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 $(D \rightarrow E)$ to denote the set of all functions from D to E.

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

Def. If D and E are *posets*, a function $f{:}D \rightarrow E$ 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 $f:D \rightarrow E$, 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) $\quad u \leq u' \Leftrightarrow math(u') \subseteq math(u)$

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* ≤ 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.

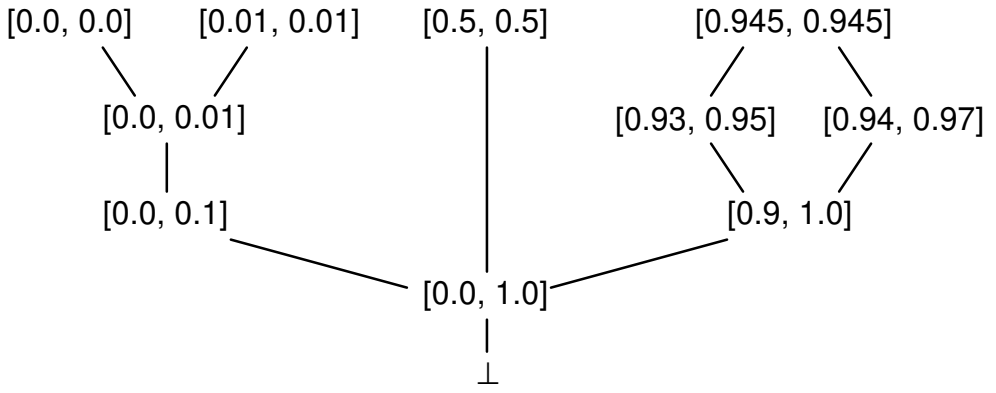

Figure 3.1. Order relation of a continuous scalar. Closed real intervals are used as approximate representations of real numbers, ordered by the inverse of containment (that is, containing intervals are "less than" contained intervals). We also include a least element ⊥ that corresponds to a missing 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.

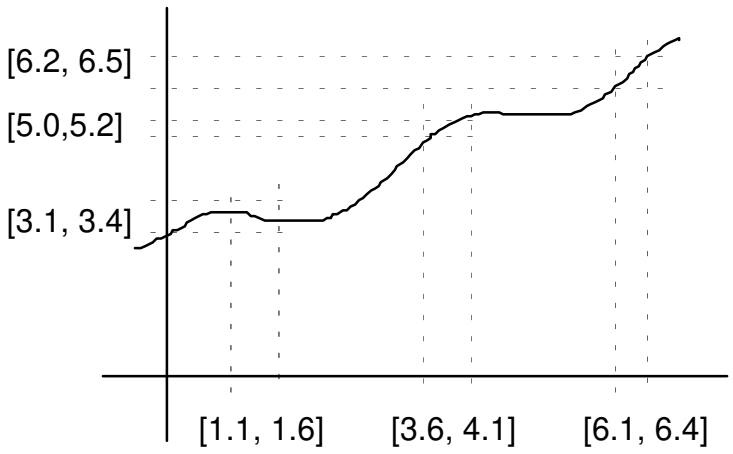

Figure 3.2. Approximating real functions by arrays.

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.

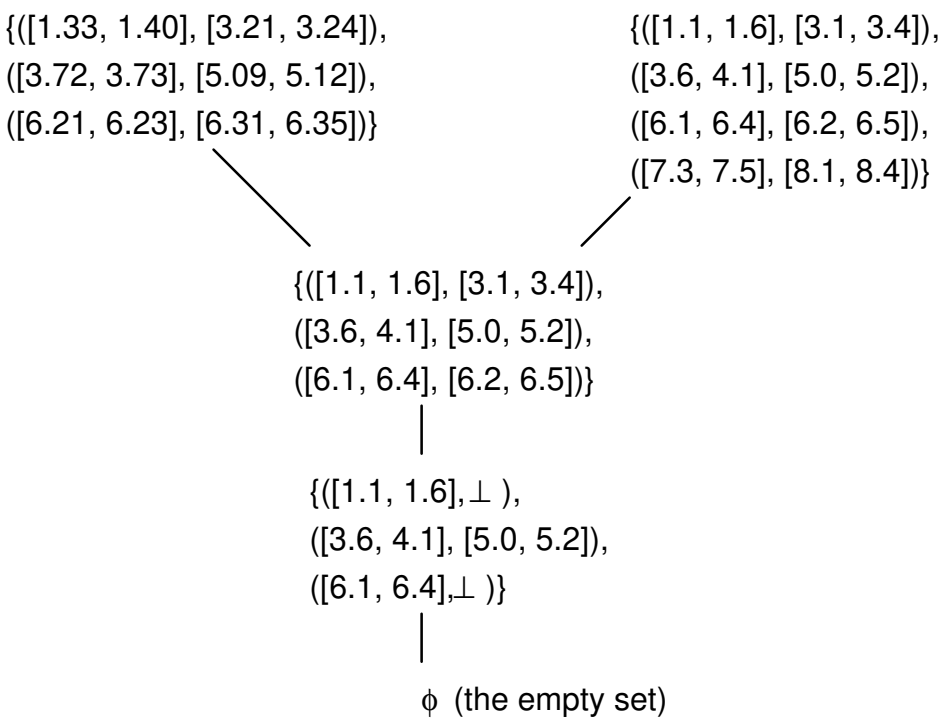

Figure 3.3. Order relation of arrays.

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 $D : U \rightarrow V$ 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* $\prec$ *false* and *undefined* $\prec$ *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 $v \in \downarrow D(X)$.

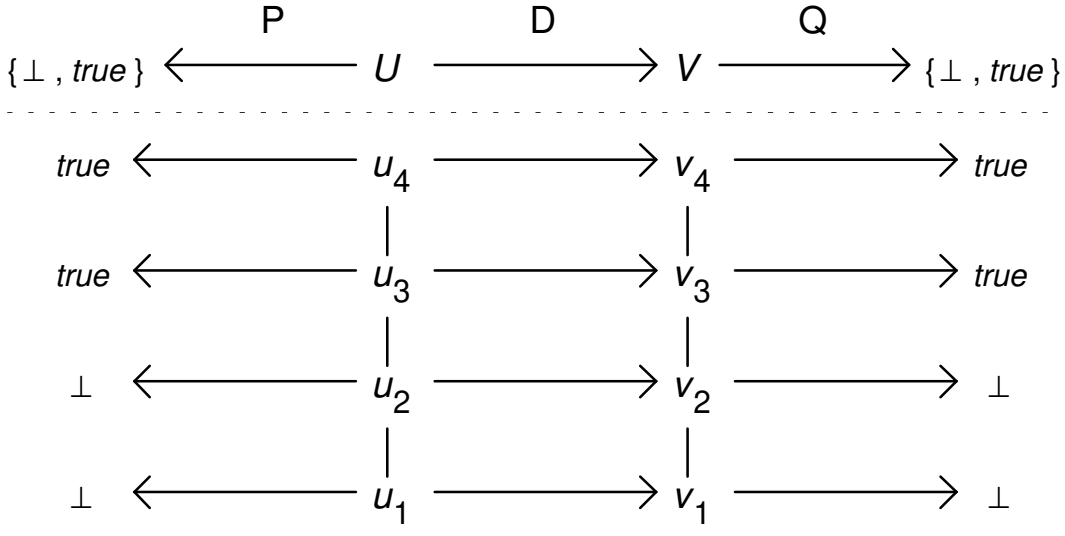

Figure 3.8. The expressiveness conditions specify that $D : U \rightarrow V$ defines a correspondence between monotone predicates on U and V.

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.

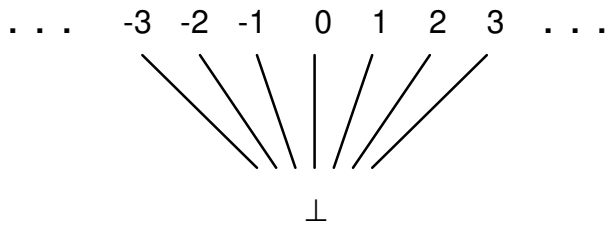

Figure 3.9. Order relation of a discrete scalar.

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.

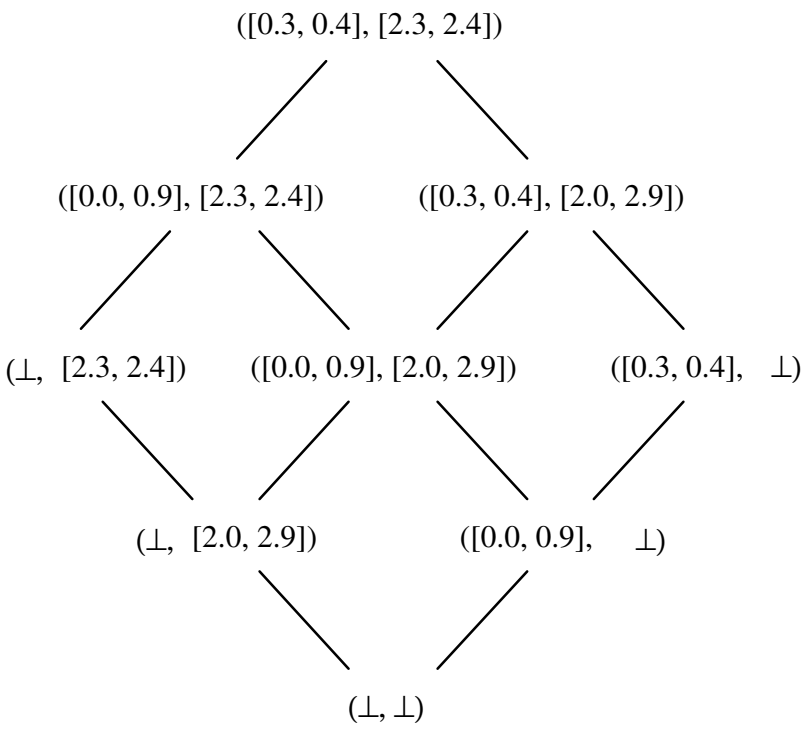

Figure 3.10. Order relation of tuples. Tuples are members of cross products. This figure shows a few elements in a cross product of two sets of continuous scalar values, plus the order relations among those elements. In a cross product, the least element is the tuple of least elements of the factor sets.

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.

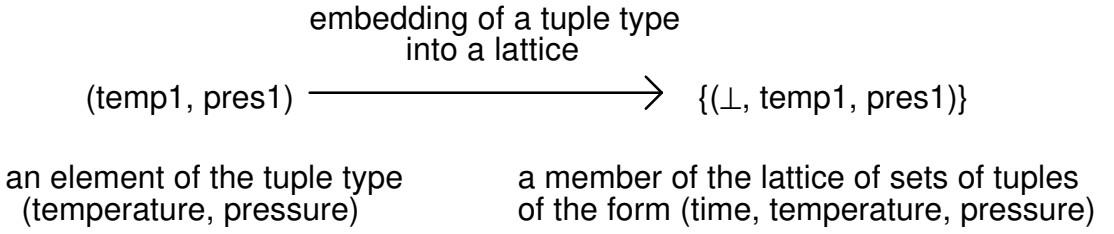

Figure 3.11. Embedding a tuple type into a lattice of sets of tuples.

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.

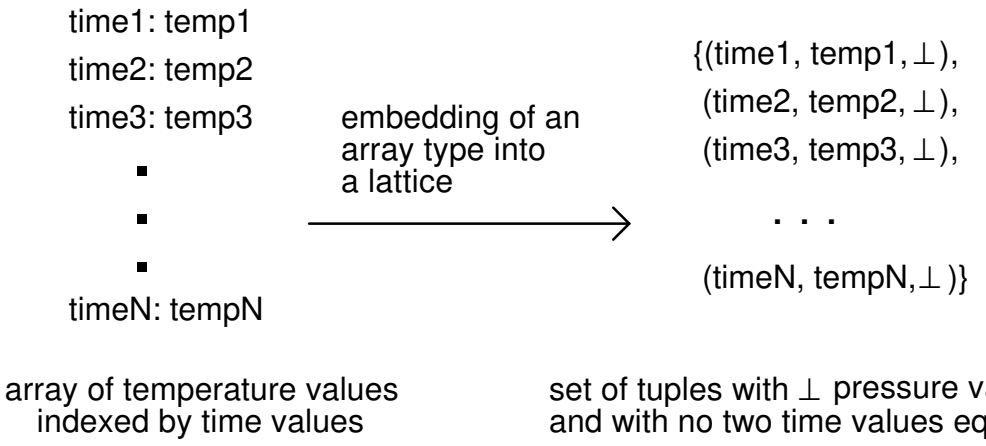

Figure 3.12. Embedding an array type into a lattice of sets of tuples.

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 $a \prec b$, 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 ϕ 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.

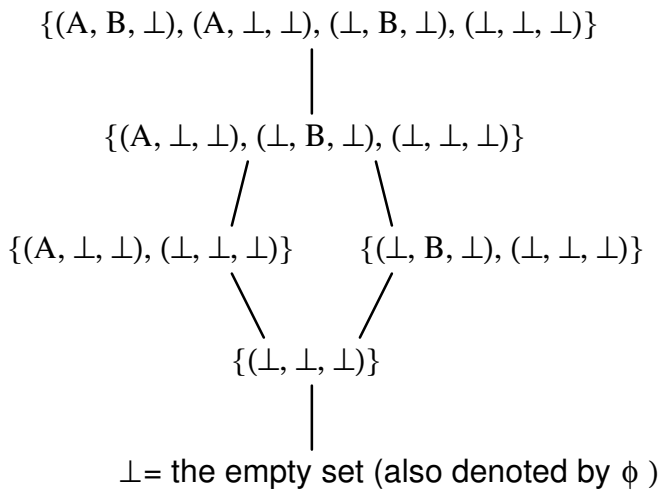

Figure 3.13. Defining an order relation on sets of tuples. The sets are all down sets and are ordered by set containment. We assume that the three scalars that define these tuples are discrete, so that the down sets in this figure are all finite.

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

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

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

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

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

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

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

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

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

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 $MAX(A) = \{a \in A \mid \forall b \in A.\ \neg(a \prec b)\}$. 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:

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

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.

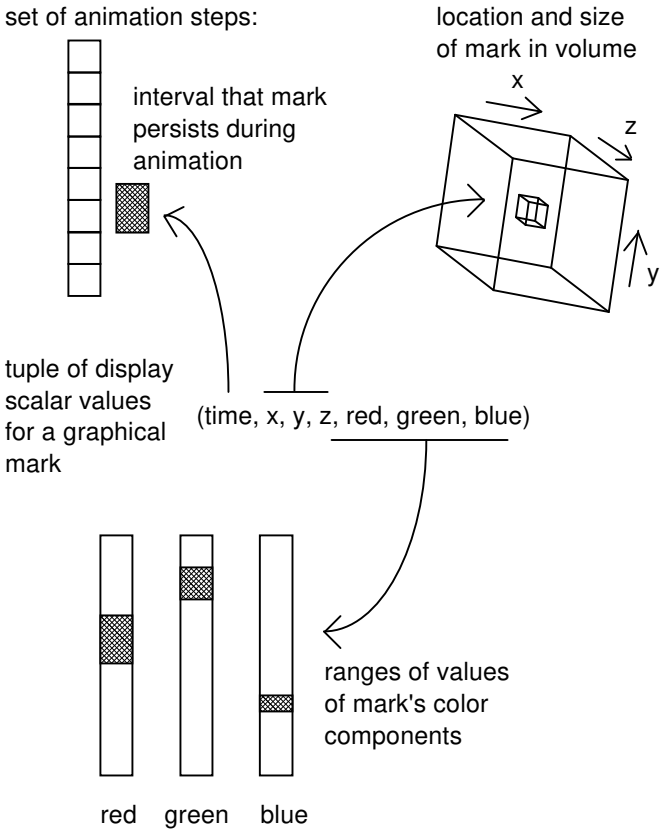

Figure 3.14. The roles of seven continuous display scalars (*x*, *y*, *z*, *red*, *green*, *blue*, *time*) in an animated three-dimensional display model.

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 $RENDER : V \rightarrow V'$ 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 $RENDER : V \rightarrow V'$ 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 $RENDER : V \rightarrow V'$, 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.

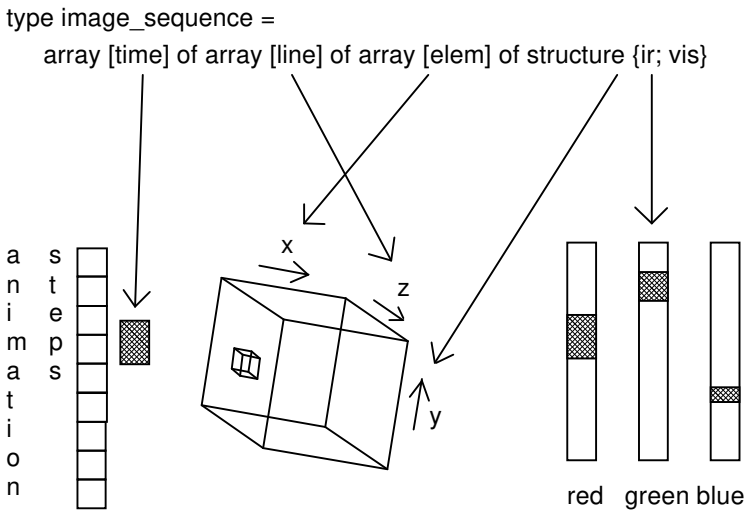

Figure 3.15. Mappings from scalars to display scalars.

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:$\mathbf{R} \rightarrow \mathbf{R}$ and h'_s:$\mathbf{R} \rightarrow \mathbf{R}$ 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:$U \rightarrow V$, a continuous scalar $s \in S$, and $[x, y] \in I_s$, then

(3.15) $g_s(x, y) = inf\{g'_s(z) \mid x \leq z \leq y\}$ and

(3.16) $h_s(x, y) = sup\{h'_s(z) \mid x \leq z \leq y\}$.

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 $D{:}U \to V$, 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 $g'_s{:}\mathbf{R} \to \mathbf{R}$ and $h'_s{:}\mathbf{R} \to \mathbf{R}$ 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 $D{:}U \to V$ 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.

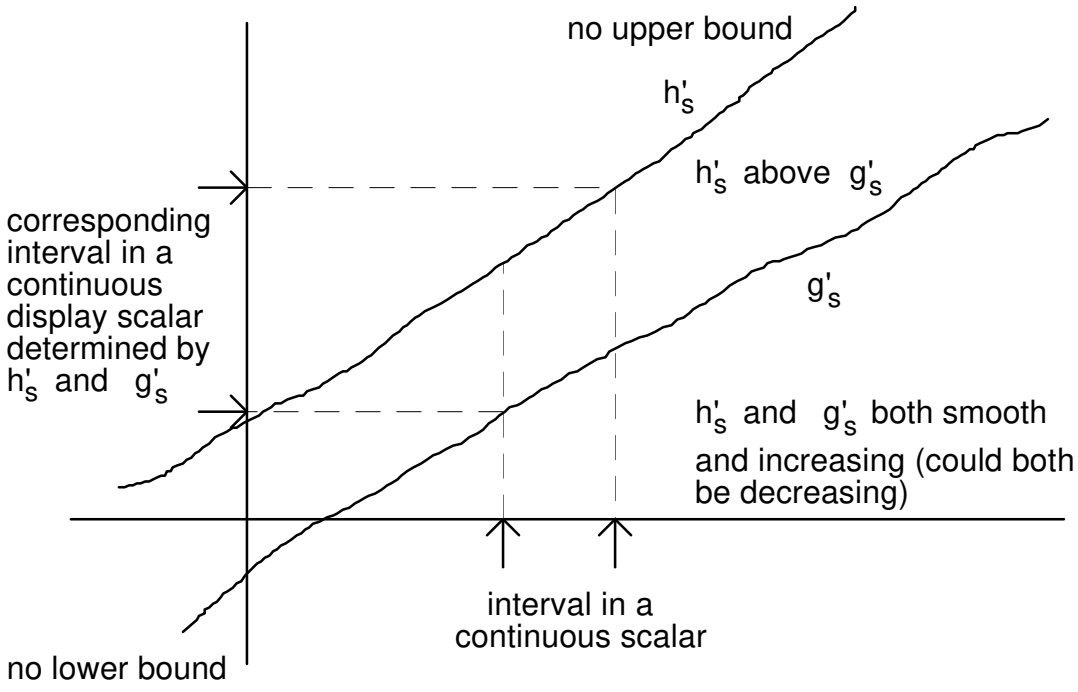

Figure 3.16. The behavior of a display function *D* on a continuous scalar interpreted in terms of the behavior of functions h'_s and g'_s.

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: $S \rightarrow POWER(DS)$ 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 $[u, v] \in I_s$, $g_s(u, v) = inf\{g'_s(z) \mid u \leq z \leq v\}$ and

$h_s(u, v) = sup\{h'_s(z) \mid u \leq z \leq v\}$,

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 $a \in I_s\backslash\{\perp\}$,

$D(\downarrow(\perp,...,a,...,\perp)) = b \in V_d$ 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. $D{:}U \rightarrow V$ 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

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

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

$$(3.22)\quad t \in S \;\&\; A = \downarrow(\perp,...,a,...,\perp) \in F_t \Rightarrow$$
$$MAX(A) = \{(\perp,...,a,...,\perp)\}$$
$$(3.23)\quad t = struct\{t_1;...;t_n\} \in T \;\&\; A = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in A_i\} \in F_t \Rightarrow$$
$$MAX(A) = \{(a_1 \vee ... \vee a_n) \mid \forall i.\ a_i \in MAX(A_i)\}$$
$$(3.24)\quad t = (array\ [w]\ of\ r) \in T \;\&\; A = \{a_1 \vee a_2 \mid g \in G \;\&\; a_1 \in E_w(g) \;\&\; a_2 \in E_r(a(g))\} \in F_t$$
$$\Rightarrow MAX(A) = \{a_1 \vee a_2 \mid g \in G \;\&\; a_1 \in MAX(E_w(g)) \;\&\; a_2 \in MAX(E_r(a(g)))\}$$

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.