_i \in B_i.\downarrow a_i = D(\downarrow b_i)\}$

(c) $\bigvee\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} = \{a_1 \vee ... \vee a_n \mid \forall i.\ a_i \in A_i\}$

**Proof.** First we prove (a). Note that the $a_i$ and $b_i$ are tuples. By Prop. D.1,

$\forall$i $\neq$ j. $\forall s \in S.$ ($b_{is} = \perp$ or $b_{js} = \perp$), so $(b_1 \vee ... \vee b_n)$ exists. Also, by Prop. D.1 and by Prop. F.12, $\forall d \in DS.$ $d \notin MAP_D(SC(t_i)) \Rightarrow a_d = \perp$, and by Prop. F.9,

$\forall$i $\neq$ j. $MAP_D(\mathrm{SC}(t_i)) \cap MAP_D(\mathrm{SC}(t_j)) = \phi$, so $\forall$i $\neq$ j. $\forall d \in DS.$ ($a_{id} = \perp$ or $a_{jd} = \perp$), and so $(a_1 \vee ... \vee a_n)$ exists. Given $\downarrow a_i = D(\downarrow b_i)$ then by Prop. F.12, the components of $b_i$ determine the components of $a_i$. If $\downarrow x = D(\downarrow(b_1 \vee ... \vee b_n))$ then the components of $(b_1 \vee ... \vee b_n)$ determine the components of $x$. Since $\forall$i $\neq$ j. $\forall s \in S.$ ($b_{is} = \perp$ or $b_{js} = \perp$), the components of $(b_1 \vee ... \vee b_n)$ are just the components of each of the $b_i$, so $x = (a_1 \vee ... \vee a_n)$, proving (a).

By Prop. F.12, for all $b_i \in B_i$ there is $a_i \in A_i = D(B_i)$ such that $\downarrow a_i = D(\downarrow b_i)$, so

$A_i \supseteq \{a_i \mid \exists b_i \in B_i.\downarrow a_i = D(\downarrow b_i)\}$. Conversely, by Prop. F.12, for all $a_i \in A_i$ there is $b_i \in B_i$ such that $\downarrow a_i = D(\downarrow b_i)$, so $A_i \subseteq \{a_i \mid \exists b_i \in B_i.\downarrow a_i = D(\downarrow b_i)\}$. Together these prove (b).

Clearly, $\bigvee\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} \supseteq \{a_1 \vee ... \vee a_n \mid \forall i.\ a_i \in A_i\}$. Pick $a \in \bigvee\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\}$. By Prop. C.10, there is a directed set $M \subseteq \bigcup\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\}$ such that $a = \bigvee M$. However,

$\bigcup\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} = \{c \mid (\forall i.\ \exists a_i \in A_i).\ c \le (a_1 \vee ... \vee a_n)\}$.

Now, for $c \le (a_1 \vee ... \vee a_n)$, by Prop. C.9, $c = ((c \wedge a_1) \vee ... \vee (c \wedge a_n))$ where $(c \wedge a_i) \in A_i$, so $c \in \{a_1 \vee ... \vee a_n \mid a_i \in A_i\}$. Thus $M \subseteq \{a_1 \vee ... \vee a_n \mid a_i \in A_i\}$ such that $a = \bigvee M$. For each $m \in M$, let $m = (m_1 \vee ... \vee m_n)$ where $m_i \in A_i$. Then, since *sups* of tuples are taken componentwise and since $\forall i \neq j.\ \forall d \in DS.\ (m_{id} = \perp \text{ or } m_{jd} = \perp)$,

$a = \bigvee M = \{(\bigvee m_1) \vee ... \vee (\bigvee m_n)) \mid m \in M\}$. However, $(\bigvee m_i) \in A_i$ since $A_i$ is closed, so $a \in \{a_1 \vee ... \vee a_n \mid a_i \in A_i\}$. This proves (c). ■

Now we show that $MAX(A)$ is finite for data objects of types $t \in T$, and demonstrate conditions on $t$ and $D$ that ensure that displays of data objects of type $t$ are in $V_{display}$.

**Prop. I.4.** If $D$ is a display function, then for all types $t \in T$ and all $A \in D(F_t)$, $MAX(A)$ is finite. Furthermore, $MAP_D(DOM(t)) \subseteq DOMDS \Rightarrow D(F_t) \subseteq V_{display}$.

**Proof.** We will demonstrate both parts of this proposition by induction on the structure of $t$. Note that if $t'$ is a subtype of $t$, then $MAP_D(DOM(t')) \subseteq MAP_D(DOM(t))$. Thus, if $t$ satisfies the hypothesis of the second part, then its subtypes also satisfy the hypothesis of the second part.

Let $t \in S$ (note that $MAP_D(DOM(t)) = \phi \subseteq DOMDS$) and let $A \in D(F_t)$. Then, by the Theorem F.14, $\exists d \in MAP_D(t).\ A \in V_d$. Furthermore,

$A \in V_d \Rightarrow \exists a \in I_d.\ A = \downarrow(\perp,...,a,...,\perp)$, so $MAX(A) = \{(\perp,...,a,...,\perp)\}$. $MAX(A)$ has a single member and is thus finite. Therefore $A \in V_{display}$ and thus

$t \in S \Rightarrow D(F_t) \subseteq V_{display}$.

Let $t = struct\{t_1;...;t_n\} \in T$. Given $A \in D(F_t)$ there is $B \in F_t$ such that $A = D(B)$ and $\exists B_1 \in F_{t_1}$ ... $\exists B_n \in F_{t_n}$. B = $\{(b_1 \vee ... \vee b_n) \mid \forall i.\ b_i \in B_i\}$. Also let $A_i = D(B_i)$. Then

$A = D(B) =$

$D(\bigvee\{\downarrow b \mid b \in B\}) =$ (by Prop. B.3)

$\bigvee\{D(\downarrow b) \mid b \in B\} =$

$\bigvee\{D(\downarrow(b_1 \vee ... \vee b_n)) \mid \forall i. b_i \in B_i\} =$ (by Prop. I.3 (a))

$\bigvee\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i. \downarrow a_i = D(\downarrow b_i) \ \&\ b_i \in B_i\} =$ (apply Prop. I.3 (b) to each $i$)

$\bigvee\{\downarrow(a_1 \vee ... \vee a_n) \mid \forall i. a_i \in A_i\} =$ (by Prop. I.3 (c))

$\{(a_1 \vee ... \vee a_n) \mid \forall i. a_i \in A_i\}$

Thus $A \in D(F_t) \Rightarrow \exists A_1 \in D(F_{t_1})$... $\exists A_n \in D(F_{t_n})$. $A = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\}$ and by Prop. I.2, $MAX(A) = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in MAX(A_i)\}$. By the inductive hypothesis, the $MAX(A_i)$ are finite, so $MAX(A)$ is finite. Now assume that $MAP_D(DOM(t)) \subseteq DOMDS$ but that $A \notin V_{display}$ (that is, assume that the second part of the proposition is not true). Then $\exists b, c \in MAX(A).\ P_{DOMDS}(b) = P_{DOMDS}(c)\ \&\ b \neq c$.

Let $b =$

$b_1 \vee ... \vee b_n$ and $c = c_1 \vee ... \vee c_n$ where $\forall i.\ b_i, c_i \in A_i$. The *sups* are taken componentwise for the tuples $b$ and $c$, so for all $d \in DS$, $b_d = b_{1d} \vee ... \vee b_{nd}$ and $c_d = c_{1d} \vee ... \vee c_{nd}$. Now

$P_{DOMDS}(b) = P_{DOMDS}(c) \Rightarrow \forall d \in DOMDS.\ b_d = c_d$. Pick $d \in DOMDS$, and we will show that $\forall i.\ b_{id} = c_{id}$. If $\exists i.\ d \in MAP_D(SC(t_i))$ then $\forall i' \neq i.\ d \notin MAP_D(SC(t_{i'}))$ and hence $\forall i' \neq i.\ b_{i'd} = \perp = c_{i'd}$ so that $b_{id} = b_d = c_d = c_{id}$, and hence $\forall i.\ b_{id} = c_{id}$. If

$\forall i.\ d \notin MAP_D(SC(t_i))$ then $\forall i.\ b_{id} = \perp = c_{id}$. Either way, $P_{DOMDS}(b) = P_{DOMDS}(c)$ implies that $\forall d \in DOMDS.\ \forall i.\ b_{id} = c_{id}$ and so $\forall i.\ P_{DOMDS}(b_i) = P_{DOMDS}(c_i)$. On the other hand, $b \neq c \Rightarrow \exists e \in DS.\ b_e \neq c_e$. However, $e \notin MAP_D(SC(t_i)) \Rightarrow b_{ie} = \perp = c_{ie}$ and $\forall i.\ e \notin MAP_D(SC(t_i))$ would imply $b_e = \perp = c_e$.

Thus $\exists j.\ e \in MAP_D(SC(t_j))$, and for this $j$, $b_{je} = b_e = c_e = c_{je}$ (since $b_{ie} = \perp = c_{ie}$ for $i \neq j$). And this implies that, for this $j$, $b_j \neq c_j$. However, by the inductive hypothesis, $b_j = c_j$, since we have already shown that $P_{DOMDS}(b_j) = P_{DOMDS}(c_j)$. Thus the assumption that $A \notin V_{display}$ has led to a contradiction, so $D(F_t) \subseteq V_{display}$.

Let $t = (array\ [w]\ of\ r) \in T$. Given $A \in D(F_t)$ there is $B \in F_t$ such that $A = D(B)$, and there is a finite set $G \in FIN(H_w)$ and a function $a \in (G \rightarrow H_r)$ such that

$$B = \{b_1 \vee b_2 \mid g \in G\ \&\ b_1 \in E_w(g)\ \&\ b_2 \in E_r(a(g))\} =$$

$$\bigcup\{\{b_1 \vee b_2 \mid b_1 \in E_w(g) \;\&\; b_2 \in E_r(a(g))\} \mid g \in G\}$$

Define $B_w(g) = E_w(g) \in F_w$, $B_r(g) = E_r(a(g)) \in F_r$, $A_w(g) = D(B_w(g)) \in D(F_w)$ and $A_r(g) = D(B_r(g)) \in D(F_r)$. Then

$$B = \bigcup\{\{b_1 \vee b_2 \mid b_1 \in B_w(g) \;\&\; b_2 \in B_r(g)\} \mid g \in G\}$$

This is a finite union of objects in $F_{struct\{w;\, r\}}$ for the tuple type $struct\{w; r\}$. Thus, since the union of a finte set of closed sets is the *sup* of those sets, and since $D$ preserves *sups*,

$$A = D(B) = \bigcup\{D(\{b_1 \vee b_2 \mid b_1 \in B_w(g) \;\&\; b_2 \in B_r(g)\}) \mid g \in G\}$$

which, as shown in the tuple case of this proof, is equal to

$$\bigcup\{\{a_1 \vee a_2 \mid a_1 \in A_w(g) \;\&\; a_2 \in A_r(g)\} \mid g \in G\}$$

Recall that $MAX(A)$ is the set of maximal elements of $A$, so it is clear that if $A = A_1 \cup A_2$, then $MAX(A) \subseteq MAX(A_1) \cup MAX(A_2)$. Thus

$$MAX(A) \subseteq \bigcup\{MAX(\{a_1 \vee a_2 \mid a_1 \in A_w(g) \;\&\; a_2 \in A_r(g)\}) \mid g \in G\}$$

and so, by Prop. I.2,

$$MAX(A) \subseteq \bigcup\{\{a_1 \vee a_2 \mid a_1 \in MAX(A_w(g)) \;\&\; a_2 \in MAX(A_r(g))\} \mid g \in G\}$$

$G$ is finite, and by the inductive hypothesis, $MAX(A_w(g))$ and $MAX(A_r(g))$ are finite, so $MAX(A)$ is finite.

Now assume that $MAP_D(DOM(t)) \subseteq DOMDS$. As shown for scalars, $MAX(A_w(g))$ has a single member, $MAX(A_w(g)) = \{a_1(g)\}$. Applying Prop. F.12,

$A_w(g) = \downarrow a_1(g) = D(E_w(g)) = D(\downarrow b_1(g))$ where $b_1(g) = (\perp,...,g,...,\perp)$. If $g \neq g'$, then

$b_1(g) \neq b_1(g')$ and $a_1(g) \neq a_1(g')$. Also, given $g$, there is $d \in MAP_D(w)$ such that

$a_1(g) = (\perp,...,a_{1d}(g),...,\perp)$. Since $w \in DOM(t)$, then $MAP_D(w) \subseteq DOMDS$ and

$d \in DOMDS$. Thus $g \neq g' \Rightarrow a_1(g) \neq a_1(g') \Rightarrow P_{DOMDS}(a_1(g)) \neq P_{DOMDS}(a_1(g'))$.

Now pick $e, f \in MAX(A)$ and assume that $P_{DOMDS}(e) = P_{DOMDS}(f)$. Let

$e = e_1 \vee e_2$ and $f = f_1 \vee f_2$ with $e_1 \in MAX(A_w(g_e))$, $f_1 \in MAX(A_w(g_f))$, $e_2 \in MAX(A_r(g_e))$ and $f_2 \in MAX(A_w(g_f))$. From what we have just seen,

$g_e \neq g_f \Rightarrow P_{DOMDS}(e_1) \neq P_{DOMDS}(f_1)$. However, since $w \notin \mathrm{SC}(r)$,

$MAP_D(w) \cap MAP_D(\mathrm{SC}(r)) = \phi$ so

$P_{DOMDS}(e_1) \neq P_{DOMDS}(f_1) \Rightarrow P_{DOMDS}(e) \neq P_{DOMDS}(f)$. This contradicts our assumption, so we must have $g_e = g_f$ and, since $MAX(A_w(g))$ has a single member for each $g$, $e_1 = f_1$. Now $e_2, f_2 \in MAX(A_r(g_e))$ and $MAP_D(w) \cap MAP_D(\mathrm{SC}(r)) = \phi$ implies that $P_{DOMDS}(e) = P_{DOMDS}(f) \Rightarrow P_{DOMDS}(e_2) = P_{DOMDS}(f_2)$. By the inductive hypothesis, $A_r(g_e) \in V_{display}$, so $P_{DOMDS}(e_2) = P_{DOMDS}(f_2) \Rightarrow e_2 = f_2$. Thus $e = e_1 \vee e_2 = f_1 \vee f_2 = f$, establishing that $A \in V_{display}$ and that $D(F_t) \subseteq V_{display}$. ■

The next proposition shows that the auxiliary function $D'$ provides a way to compute the maximal tuples of display objects.

**Prop. I.5.** If $D$ is a display function, if $D'$ is the auxiliary function defined in Appendix H, if $t \in T$ and if $A \in F_t$, then $MAX(D(A)) = \{D'(a) \mid a \in MAX(A)\}$

**Proof.** By Prop. H.5, $D(A) = \{D'(a) \mid a \in A\}$. By Prop. H.2, $D'$ is an order embedding, so, given $a$, $b \in A$, $\neg(a < b) \Leftrightarrow \neg(D'(a) < D'(b))$. Thus

$a \in MAX(A) \Leftrightarrow D'(a) \in MAX(D(A))$. ■

The inverse of the second part of Prop. I.4 is almost true. The next two propositions make this precise.

**Prop. I.6.** If $D$ is a display function, if $t = (array\ [w]\ of\ r) \in T$, and if

$\exists g_1, g_2 \in H_w.\ (g_1 \neq g_2\ \&\ D(\downarrow(\bot,...,g_1,...,\bot)) = \downarrow b_1 \in V_{d_1}\ \&$

$D(\downarrow(\bot,...,g_2,...,\bot)) = \downarrow b_2 \in V_{d_2}\ \&\ d_1, d_2 \notin DOMDS)$,

then $\exists A \in D(F_t).\ A \notin V_{display}$.

**Proof.** Let $G = \{g_1, g_2\} \in FIN(H_w)$, pick $C \in H_r$, and define $f \in (G \rightarrow H_r)$ by $f(g_1) = C$ and $f(g_2) = C$. Pick $c \in E_r(C)$ such that $D(\downarrow c) = \downarrow a$ and $a \in MAX(D(E_r(C)))$. Then $(\bot,...,g_1,...,\bot)\vee c$ and $(\bot,...,g_1,...,\bot)\vee c$ are both members of $E_t(f) \in F_t$. Note that $D(\downarrow((\bot,...,g_1,...,\bot)\vee c)) = \downarrow(a \vee b_1)$ and $D(\downarrow((\bot,...,g_2,...,\bot)\vee c)) = \downarrow(a \vee b_2)$, so $a \vee b_1$ and $a \vee b_2$ are both members of $D(E_t(f))$. Clearly $b_1 \in MAX(D(\downarrow(\bot,...,g_1,...,\bot)))$ and $b_2 \in MAX(D(\downarrow(\bot,...,g_2,...,\bot)))$ (since $b_1$ and $b_2$ are maximal in $\downarrow b_1$ and $\downarrow b_2$). Furthermore, since $w \notin SC(r)$, $d_1 \notin MAP_D(SC(r))$ and $d_2 \notin MAP_D(SC(r))$, so $a \vee b_1$ and $a \vee b_2$ are members of $MAX(D(E_t(f)))$. For all d $\in$ DOMDS, $b_{1d} = \bot$ and $b_{2d} = \bot$, so $P_{DOMDS}(a \vee b_1) = P_{DOMDS}(a \vee b_2)$. Since $w \notin SC(r)$, $d_1 \notin MAP_D(SC(r))$ and $d_2 \notin MAP_D(SC(r))$, so $a_{d_1} = \bot$ and $a_{d_2} = \bot$. However, $g_1 \neq g_2$ so $b_1 \neq b_2$ and hence $(a \vee b_1)_{d_1} \neq (a \vee b_2)_{d_1}$ and $(a \vee b_1)_{d_2} \neq (a \vee b_2)_{d_2}$ ($d_1$ and $d_2$ may or may not be the same). Thus $(a \vee b_1) \neq (a \vee b_2)$, so $D(E_t(f)) \notin V_{display}$. ■

**Prop. I.7.** If $D$ is a display function, if $t \in T$, and if $t$ has a sub-type $t'$ such that $\exists A' \in D(F_{t'}).\ A' \notin V_{display}$, then $\exists A \in D(F_t).\ A \notin V_{display}$.

**Proof.** By an inductive argument, it is enough to prove this when $t'$ is an immediate sub-type of $t$. First, let $t$ be a tuple $t = struct\{t_1;...;t_n\}$ where $t' = t_k$. Let $A_k = A'$ and pick $a_k, a_k' \in MAX(A_k)$ such that $P_{DOMDS}(a_k) = P_{DOMDS}(a_k')$ and $a_k \neq a_k'$. For $i \neq k$, pick $A_i \in D(F_{t_i})$ and $a_i \in MAX(A_i)$. Then define $A = \{b_1 \vee ... \vee b_n \mid b_i \in A_i\} \in D(F_t)$. For $i \neq j$, $MAP_D(SC(t_i)) \cap MAP_D(SC(t_j)) = \phi$ so $a = a_1 \vee ... \vee a_k \vee ... \vee a_n \in MAX(A)$ and $a' = a_1 \vee ... \vee a_k' \vee ... \vee a_n \in MAX(A)$. Now $P_{DOMDS}(a_1 \vee ... \vee a_n) = P_{DOMDS}(a_1) \vee ... \vee P_{DOMDS}(a_n)$ and $P_{DOMDS}(a_k) = P_{DOMDS}(a_k')$ so $P_{DOMDS}(a_1 \vee ... \vee a_k \vee ... \vee a_n) = P_{DOMDS}(a_1 \vee ... \vee a_k' \vee ... \vee a_n)$. However, $a_k \neq a_k'$ so $a_1 \vee ... \vee a_k \vee ... \vee a_n \neq a_1 \vee ... \vee a_k' \vee ... \vee a_n$. Thus $A \notin V_{display}$.

Next, let $t$ be an array $t = (array\ [w]\ of\ r)$. In the proof of Prop. I.4 we saw that $MAX(B')$ has only a single member for any $B' \in D(F_w)$, and hence $B' \in V_{display}$. Thus $t' = r$ and $A' \in D(F_r)$. Pick $G = \{g\} \in FIN(H_w)$, pick $b, c \in MAX(A')$ such that $P_{DOMDS}(b) = P_{DOMDS}(c)$ and $b \neq c$, and define $f \in (G \rightarrow H_r)$ by $f(g) = E_r^{-1}(D^{-1}(A'))$ ($A' \in D(F_r)$ implies that $D^{-1}(A')$ exists, and $D^{-1}(A') \in F_r$ implies that $E_r^{-1}(D^{-1}(A'))$ exists). If $D(\downarrow(\perp,...,g,...,\perp)) = \downarrow a$ then $a \in MAX(D(E_w(g)))$ and so $a \vee b$ and $a \vee c$ are members of $MAX(D(E_t(f)))$ (since $MAP_D(w) \cap MAP_D(SC(r)) = \phi$). However, $P_{DOMDS}(a \vee b) = P_{DOMDS}(a \vee c)$ but $a \vee b \neq a \vee c$. Thus $A \notin V_{display}$. ■

**Bibliography**

Avila, R., Taosong H., Lichan H., A. Kaufman, H. Pfister, C. Silva, L. Sobierakski, and S. Wang, 1994; VolVis: a diversified volume visualization system. Proc. IEEE Visualization '94, 31-38.

Bancroft, G. V., F. J. Merrit, T. C. Plessel, P. G. Kelaita, R. K. McCabe, and A. Globus, 1990; FAST: a multi-processed environment for visualization of computational fluid dynamics. Proc. IEEE Visualization '90, 14-27.

Bertin, J., 1983; Semiology of Graphics. W. J. Berg, Jr. University of Wisconsin Press.

Beshers, C., and S. Feiner, 1992; Automated design of virtual worlds for visualizing multivariate relations. Proc. Visualization '92, IEEE. 283-290.

Bier, E. A., M. C. Stone, K. Pier, W. Buxton, and T. Rose, 1994; Toolglass and magic lenses: the see-through interface. Proc. ACM Siggraph, 73-80.

Brittain, D. L., J. Aller, M. Wilson, S-L. C. Wang, 1990; Design of an end-user data visualization system. Proc. IEEE Visualization '90, 323-328.

Brodlie, K., A. Poon, H. Wright, L. Brankin, G. Banecki, and A. Gay, 1993; GRASPARC - a problem solving environment integrating computation and visualization. Proc. IEEE Visualization '93, 102-109.

Brown, M. H., and R. Sedgewick, 1984; A System for algorithm animation. Computer Graphics 18(3), 177-186.

Chen, M., S. J. Mountford, and A. Sellen, 1988; A study in interactive 3-D rotation using 2-D control devices. Computer Graphics 22(4), 121-129.

Corrie, B., and P. Mackerras, 1993; Data shaders. Proc. IEEE Visualization '93, 275-282.

Davey, B. A., and H. A. Priestly, 1990; Introduction to Lattices and Order. Cambridge University Press.

DeFanti, T. A., M. D. Brown, and B. H. McCormick, 1989; Visualization: expanding scientific and engineering research opportunities. IEEE Computer 22(8), 12-25.

Domik, G. O., and B. Gutkauf, 1994; User modeling for adaptive visualization systems. Proc. IEEE Visualization '94, 217-223.

Duff, T., 1992; Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. Computer Graphics 26(2), 131-138.

Foley, J. D., and A. Van Dam, 1982; Fundamentals of Interactive Computer Graphics. Addison-Wesley.

Gierz, G., K. H. Hofmann, K. Keimal, J. D. Lawson, M. Mislove and D. Scott, 1980; A Compendium of Continuous Lattices. Springer-Verlag.

Globus, A., C. Levit, and T. Lasinski, 1991; A tool for visualizing the topology of three-dimensional vector fields. Proc. IEEE Visualization '91, 33-40.

Green, N., and M. Kass, 1994; Error-bounded antialiased rendering of complex environments. Proc. ACM Siggraph, 59-66.

Gunter, C. A., and D. S. Scott, 1990; Semantic domains. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 633-674.

Haber, R. B., B. Lucas and N. Collins, 1991; A data model for scientific visualization with provisions for regular and irregular grids. Proc. Visualization 91. IEEE. 298-305.

Haberli, P., 1988; ConMan: A visual programming language for interactive graphics. Computer Graphics 22(4), 103-111.

Haeberli, P., and K. Akeley, 1990; The accumulation buffer: hardware support for high-quality rendering. Computer Graphics 24(4), 309-318.

Haltiner, G. J., and R. T. Williams, 1980; Numerical prediction and dynamic meteorology, second edition. John Wiley & Sons, p. 40.

Hanrahan, P., and J. Lawson, 1990; A language for shading and lighting calculations. Computer Graphics 24(4), 289-298.

Hansen, C. D., and P. Hinker, 1992; Massively parallel isosurface extraction. Proc. IEEE Visualization '92, 77-81.

Helman, J. L., and L. Hesselink, 1990; Surface representations of two- and three-dimensional fluid flow topology. Proc. IEEE Visualization '90, 6-13.

Hibbard, W., 1986; Computer generated imagery for 4-D meteorological data. Bull. Amer. Met. Soc., 67, 1362-1369.

Hibbard, W., 1986; 4-D display of meteorological data. Proceedings, 1986 Workshop on Interactive 3D Graphics. Chapel Hill, Siggraph, 23-36.

Hibbard, W., and D. Santek, 1989; Interactivity is the key. Chapel Hill Workshop on Volume Visualization, University of North Carolina, Chapel Hill, 39-43.

Hibbard, W., and D. Santek, 1989; Visualizing large data sets in the earth sciences. IEEE Computer 22(8), 53-57.

Hibbard, W., L. Uccellini, D. Santek, and K. Brill, 1989; Application of the 4-D McIDAS to a model diagnostic study of the Presidents' Day cyclone. Bull. Amer. Met. Soc., 70(11), 1394-1403.

Hibbard, W., and D. Santek, 1990; The VIS-5D system for easy interactive visualization. Proc. Visualization '90, IEEE. 28-35.

Hibbard, W., D. Santek, and G. Tripoli, 1991; Interactive atmospheric data access via high speed networks. Computer Networks and ISDN Systems, 22, 103-109.

Hibbard, W., C. R. Dyer, and B. E. Paul, 1992; Display of scientific data structures for algorithm visualization. Proc. IEEE Visualization '92, 139-146.

Hibbard. W. L., C. R. Dyer, and B. E. Paul, 1994; A lattice model for data display. Proc. IEEE Visualization '94, 310-317.

Hibbard, W. L., B. E. Paul, A. L. Battaiola, D. A. Santek, M-F. Voidrot-Martinez and C. R. Dyer, 1994; Interactive Visualization of Computations in the Earth and Space Sciences. IEEE Computer 27(7), 65-72.

Hultquist, J. P. M., and E. L. Raible, 1992; SuperGlue: A programming environment for scientific visualization. Proc. Visualization '92, IEEE. 243-250.

Itoh, T., and K. Koyamada, 1994; Isosurface generation by using extrema graphs. Proc. IEEE Visualization '94, 77-83.

Kass, M., 1992; CONDOR: constraint-based dataflow. Computer Graphics 26(2), 321-330.

Kochevar, P., Z. Ahmed, J. Shade, and C. Sharp, 1993; Bridging the gap between visualization and data management: a simple visualization management system. Proc. IEEE Visualization '93, 94-101.

Lang, U., R. Lang, and R. Ruhle, 1991; Integration of visualization and scientific calculation in a software system. Proc. IEEE Visualization '91, 268-273.

Lee, J. P., and G. G. Grinstein, 1994; Database Issues for Data Visualization. Proc. of IEEE Visualization '93 Workshop. Springer-Verlag.

Levkowitz, H., 1991; Color icons: merging color and texture perception for integrated visualization of multiple parameters. Proc. IEEE Visualization '91, 164-170.

Lischinski, D., B. Smits and D. P. Greenberg, 1994; Bounds and error estimates for radiosity. Proc. ACM Siggraph, 67-74.

Lohse, J., H. Rueter, K. Biolsi, and N. Walker, 1990; Classifying visual knowledge representations: a foundation for visualization research. Proc. IEEE Visualization '90, 131-138.

Lorensen, W., and H. Cline, 1987; Marching cubes: a high-resolution 3D surface construction algorithm. Computer Graphics, 21(4), 163-170.

Lorenz, E. N., 1963; Deterministic nonperiodic flow. J. Atmos. Sci., 20, 130-141.

Lucas, B., G. D. Abrams, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe, 1992; An architecture for a scientific visualization system. Proc. IEEE Visualization '92, 107-114.

Mackinlay, J., 1986; Automating the design of graphical presentations of relational information. ACM Transactions on Graphics, 5(2), 110-141.

Matveyev, S. V., 1994; Approximation of isosurface in the marching cube: ambiguity problem. Proc. IEEE Visualization '94, 288-292.

McConnell, C., and D. Lawton, 1988; IU software environments. Proc. IUW, 666-677.

McCormick, B.H., T.A. DeFanti and M.D. Brown, eds., 1987; Visualization in scientific computing. Computer Graphics, 21(6).

Montani, C., R. Scateni, and R. Scopigno, 1994; Discretized marching cubes. Proc. IEEE Visualization '94, 281-287.

Moore, R. E., 1966; Interval Analysis. Prentice Hall.

Nadas, T., and A. Fournier, 1987; GRAPE: An environment to build display processes. Computer Graphics 21(4), 103-111.

Nielson, G. M., and B. Hamann, 1991; The asymptotic decider: resolving the ambiguity in marching cubes. Proc. IEEE Visualization '91, 83-91.

Ning, P., and L. Hesselink, 1993; Fast volume rendering of compressed data. Proc. IEEE Visualization '93, 11-18.

Perlin, K., and D. Fox, 1993; Pad: an alternative approach to the computer interface. Proc. ACM Siggraph, 57-64.

Potmesil, M., and E. Hoffert, 1987; FRAMES: Software tools for modeling, animation and rendering of 3D scenes. Computer Graphics 21(4), 75-84.

Rabin, R. M., S. Stadler, P. J. Wetzel, D. J. Stensrud, and M. Gregory, 1990; Observed effects of landscape variability on convective clouds. Bull. Amer. Meteor. Soc., 71, 272-280.

Ranjan, V., and A. Fournier, 1994; Volume models for volumetric data. IEEE Computer, 27(7), 28-36.

Rasure, J., D. Argiro, T. Sauer, and C. Williams, 1990; A visual language and software development environment for image processing. International J. of Imaging Systems and Technology, Vol. 2, 183-199.

Read, R. L., D. S. Fussell and A. Silberschatz, 1993; Algorithms for the sandbag: an approach to imprecise set representation. Technical Report TR-93-12, Department of Computer Sciences, University of Texas at Austin.

Robertson, P. K., 1990; A methodology for scientific data visualization: choosing representations based on a natural scene paradigm. Proc. IEEE Visualization '90, 114-123.

Robertson, P. K., 1991; A methodology for choosing data representations. Computer Graphics and Applications, 11(3), 56-67.

Robertson, P. K., R. A. Earnshaw, D. Thalman, M. Grave, J. Gallup and E. M. De Jong, 1994; Research issues in the foundations of visualization. Computer Graphics and Applications 14(2), 73-76.

Rogowitz, B. E., and L. A. Treinish, 1993; An architecture for rule-based visualization. Proc. IEEE Visualization '93, 236-243.

Rolf, J., and J. Helman, 1994. IRIS Performer: a high performance multiprocessing toolkit for real-time 3D graphics. Proc. ACM Siggraph, 381-394.

Sanders, W. T., R. J. Edgar, M. Juda, W. L. Kraushaar, D. McCammon, S. L. Snowden, J. Zhang, M. A. Skinner, K. Jahoda, R. Kelley, A. Smalle, C. Stahle, and A. Szymkowiak, 1993; Preliminary results from the Diffuse X-ray Spectrometer. EUV, X-ray, and Gamma-ray Instrumentation for Astronomy IV. SPIE, Vol. 2006, 221-232.

Schmidt, D. A., 1986; Denotational Semantics. Wm.C.Brown.

Schroeder, W. J., W. E. Lorenson, G. D. Montanaro and C. R. Volpe, 1992; VISAGE: An object-oriented scientific visualization system. Proc. Visualization '92, IEEE. 219-226.

Scott, D. S., 1971; The lattice of flow diagrams. In Symposium on Semantics of Algorithmic Languages, E. Engler. ed. Springer-Verlag, 311-366.

Scott, D. S., 1976; Data types as lattices. Siam J. Comput., 5(3), 522-587.

Scott, D. S., 1982; Lectures on a mathematical theory of computation, in: M. Broy and G. Schmidt, eds., *Theoretical Foundations of Programming Methodology*, NATO Advanced Study Institutes Series (Reidel, Dordrecht, 1982) 145-292.

Segal, M., 1990; Using tolerances to guarantee valid polyhedral modeling results. Computer Graphics 24(4), 105-114.

Senay, H., and E. Ignatius, 1991; Compositional analysis and synthesis of scientific data visualization techniques. In Scientific Visualization of Physical Phenomena, N. M. Patrikalakis, ed. Springer-Verlag, 269-281.

Senay, H., and E. Ignatius, 1994; A knowledge-based system for visualization design. Computer Graphics and Applications, 14(6), 36-47.

Snyder, J. M., 1992; Interval Analysis for computer graphics. Computer Graphics 26(2), 121-130.

Springmeyer, R. R., M. M. Blattner, and N. L. Max, 1992; A characterization of the scientific data analysis process. Proc. IEEE Visualization '92, 235-242.

Treinish, L. A., 1991; SIGGRAPH '90 workshop report: data structure and access software for scientific visualization. Computer Graphics 25(2), 104-118.

Tuchman, A., D. Jablonowski, and G. Cybenko, 1991; Run-time visualization of program data. Proc. IEEE Visualization '91, 255-261.

Twiddy, R., J. Cavallo, and S. M. Shiri, 1994; Restorer: a visualization technique for handling missing data. Proc. IEEE Visualization '94, 212-216.

Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, 1989; The application visualization system: A computational environment for scientific visualization. Computer Graphics and Applications, 9(4), 30-42.

Wehrend, S., and C. Lewis, 1990; A problem-oriented classification of visualization techniques. Proc. IEEE Visualization '90, 139-143.

Williams, C, J. Rasure, and C. Hansen, 1992; The state of the art of visual languages for visualization. Proc. IEEE Visualization '92, 202-209.

Wyvill, G., McPheeters, C., and B. Wyvill, 1986; Data structures for soft objects. Visual Computer 2(4), 227-234.