A Lattice Model for Data Display

William L. Hibbard[1&2], Charles R. Dyer[2] and Brian E. Paul[1]
[1]Space Science and Engineering Center
[2]Computer Sciences Department
University of Wisconsin - Madison

Abstract

In order to develop a foundation for visualization, we develop lattice models for data objects and displays that focus on the fact that data objects are approximations to mathematical objects and real displays are approximations to ideal displays. These lattice models give us a way to quantize the information content of data and displays and to define conditions on the visualization mappings from data to displays. Mappings satisfy these conditions if and only if they are lattice isomorphisms. We show how to apply this result to scientific data and display models, and discuss how it might be applied to recursively defined data types appropriate for complex information processing.

1 Introduction

Robertson et.al. have described the need for formal models that can serve as a foundation for visualization techniques and systems [13]. Models can be developed for data (e.g., the fiber bundle data model [4] describes the data objects that computational scientists use to approximate functions between differentiable manifolds), displays (e.g., Bertin's detailed analysis of static 2-D displays [1]), users (i.e., their tasks and capabilities), computations (i.e., how computations are expressed and executed), and hardware devices (i.e., their capabilities).

Here we focus on the process of transforming data into displays. We define a data model as a set U of data objects, a display model as a set V of displays, and a visualization process as a function D: $U \rightarrow V$. The usual approach to visualization is synthetic, constructing the function D from simpler functions. The function may be synthesized using rendering pipelines [5, 11, 12], defining different pipelines appropriate for different types of data objects within U. Object oriented programming may be used to synthesize a polymorphic function D [9, 15] that applies to multiple data types within U.

We will try to address the need for a formal foundation for visualization by taking an analytic approach to defining D. Since an arbitrary function D: $U \rightarrow V$ will not produce displays $D(u)$ that effectively communicate the information content of data objects $u \in U$, we seek to define conditions on D to ensure that it does. For example, we may require that D be injective (i.e., one-to-one), so that no two data objects have the same display. However, this is clearly not enough. If we let U and V both be the set of images of 512 by 512 pixels with 24 bits of color per pixel, then any permutation of U can be interpreted as an injective function D from U to V. But an arbitrary permutation of images will not effectively communicate information. Thus we need to define stronger conditions on the function D. Our investigation depends on some complex mathematics, although we will only present the conclusions in this paper. The details are available in [7].

2 Lattices as data and display models

The purpose of data visualization is to communicate the information content of data objects in displays. Thus if we can quantify the information content of data objects and displays this may give us a way to define conditions on the visualization function D. The issue of information content has already been addressed in the study of programming language semantics [14], which seeks to assign meanings to programs. This issue arises because 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 contains less information (i.e., is less precise) than any of the values that a program might produce if it terminates, and thus introduces an order relation based on information content into the set of program meanings. In order to define a correspondence between the ways that programs are constructed, and the sets of meanings of programs, Scott developed an elegant lattice theory for the meanings of programs [16].

Scientists have data with undefined values, although their sources are numerical problems and failures of observing instruments rather than non-terminating computations. An undefined value for pixels in satellite images contains less information than valid pixel radiances and thus creates an order relation between data values. Data are often accompanied by metadata [18] that describe their accuracy, for example as error bars, and these accuracy estimates also create order relations between data values based on information content (i.e., precision). Finally, array data objects are often approximations to functions, as for example a satellite image is a finite approximation (i.e., a finite sampling in both space and radiance) to a continuous radiance field, and such arrays may be ordered based on the resolution with which they sample functions.

In general scientists use computer data objects as finite approximations to the objects of their mathematical models, which contain infinite precision numbers and functions with infinite ranges. Thus metadata for missing data indicators, numerical accuracy and function sampling are really central to the meaning of scientific data and should play an important role in a data model. We define a data model U as a lattice of data objects, ordered by how precisely they approximate mathematical objects. To say that U is a *lattice* [2] means that there is a partial order on U (i.e., a binary relation such that, for all $u_1, u_2, u_3 \in U$, $u_1 \leq u_1$, $u_1 \leq u_2$ & $u_2 \leq u_1 \Rightarrow u_1 = u_2$ and $u_1 \leq u_2$ & $u_2 \leq u_3 \Rightarrow u_1 \leq u_3$) and that any pair $u_1, u_2 \in U$ have a least upper bound (denoted by $u_1 \vee u_2$) and a greatest lower bound (denoted by $u_1 \wedge u_2$).

The notion of precision of approximation also applies to displays. They have finite resolutions in space, color and time (i.e., animation). 2-D images and 3-D volume renderings are composed of finite numbers of pixels and voxels and are finite approximations to idealized mathematical displays. Thus we will assume that our display model V is a lattice and that displays are ordered according to their information content (i.e., precision of approximation to ideal displays). In Sections 4 and 5 we will present examples of scientific data and display lattices.

We assume that U and V are complete lattices, so that they contain the mathematical objects and ideal displays that are limits of sets of data objects and real displays (a lattice is *complete* if any subset has a least upper bound and a greatest lower bound). Just as we study functions of rational numbers in the context of functions of real numbers (the completion of the rational numbers), we will study visualization functions between the complete lattices U and V, recognizing that data objects and real displays are restricted to countable subsets of U and V.

3 Conditions on visualization functions

The lattice structures of U and V provide a way to quantize information content and thus to define conditions on functions of the form $D: U \rightarrow V$. In order to define these conditions we draw on the work of Mackinlay [10]. 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 (i.e., 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 (i.e., a function of the form $P: U \rightarrow \{false, true\}$). However, since data objects are approximations to mathematical objects, we should avoid predicates such that providing more precise information about a mathematical object (i.e., going from u_1 to u_2 where $u_1 \leq u_2$) changes the truth value of the predicate (e.g., $P(u_1) = true$ but $P(u_2) = false$). Thus we consider predicates that take values in $\{undefined, false, true\}$ (where $undefined < false$ and $undefined < true$), and we require predicates to preserve information ordering (that is, if $u_1 \leq u_2$ then $P(u_1) \leq P(u_2)$; functions that preserve order are called *monotone*). We also observe that a predicate of the form $P: U \rightarrow \{undefined, false, true\}$ can be expressed in terms of two 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 $P(u_1) \neq \mathrm{P}(u_2)$, 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 $v \in V$.

This requires that D^{-1} be a function from V to U, and hence that D be bijective (i.e., one-to-one and onto). However, it is too strong to require that a data model realize every possible display. Since U is a complete lattice it contains a maximal data object X (the least upper bound of all members of U). Then $D(X)$ is the display of X and the notation $\downarrow D(X)$ represents the complete lattice of all displays less than $D(X)$. 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 $v \in \downarrow D(X)$.

These conditions quantify the relation between the information content of data objects and the information content of their displays. We use them to define a class of functions:

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

In [7] we prove the following result about display functions:

Proposition 1. A function D: $U \rightarrow V$ is a display function if and only if it is a lattice isomorphism from U onto $\downarrow D(X)$ [i.e., for all $u_1, u_2 \in U$, $D(u_1 \vee u_2) = D(u_1) \vee D(u_2)$ and $D(u_1 \wedge u_2) = D(u_1) \wedge D(u_2)$].

This result may be applied to any complete lattice models of data and displays. In the next three sections we will explore its consequences in one setting.

4 A Scientific data model

We will develop a scientific data model that integrates metadata for missing data indicators, numerical accuracy and function sampling. We will develop this data model in terms of a set of data types, starting with *scalar* types used to represent the primitive variables of mathematical models. Given a scalar type s, let I_s denote the set of possible values of a data object of the type s. First we define *continuous* scalars to represent real variables, such as *time*, *temperature* and *latitude*. If s is continuous then I_s includes the undefined value, which we denote by the symbol $\perp$ (usually used to denote the least element of a lattice), and also includes all closed real intervals. We interpret the closed real interval $[x, y]$ as an approximation to an actual value that lies between x and y. In our lattice structure, these intervals are ordered by the *inverse* of set containment, since a smaller interval provides more precise information than a containing interval. Figure 1 illustrates the order relation on a continuous scalar type. Of course, an actual implementation can only include a countable number of closed real intervals (such as the set of rational intervals).

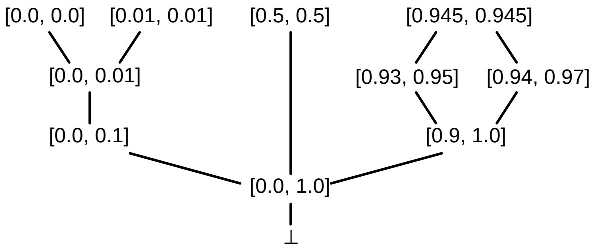

Figure 1. The order relations among a few values of a continuous scalar.

We also define *discrete* scalars to represent integer and string variables, such as *year*, *frequency_count* and *satellite_name*. If s is discrete then I_s includes $\perp$ and a countable set of incomparable values (no integer is more precise than any other integer). Figure 2 illustrates the order relation on a discrete scalar type.

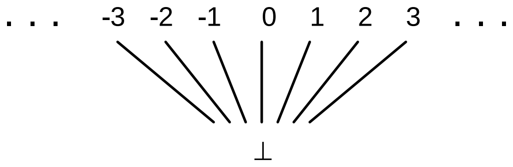

Figure 2. The order relations among a few values of a discrete scalar.

Complex data types are constructed from scalar data types as arrays and tuples. An array data type represents a function between mathematical variables. For example, a function from *time* to *temperature* is approximated by data objects of the type (*array* [*time*] *of temperature*;). We say that *time* is the *domain* type of this array, and *temperature* is its *range* type. Values of an array type are sets of 2-tuples that are (domain, range) pairs. The set {([1.1, 1.6], [3.1, 3.4]), ([3.6, 4.1], [5.0, 5.2]), ([6.1, 6.4], [6.2, 6.5])} is an array data object that contains three samples of a function from *time* to *temperature*. The domain value of a sample lies in the first interval of a pair and the range value lies in the second interval of a pair, as illustrated in Figure 3. Adding more samples, or increasing the precision of samples, will create a more precise approximation to the function. Figure 4 illustrates the order relation on an array data type. The domain of an array must be a scalar type, but its range may be any scalar or complex type (its definition may not include the array's domain type).

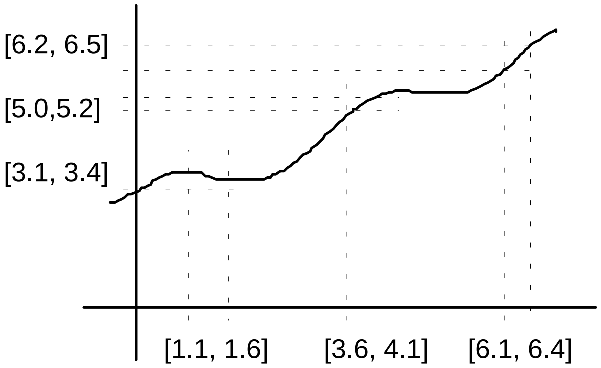

Figure 3. An array samples a real function as a set of pairs of intervals.

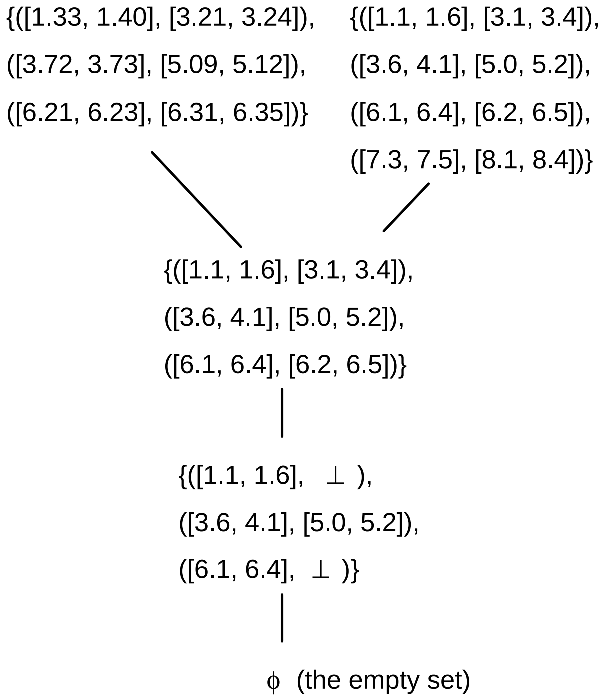

Figure 4. The order relations among a few arrays.

Tuple data types represent tuples of mathematical objects. For example, a 2-tuple of values for temperature and pressure is represented by data objects of the type *struct*{*temperature*; *pressure*;}. Data objects of this type are 2-tuples (*temp*, *pres*) where $temp \in I_{temperature}$ and $pres \in I_{pressure}$. We say that *temperature* and *pressure* are *element* types of the tuple. The elements of a tuple type may be any complex types (they must be defined from disjoint sets of scalars). A tuple data object *x* is less than or equal to a tuple data object *y* if every element of *x* is less than or equal to the corresponding element of *y*, as illustrated in Figure 5.

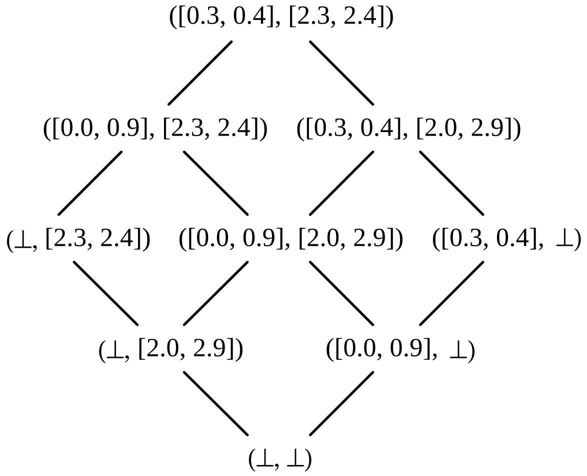

Figure 5. The order relations among a few tuples.

This data model is applied to a particular application by defining a finite set *S* of scalar types (these would represent the primitive variables of the application), and defining *T* as the set of all types that can be constructed as arrays and tuples from the scalar types in *S*. For each type $t \in T$ we can define a countable set H_t of data objects of type *t* (these correspond to the data objects that are realized by an implementation).

In order to apply our lattice theory to this data model, we must define a single lattice *U* and embed each H_t in *U*. First define $X = \mathbf{X}\{I_s \mid s \in S\}$ as the cross product of the value sets of the scalars in *S*. Its members are tuples with one value from each scalar in *S*, ordered as illustrated in Figure 5. Now we would like to define *U* as the power set of *X* (i.e., the set of all subsets of *X*). However, power sets have been studied for the semantics of parallel languages and there is a well known problem with constructing order relations on power sets [14]. We expect this order relation to be consistent with the order relation on *X* and also consistent with set containment. 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:

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

However, given $a < b$, (1) implies that $\{b\} \leq \{a, b\}$ and $\{a, b\} \leq \{b\}$ are both true, which contradicts $\{b\} \neq \{a, b\}$. This problem can be resolved by restricting the lattice *U* to sets of tuples such every tuple is maximal in the set. That is, a set $A \subseteq X$ belongs to the lattice *U* if

$a < b$ is not true for any pair a, $b \in A$. The members of U are ordered by (1), as illustrated in Fig. 6, and form a complete lattice (see [7] for more details).

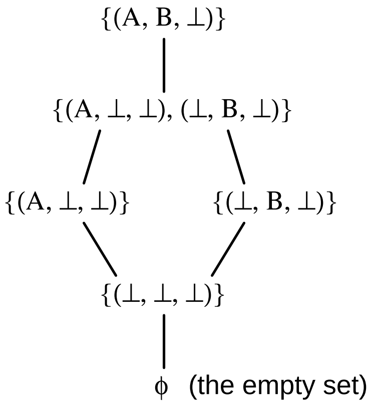

Figure 6. The order relations among a few members of a data lattice *U* defined by three scalars.

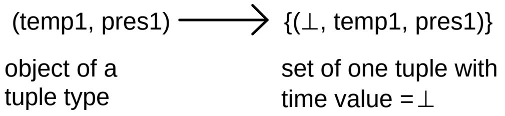

Figure 7. An embedding of a tuple type into a lattice.

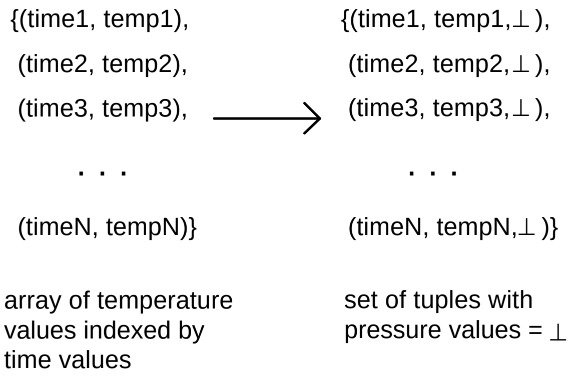

Figure 8. An embedding of an array type into a lattice.

To see how the data objects in H_t are embedded in U, consider a data lattice U defined from the three scalars *time*, *temperature* and *pressure*. Objects in the lattice U are sets of tuple of the form (*time*, *temperature*, *pressure*). We can define a tuple data type *struct*{*temperature*; *pressure*;}. A data object of this type is a tuple of the form (*temp*, *pres*) and can be mapped to a set of tuples (actually, it is a set consisting of one tuple) in U with the form {(⊥, *temp*, *pres*)}. This embeds the tuple data type in the lattice U, as illustrated in Figure 7.

Similarly, we can embed array data types in the data lattice. For example, consider an array data type (*array* [*time*] *of temperature*;). A data object of this type consists of a set of pairs of (*time*, *temp*). This array data object can be embedded in U as a set of tuples of the form (*time*, *temp*, ⊥). Figure 8 illustrates this embedding. The basic ideas presented in Figs. 7 and 8 can be combined to embed complex data types, defined as hierarchies of tuples and arrays, in data lattices (see [7] for details).

5 A scientific display model

For our scientific display model, we start with Bertin's analysis of static 2-D displays [1]. He modeled displays as sets of graphical marks, where each mark was described by an 8-tuple of graphical primitive values (i.e., two screen coordinates, size, value, texture, color, orientation and shape). The idea of a display as a set of tuple values is quite similar to the way we constructed the data lattice U. Thus 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 subsets A of Y such that every tuple is maximal in A.

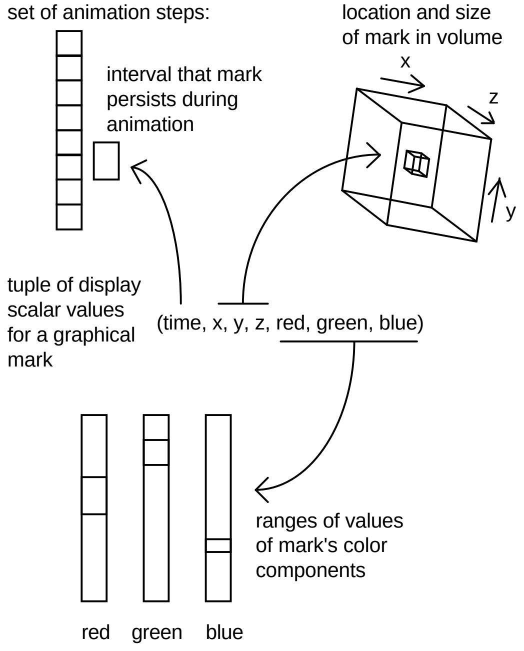
set of animation steps:
interval that mark persists during animation
location and size of mark in volume
x
z
y
tuple of display scalar values for a graphical mark
(time, x, y, z, red, green, blue)
ranges of values of mark's color components
red
green
blue

Figure 9. The roles of display scalars in an animated 3-D display model.

We can define a specific lattice V to model animated 3-D displays in terms of a set of seven continuous display scalars: (*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 place and duration of persistence of marks in an animation sequence. This is illustrated in Figure 9. A display in V is a set of tuples, representing a set of graphical marks.

Display scalars can be defined for a wide variety of attributes of graphical marks, and need not be limited to simple values. For example, a discrete display scalar may be an index into a set of complex shapes (i.e., icons).

6 Scalar mapping functions

Proposition 1 said that a function of the form $D: U \rightarrow V$ satisfies the expressiveness conditions (i.e., is a display function) if and only if D is a lattice isomorphism from U onto $\downarrow D(X)$, a sublattice of V. We can now apply this to the scientific data and display lattices described in Section 4 and 5.

The scalar and display scalar types play a special role in characterizing display functions in the context of our scientific models. Given a scalar type $s \in S$, define $U_S \subseteq U$ as the set of embeddings of objects of type s in U. That is, U_S consists of sets of tuples of the form $\{(\perp,...,b,...,\perp)\}$ (this notation indicates that all components of the tuple are $\perp$ except the s component, which is b). Similarly, given a display scalar type $d \in DS$, define $V_d \subseteq V$ as the set of embeddings of objects of type d in V. In [7] we prove the following result:

Proposition 2. If $D: U \rightarrow V$ is a display function, then we can define a mapping $MAP_D: S \rightarrow POWER(DS)$ (this is the power set of DS) such that for all scalars $s \in S$ and all for $a \in U_S$, 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 its values 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 tells us that display functions map scalars, which represent primitive variables like *time* and *temperature*, to display scalars, which represent graphical primitives like screen axes and color components. Most displays are already designed in this way, as, for example, a time series of temperatures may be displayed by mapping *time* to one axis and *temperature* to another. The remarkable thing is that Proposition 2 tells us that we don't have to take this way of designing displays as an assumption, but that it is a consequence of a more fundamental set of expressiveness conditions. Figure 10 provides examples of mappings from scalars to display scalars (*lat_lon* is a *real2d* scalar, as described in Section 7).

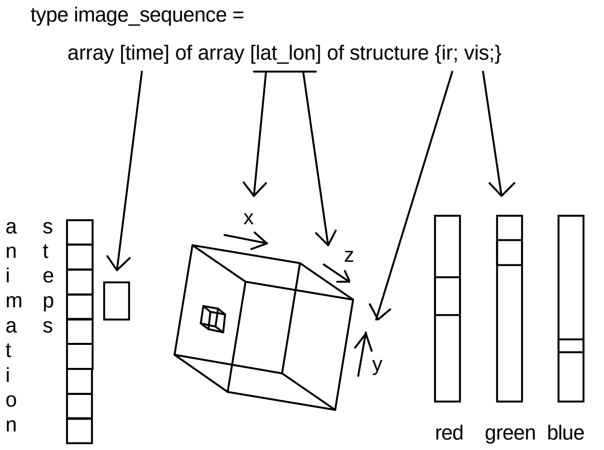

Figure 10. Mappings from scalars to display scalars.

In [7] we present a precise definition (the details are complex) of *scalar mapping functions* and show that $D: U \rightarrow V$ is a display function if and only if it is a scalar mapping function. Here we will just describe the behavior of display functions on continuous scalars. If s is a continuous scalar and $MAP_D(s) = d$, then D maps U_S to V_d. This can be interpreted by a pair of functions g_S:$\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ and h_S:$\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ (where $\mathbf{R}$ denotes the real numbers) such that for all $\{(\perp,...,[x, y],...,\perp)\}$ in U_S, $D(\{(\perp,...,[x, y],...,\perp)\}) = \{(\perp,...,[g_S(x, y), h_S(x, y)],...,\perp)\}$, which is a member of V_d. 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 $h'_S(z) = h_S(z, z)$. Then the functions g_S and h_S can be defined in terms of g'_S and h'_S as follows:

(2) $g_S(x, y) = min\{g'_S(z) \mid x \leq z \leq y\}$ and

(3) $h_S(x, y) = max\{h'_S(z) \mid x \leq z \leq y\}$.

These functions must satisfy the conditions illustrated in Figure 11.

Although the complete lattices U and V include members containing infinite numbers of tuples (these are mathematical objects and ideal displays) in [7] we prove the following:

Proposition 3. Given a display function $D: U \rightarrow V$, a data type $t \in T$ and an embedding of a data object from H_t to $a \in U$, then a contains a finite number of tuples and $D(a) \in V$ contains a finite number of tuples.

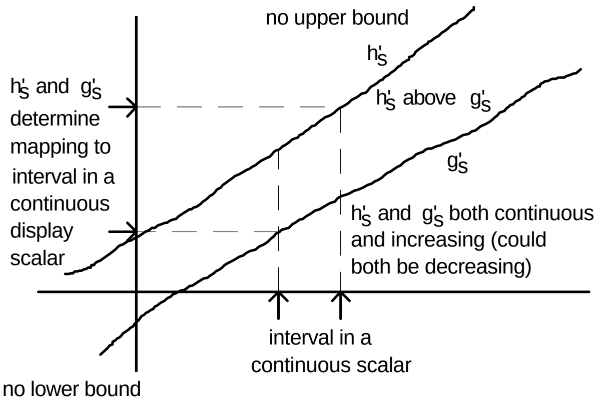
no upper bound
h'_S
h'_S and g'_S determine mapping to interval in a continuous display scalar
h'_S above g'_S
g'_S
h'_S and g'_S both continuous and increasing (could both be decreasing)
interval in a continuous scalar
no lower bound

Figure 11. The behavior of a display function *D* on a continuous scalar interpreted in terms of the behavior of functions h'_S and g'_S.

7 Implementation

The data and display models described in Sections 4 and 5, and the scalar mapping functions described in Section 6, are implemented in our VIS-AD system [6, 8]. This system is intended to help scientists experiment with their algorithms and steer their computations. It includes a programming language that allows users to define scalar and complex data types and to express scientific algorithms. The scalars in this language are classified as *real* (i.e., continuous), *integer* (discrete), *string* (discrete), *real2d* and *real3d*. The *real2d* and *real3d* scalars have no analog in the data model presented in Section 4, but are very useful as the domains of arrays that have non-Cartesian sampling in two and three dimensions. Users control how data are displayed by defining a set of mappings from scalar types (that they declare in their programs) to display scalar types. By defining a set of mappings a user defines a display function D: $U \rightarrow V$ that may be applied to display data objects of any type.

The VIS-AD display model includes the seven display scalars described for animated 3-D displays in Section 5, and also includes display scalars named *contour* and *selector*. Multiple copies of each of these may exist in a display lattice (the numbers of copies are determined by the user's mappings). Scalars mapped to *contour* are depicted by drawing isolevel curves and surfaces through the field defined by the *contour* values in graphical marks. For each *selector* display scalar, the user selects a set of values and only those graphical marks whose *selector* values that overlap this set are displayed. *Contour* is a *real* display scalar and *selector* display scalars take the type of the scalar mapped to them. We plan to add *real* display scalars for *transparency* and *reflectivity* to the system (to be interpreted by complex volume rendering of graphical marks), as well as a *real3d* display scalar for *vector* (to be interpreted by flow rendering techniques).

VIS-AD is available by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/visad directory. Get the README file for complete installation instructions.

8 Recursively defined data types

The data model in Section 4 is adequate for scientific data, but is inadequate for complex information processing which involves *recursively defined data types* [14]. For example, binary trees may be defined by the type *bintree* = *struct*{*bintree*; *bintree*; *value*;} (a leaf node is indicated when both *bintree* elements of the tuple are undefined). Several techniques have been developed to model such data using lattices. In the current context, the most promising is called *universal domains* [3, 17]. Just as we embedded data objects of many different types in the domain *U* in Section 4, data objects of many different recursively defined data types are embedded in a universal domain (which we also denote by *U*). However, these embeddings have been defined in order to study programming language semantics, and have a serious problem in the visualization context. Data objects of many different types are mapped to the same member of *U*. For example, an integer and a function from the integers to the integers may be mapped to the same member of *U*, and thus any display function of the form D: $U \rightarrow V$ will generate the same display for these two data objects. Thus, in order to extend our lattice theory of visualization to recursively defined data types, other embeddings into universal domains must be developed.

A suitable display lattice *V* must also be developed such that there exist lattice isomorphisms from a universal domain *U* into *V*. Displays involving *diagrams* and *hypertext links* are analogous to the pointers usually used to implement recursively defined data types. Thus the interpretation of *V* as a set of actual displays may involve these graphical techniques. However, since a large class of recursively defined data types can be embedded in *U*, and since *V* is isomorphic to *U*, these graphical techniques must be applied in a very abstract manner to define a suitable lattice *V*.

9 Conclusions

It is easy to think of metadata as secondary when we are focused on the task of making visualizations of data. However, it is central to the meaning of scientific data that they are approximations to mathematical objects, and lattices provide a way to integrate metadata about precision of approximation into a data model. By bringing the approximate nature of data and displays into central focus, lattices provide a foundation for understanding the visualization process and an analytic approach to defining the mapping from data to displays. While Proposition 2 just confirms standard practice in designing displays, it is remarkable that this practice can be deduced from the expressiveness conditions.

Although we have not derived any new rendering techniques by using lattices, the high level of abstraction of scalar mapping functions do provide a very flexible user interface for controlling how data are displayed.

There will be considerable technical difficulties in extending this work to recursively defined data types, but we are confident that the results will be interesting.

Acknowledgments

This work was support by NASA grant NAG8-828, and by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

References

[1] Bertin, J., 1983; Semiology of Graphics. W. J. Berg, Tr. University of Wisconsin Press.

[2] Davey, B. A. and H. A. Priestly, 1990; Introduction to Lattices and Order. Cambridge University Press.

[3] Gunter, C. A. and Scott, D. S., 1990; Semantic domains. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 633-674.

[4] 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.

[5] Haberli, P., 1988; ConMan: A visual programming language for interactive graphics; Computer Graphics 22(4), 103-111.

[6] Hibbard, W., C. Dyer and B. Paul, 1992; Display of scientific data structures for algorithm visualization. Visualization '92, Boston, IEEE, 139-146.

[7] Hibbard, W. L., and C. R. Dyer, 1994; A lattice theory of data display. Tech. Rep. # 1226, Computer Sciences Department, University of Wisconsin-Madison. Also available as compressed postscript files by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/lattice directory.

[8] Hibbard, W. L., B. E. Paul, D. A. Santek, C. R. Dyer, A. L. Battaiola, and M-F. Voidrot-Martinez, 1994; Interactive visualization of Earth and space science computations. IEEE Computer special July issue on visualization.

[9] Hultquist, J. P. M., and E. L. Raible, 1992; SuperGlue: A programming environment for scientific visualization. Proc. Visualization '92, 243-250.

[10] Mackinlay, J., 1986; Automating the design of graphical presentations of relational information; ACM Transactions on Graphics, 5(2), 110-141.

[11] Nadas, T. and A. Fournier, 1987; GRAPE: An environment to build display processes, Computer Graphics 21(4), 103-111.

[12] Potmesil, M. and E. Hoffert, 1987; FRAMES: Software tools for modeling, animation and rendering of 3D scenes, Computer Graphics 21(4), 75-84.

[13] 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.

[14] Schmidt, D. A., 1986; Denotational Semantics. Wm.C.Brown.

[15] Schroeder, W. J., W. E. Lorenson, G. D. Montanaro and C. R. Volpe, 1992; VISAGE: An object-oriented scientific visualization system, Proc. Visualization '92, 219-226.

[16] Scott, D. S., 1971; The lattice of flow diagrams. In Symposium on Semantics of Algorithmic Languages, E. Engler. ed. Springer-Verlag, 311-366.

[17] Scott, D. S., 1976; Data types as lattices. Siam J. Comput, 5(3), 522-587.

[18] Treinish, L. A., 1991; SIGGRAPH '90 workshop report: data structure and access software for scientific visualization. Computer Graphics 25(2), 104-118.