IN-61-CR
8147
P. 32

The VIS-AD Data Model: Integrating Metadata and Polymorphic Display with a Scientific Programming Language

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
whibbard@macc.wisc.edu

Abstract. The VIS-AD data model integrates metadata about the precision of values, including missing data indicators and the way that arrays sample continuous functions, with the data objects of a scientific programming language. The data objects of this data model form a lattice, ordered by the precision with which they approximate mathematical objects. We define a similar lattice of displays and study visualization processes as functions from data lattices to display lattices. Such functions can be applied to visualize data objects of all data types and are thus polymorphic.

1. Introduction

Computers have become essential tools to scientists. Scientists formulate models of natural phenomena using mathematics, but in order to simulate complex events they must automate their models as computer algorithms. Similarly, scientists analyze their observations of nature in terms of mathematical models, but the volumes of observed data dictate that these analyses be automated as computer algorithms. Unlike hand computations, automated computations are invisible, and their sheer volume makes them difficult to comprehend. Thus scientists need tools to make their computations visible, and this has motivated active development of scientific visualization systems. Explicitly or implicitly, these systems are based on:

1. A data model - how scientific data are defined and organized.
2. A computational model - how computations are expressed and executed.
3. A display model - how data and information are communicated to a the user.
4. A user model - the tasks and capabilities (e.g., perceptual) of users.
5. A hardware model - characteristics of equipment used to store, compute with, and display data.

Robertson et. al. [11] describe the need for a foundation for visualization based on such formal models. The user and hardware models help define the context and requirements for a system design, whereas the data, computational and display models are actually high level components of a system design. Because

scientists explore into unknown areas of nature, they need models of data, computation, and display that can adapt to change.

2. Data Model Issues

2.1 Levels of Data Models

A data model defines and organizes a set of data objects. Data models can be defined at various levels of functionality [16]. Data models can describe:

1. The physical layout and implementation of data objects. At the lowest level, a data model may describe the physical layout of bits in data objects. It is widely acknowledged that this level should be hidden from users, and even hidden from systems developers as much as possible. At a slightly higher level, a data model may describe the data objects of a visualization system in terms of the data objects of the programming language(s) used to implement the system.
2. The logical structure of data. This level describes the mathematical and logical properties of primitive data values, how complex data objects are composed from simpler data objects, and relations between data objects.
3. The behavior of data in computational processes. This is a pure object-oriented view of data. The internal structure of data objects is invisible, and all that is specified is the behavior of functions operating on data objects.

While purely behavioral models of scientific data are possible, it is rare to see a behavioral data model that does not refer to the logical structural of data. That is, the behaviors of functions operating on objects are usually explained in terms like "returns a component object" or "follows a reference to another object." In particular, most data models that are described as "object oriented" are object oriented implementations of structural data models. In these cases, the internal structure of objects is hidden in the sense of programming language scope rules, but is not hidden in the user's understanding of object behavior. The idea of defining complex things in terms of simpler things is extremely natural and convenient, so it is not surprising that most data models are essentially structural. Furthermore, structural data models permit automated analysis of data syntax (e.g., for query optimization), but it is difficult to apply similar analyses to purely functional specifications of data.

2.2 Structural Data Models

The physical and implementation levels address issues that should not be visible to scientists using a system, and purely behavioral data models are rare. Thus we focus on the structural level. At this level a data model needs to address the following issues:

1. The types of primitive data values occurring in data objects. A primitive type defines a set of primitive values. It may also define an order relation, basic operations (e.g., addition, negation, string concatenation), and a topology (e.g., the discrete topology of integers, the continuous topology of real numbers) on the set of values.
2. The ways that primitive values are aggregated into data objects. These may be simple tuples of values, they may be functional relations between variables, or they may be complex networks of values.
3. Metadata about the relation between data and the things that they represent. For example, given a meteorological temperature, metadata includes the fact that it is a temperature, its scale (e.g., Fahrenheit, Kelvin), the location of the temperature and whether it is a point or volume sample, the time of the temperature, an estimate of its accuracy, how it was produced (e.g. by a simulation, by direct observation, or deduced from a satellite radiance), and whether the value is missing (e.g., in case of a sensor failure).

A structural data model defines behavior rather than implementation, but does so in terms of an underlying structure. That is, primitive types describe the operations that can be applied to primitive objects, but do so under the assumption that the state of a primitive object is a simple mathematical value. Similarly, aggregate types describe operations that return objects as functions of other objects, but do so in terms of hierarchical and network relations between objects. A purely behavioral model would not place such constraints on operations on objects.

2.3 Extensive Versus Intensive Models for Types of Data Aggregates

The way that a structural data model defines types of data aggregates is an important issue in the context of data visualization. Many visualization systems define data models that are essentially finite enumerations of those aggregate types for which the systems have implemented display functions. For example, a visualization system's data model may include images, 3-D scalar and vector fields, vector and polygon lists, and color maps. On the other hand, scientists writing programs require flexibility in defining aggregate types. Thus programming languages usually define data models in terms a set of techniques that let users define their own (potentially infinite) sets of aggregate types. That is, users are given language features like tuples (i.e., structures in C), arrays and pointers for building their own structures. Visualization systems stress aspects of their data models related to display models, whereas programming languages stress aspects of their data models related to computational models.

In set theory, a set may be defined *extensively* as a list of members, or defined *intensively* by a logical condition for membership. We borrow these terms, saying that an extensive data model is one that defines a finite enumeration of aggregate types, and saying that an intensive data model is one that defines a set of techniques for building a potentially infinite set of aggregate types. Systems designed for particular applications, including many scientific visualization

systems, tend to define extensive data models, while programming languages tend to define intensive data models.

Scientists need data models that can support general computational models and can also support general display models for all data objects. Object oriented techniques provide one approach to this need. Each aggregate type in an extensive data model can be defined as a different object class. Inheritance between classes simplifies the task of designing new types of aggregates, and polymorphism allows analysis and display functions to be applied uniformly to many different aggregate types. However, this approach still requires users to explicitly define new object classes and their display functions. An approach based on intensive data models may be easier for scientists to use.

2.4 Models for Metadata

Programming languages and visualization systems differ in their level of support for metadata. While programming languages offer users the flexibility to build their own logic for managing metadata, they have no intrinsic semantics for the relation between data and its metadata. For example, scientists may adopt the convention that -999 represents a missing value, but since the programming languages that they use do not implement any special semantics for this value, their programs must include explicit tests for this value. On the other hand, many scientific visualization systems do intelligently manage the relation between data and its metadata. For example, some systems implement missing data codes, some systems manage information about the spatial locations of data (sometimes called data navigation), and some systems manage information needed to normalize observations to common scales (sometimes called data calibration).

3. Data Lattices

Mathematical models define infinite precision real numbers and functions with infinite domains, whereas computer data objects contain finite amounts of information and must therefore be approximations to the mathematical objects that they represent. For example, a 32-bit floating point number represents real numbers by a set of roughly 2^32 different values in the range between -10^38 and +10^38, plus a few special codes for illegal and out-of-range values. Since most real numbers are not members of this set of 2^32 values, they can only be approximately represented by floating point numbers. As another example, a satellite image is a finite sampling of a continuous radiance field over the Earth. The image contains a finite number of pixels, and pixels sample radiance values in finite numbers of bits (8-bit values are common). Thus the satellite image can only approximate the continuous radiance field. Satellites and other sensor systems are fallible, so scientists usually define missing data codes to represent values where sensors failed. These missing data codes may be interpreted as approximations that contain no information about the mathematical values that they represent.

We can define an order relation between data objects based on the fact that some are better approximations than others. That is, if x and y are data objects, then we define $x \leq y$ to mean that y is consistent with x, and that y provides more precise information than x does. We illustrate this order relation using closed real intervals as approximations to real numbers. If w is a real number, and if $[a, b]$ is the real interval of numbers between a and b, then $[a, b]$ is an approximation to w if w belongs to the interval $[a, b]$ (i.e., if $a \leq w \leq b$). Given two intervals $[a, b]$ and $[c, d]$, we say that $[a, b] \leq [c, d]$ if $[c, d] \subseteq [a, b]$. This is because the smaller interval provides more precise information about a value than the containing interval does. Letting the symbol $\perp$ represent a missing data code, then $\perp$ provides less precise information about a real value than any interval, so we can say that $\perp <$ $[a, b]$ for any interval $[a, b]$. Figure 1 shows a few closed real intervals and the order relations among those intervals.

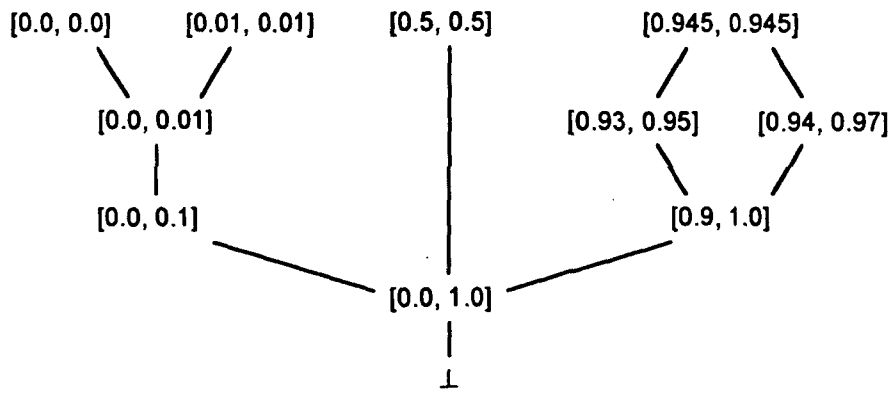

Figure 1. Closed real intervals are used as approximate representations of real numbers, ordered by the inverse of containment (i.e., containing intervals are "less than" contained intervals). We also include a least element $\perp$ that corresponds to a missing data indicator. This figure shows a few intervals, plus the order relations among those intervals. The intervals in the top row are all maximal, since they contain no smaller interval.

We interpret arrays as finite samplings of functions. For example, a function of a real variable may be represented by a set 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])} contains three samples of a function. The domain value of a sample lies in the first interval of a pair and the range values lies in the second interval of a pair, as illustrated in Fig. 2. Adding more samples, or increasing the precision of samples, will create a more precise approximation to the function. Figure 3 shows the order relations between a few array data objects.

In general we can order arrays to reflect how precisely they approximate functions. If x and y are two array data objects that are both finite samplings of a function, and if, for each sample of x, there is a collection of samples of y that improve the resolution of the function's domain and range over the sample of x, then $x \leq y$. Intuitively, y contains more information about the function than x does.

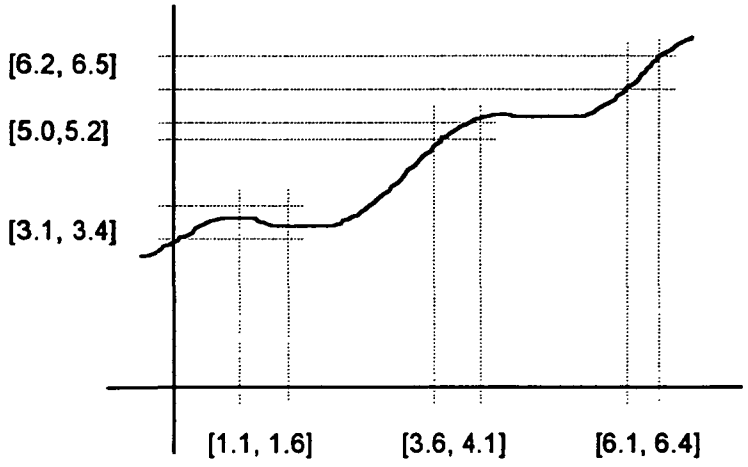

Figure 2. An approximate representation of a real function as a set of pairs of intervals.

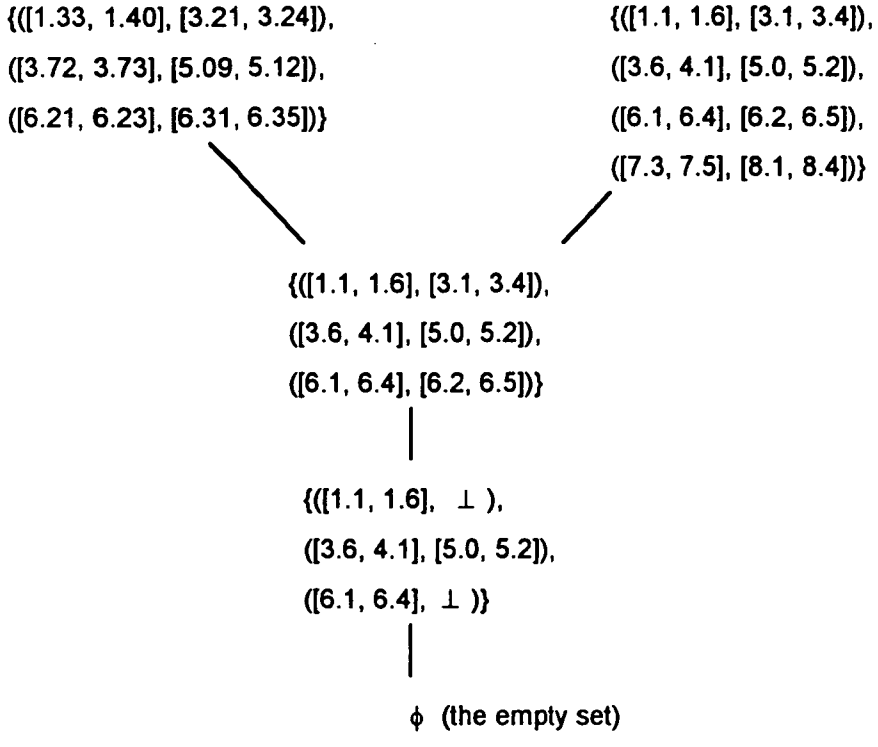

Figure 3. A few finite samplings of functions, and the order relations among them.

3.1 A Scientific Data Model

Now we will define a data model that is appropriate for scientific computations. We describe this data model in terms of the way it defines primitive values, how those values are aggregated into data objects, and metadata that describes the relation between data objects and the mathematical objects that they represent.

The data model defines two kinds of primitive values, appropriate for representing real numbers and integers. We call these two kinds of primitives *continuous scalars* and *discrete scalars*, reflecting the difference in topology between real numbers and integers. A continuous scalar takes the set of closed real intervals as values, ordered by the inverse of containment, as illustrated in Fig. 1. A discrete scalar takes any countable set as values, without any order relation between them (since no integer is more precise than any other). Figure 4 illustrates the order relations between values of a discrete scalar. Note that discrete scalars may represent text strings as well as integers. The value sets of continuous and discrete scalars always include a minimal value $\perp$ corresponding to missing data.

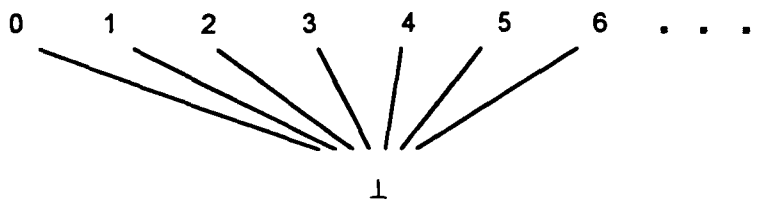

Figure 4. A discrete scalar is a countable (possibly finite) set of incomparable elements, plus a least element $\perp$.

Our data model defines a set T of data types as ways of aggregating primitive values into data objects. Rather than enumerating a list of data types in T, the data model starts with a finite set S of scalar types, representing the primitive variables of a mathematical model, and defines three rules by which data types in T can be defined. These rules are:

1. Any continuous or discrete scalar in S is a data type in T. A scientist using this data model typically defines one scalar type in S for each variable in his or her mathematical model.
2. If t_1, ..., t_n are types in T, then *struct*$\{t_1;...;t_n\}$ is a *tuple* type in T with *element* types t_i. Data objects of tuple types 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, then (*array* [w] *of* r) is an *array* type with *domain* type w and *range* type r. Data objects of array types are finite samplings of functions from the primitive variable represented by their domain type to the set of values represented by their range type. That is, they are sets of data objects of their range type, indexed by values of their domain type.

Each data type in T defines a set of data objects. Continuous and discrete scalars define sets of values as we have described previously. The set of objects of a tuple type is the cross product of the sets of objects of its element types. The set of objects of an array type is not quite the space of all functions from the value set of its domain type to the set of objects of its range type. Rather, it is the union of such function spaces, taken over all finite subsets of the domain's value set.

A tuple of data objects represent 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 5 illustrates the order relations between a few tuples.

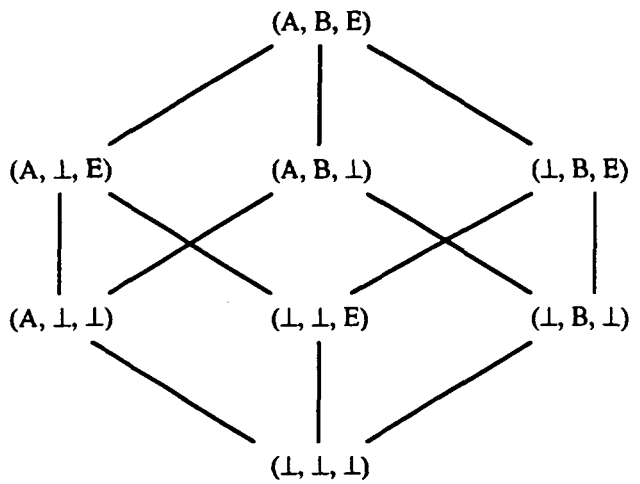

Figure 5. Defining an order relation on a cross product. Members of cross products are tuples. This figure shows a few elements in a cross product of three sets, plus the order relations among those elements. In a cross product, the least element is the tuple of least elements of the factor sets.

An array data object is a finite sampling of a function, and the precision of approximation depends on how precisely the function's domain is sampled and the precision of the array's range values. If an array is indexed by a continuous scalar, the interval values of the index indicate how precisely the function's domain is sampled, as illustrated in Figs. 2 and 3.

By building hierarchies of tuples and arrays, it is possible to define data types in T that represent virtually any mathematical model used in the physical sciences. For example, consider a set of data types appropriate for analyzing meteorological observations. The scalar types used to represent primitive variables for this analysis include:

temperature - thermometer reading (continuous)

dew_point	- wet bulb thermometer reading (continuous)
pressure	- barometer reading (continuous)
count	- frequency count of values in a histogram (discrete)
station_name	- name of observing station (discrete)
latitude	- latitude of observing station (continuous)
longitude	- longitude of observing station (continuous)
time	- time of observation (continuous)

The complex data types for this analysis include:

```
station_reading = struct{
                        .sta_temp = temperature;
                        .sta_dew = dew_point;
                        .sta_pres = pressure;
                }
station_series = (array [time] of station_reading)
station_set = (array [station_name] of
                struct{
                        .set_series = station_series;
                        .set_lat = latitude;
                        .set_lon = longitude;
                }
temperature_histogram   = (array [temperature] of count)
```

A data object of the *station_reading* type includes one value for each instrument at a weather observing station. A data object of type *station_series* contains a sequence of *station_reading* objects, that finitely sample continuous functions of meteorological fields over *time*. A data object of type *station_set* is an array that associates a time series of readings and *latitude* and *longitude* locations to each of a finite set of *station_names*. A data object of the *temperature_histogram* type contains frequency *counts* of intervals of *temperatures*. In this case, the interval values of the *temperature* represent the bins used for histogram calculation.

The lattice data model defines certain metadata about the relation between data objects and the mathematical objects that they represent, including:

1. Every primitive value in a data object is identified by the name of the primitive mathematical variable.
2. An array data object is a finite sampling of a mathematical function. The set of index values of the array 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 intervals provide information about the accuracy of their approximations.

4. Any scalar data object may take the missing value (denoted by $\perp$) and this provides information about accuracy (i.e., the fact that the value has no accuracy).

3.2 Interpreting the Data Model as a Lattice

We view a data display 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 display function must be defined for each different data type. However, we can define a lattice U of data objects and natural embeddings of data objects of all data types into U. The lattice U provides us with a unified model for all of our scientific data objects, and enables us to define display functions that are applicable to all data types (i.e., these display functions are polymorphic). Our analysis of the properties of display functions will thus be independent of particular data types.

A *lattice* is an ordered set U in which every pair of elements x and y has a least upper bound $sup\{x, y\}$ [this is z such that $x \le z, y \le z$ and $\forall w \in U. (x \le w \ \& \ y \le w \Rightarrow z \le w)$] and a greatest lower bound $inf\{x, y\}$. A lattice U is *complete* if it contains the least upper bound $sup(A)$ and the greatest lower bound $inf(A)$ for any subset $A \subseteq U$.

We define a data lattice U whose members are sets of tuples. The primitive domains of this data lattice are defined by a finite set S of scalar types, and the tuple space is the cross product of the sets of values of the scalar types in S. Define I_s as the set of values of a scalar $s \in S$ and define $X = \mathbf{X}\{I_s \mid s \in S\}$ as the cross product of these scalar value sets. Members of our data lattice are subsets of X. Figures 1 and 4 illustrate the order relations on the scalar value sets I_s, and Fig. 5 illustrates the order relation on the set X of tuples.

Members of U are subsets of X. However, there is a problem with defining an order relation between subsets of X that is consistent with the order relation on X and is 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:

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

However, given $a < b$, (1) implies that $\{b\} \le \{a, b\}$ and $\{a, b\} \le \{b\}$ are both true, which contradicts $\{b\} \ne \{a, b\}$. 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 $a, b \in A$. (Actually, the situation is a bit more complex - see [7] for the details.) The members of U are ordered by (1), as illustrated in Fig. 6, and form a complete lattice.

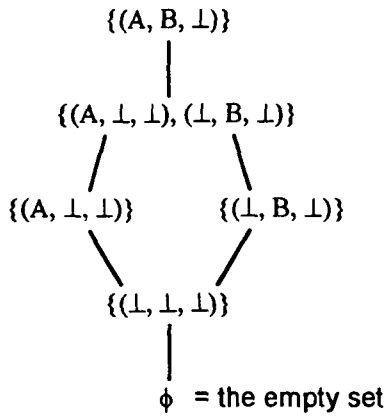

Figure 6. A few members of a data lattice U defined by three scalars, and the order relations between them.

To get an intuition of how data types are embedded in the lattices, 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 (*temperature*, *pressure*) and can be modeled as a set of tuples (actually, it is a set consisting of one tuple) in U with the form {(⊥, *temperature*, *pressure*)}. This embeds the tuple data type in the lattice U, and Fig. 7 illustrates this embedding.

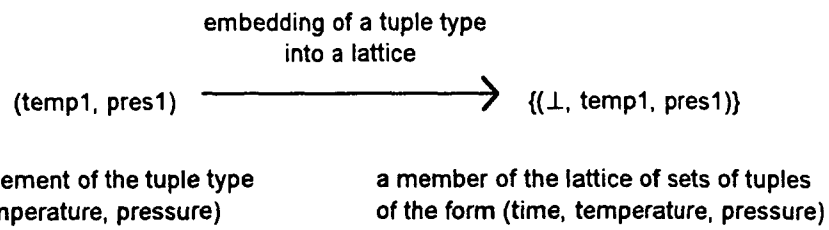

Figure 7. An embedding of a tuple type into a lattice of sets of tuples.

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 consists of a set of pairs of (*time*, *temperature*). This array data object can be embedded in U as a set of tuples of the form (*time*, *temperature*, ⊥). 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. The details are explained in [6] and [7].

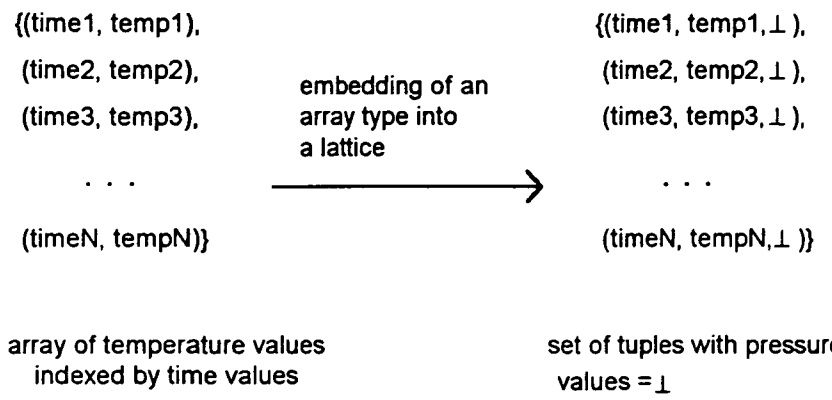

Figure 8. An embedding of an array type (a functional dependency between scalar types) into a lattice of sets of tuples.

If $x \in X$ is the embedding of a data object of a type $t \in T$, and if the scalar s does not occur in the definition of t, then the s values of all the tuples in x will be $\perp$. Also, in order to embed data objects in the data lattice U, we must restrict T to the set of data types t such that no scalar s occurs more than once in the definition of t. We note that, for each type in $t \in T$, the embedding of data objects of type t into U is an order embedding. This means that if a and b are objects of type t then $a \leq b$ if and only if $E_t(a) \leq E_t(b)$, where is E_t is the embedding of objects of type t.

Lattices and other kinds of ordered sets have played an important role in the denotational semantics of programming languages [2, 12, 13, 14, 15], and they can also play an important role in visualization.

3.3 Display Lattices

Our lattice structure can also be used to model displays. This is motivated by analogy with the display model of Bertin [1]. He defined a display as a set of graphical marks, and identified eight primitive variables of a graphical mark: two spatial coordinates of the mark in a graphical plane (he restricted his attention to static 2-D graphics), plus size, value, texture, color, orientation, and shape. Bertin defined diagrams, networks and maps as spatial aggregates of graphical marks. By defining a graphical mark as a tuple of its graphical primitive values, a display can be viewed as a set of tuples.

We define a finite set DS of display scalars that represent graphical primitives and we interpret a tuple of values of the display scalars as a graphical mark. Similar to the data lattice U, we define a display lattice V whose members are sets of tuples of values of display scalars.

We can define a display lattice for static 2-D displays using five continuous display scalars: two for image coordinates plus three for color components (e.g., red, green and blue). In this model, a display is a set of colored rectangles. The interval values of the image coordinate scalars in a tuple specify the size and location of the

rectangle on the screen, and the interval values of the color component scalars specify the range of colors used in the rectangle. This model can be extended to dynamic 3-D displays, by adding two more display scalars: one for a third image coordinate and another for indicating a graphical mark's location in an animation sequence. The three image coordinates then specify the locations and sizes of 3-D rectangles that must be projected onto a 2-D display screen (where multiple rectangles are projected to the same screen location, their colors must be combined according to some compositing algorithm). The values of the animation scalar are used to select tuples for display. At any instant during data display, an animation index takes an interval value, and only those tuples whose animation scalar intervals overlap this animation index value are displayed. By sequencing through values of the animation index, the display screen contents will change, providing a dynamic display. Figure 9 illustrates the role of the various display scalars in this display model.

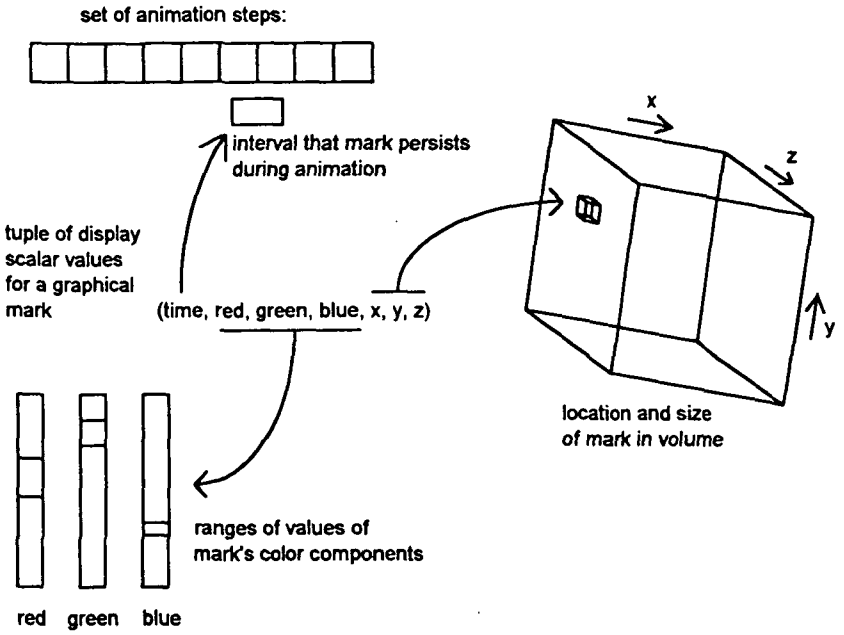

Figure 9. The roles of the display scalars in an animated 3-D display model.

Just like computer data objects, computer displays contain finite amounts of information. Pixels and voxels have limited resolution, colors are specified with limited precision, animation sequences consist of finite numbers of steps, etc. The lattice structure of V orders displays based on their information content.

Display models need not be limited to such primitive values as spatial coordinates, color components and animation indices. For example, consider a display model where a display consists of a set of graphical icons distributed at various locations in a display screen. This display model could be defined using

three display scalars: a horizontal screen coordinate, a vertical screen coordinate, and an icon identifier. Then a single value of the icon identifier display scalar would represent the potentially complex shape of a graphical icon. Or, a set of display scalars may form the parameters of a complex graphical shape. For example, 2-D ellipses may be used as graphical marks, parameterized by five display scalars for their center coordinates, orientations, and the lengths of their major and minor axes.

3.4 Data Display as a Mapping From a Data Lattice to a Display Lattice

We model a display process as a function $D:U \rightarrow V$ that generates a display in V from any data object in U. Rather than defining such functions constructively, in terms of algorithms for calculating a display $D(u)$ from a data object $u \in U$, we will define conditions on D and study the class of functions satisfying those conditions. For our conditions, we interpret Mackinlay's *expressiveness* conditions [8] in the lattice context. These conditions require that a display encode all the facts about a data object, and only those facts. As we show in [6] and [7], we can interpret these conditions as:

Condition 1. $\forall P \in MON(U \rightarrow \{\perp, 1\})$.
$\exists Q \in MON(\downarrow D(MAX(X)) \rightarrow \{\perp, 1\}).\ P = Q \circ D$

Condition 2. $\forall u \in U.\ D(u) \in \downarrow D(X)$ and $\forall Q \in MON(\downarrow D(MAX(X)) \rightarrow \{\perp, 1\})$.
$\exists P \in MON(U \rightarrow \{\perp, 1\}).\ Q = P \circ D^{-1}$

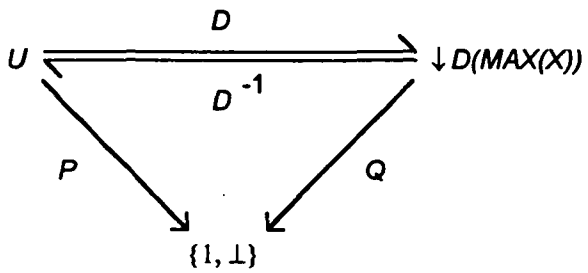

Figure 10. Expressiveness Conditions 1 and 2 interpreted as a commuting diagram. The conditions require that a display function D generate a one-to-one correspondence between the set of monotone functions P and the set of monotone functions Q, going both ways around the diagram.

Here $MON(U \rightarrow \{\perp, 1\})$ is the set of monotone functions from U to the set $\{\perp, 1\}$, $MAX(X)$ is the set of maximal tuples in X and thus the maximal element of U, and $\downarrow D(MAX(X))$ is the set of all displays in V less than the display of $MAX(X)$. A function P is monotone if $x \leq y$ implies $P(x) \leq P(y)$. We interpret facts about data objects as functions in $MON(U \rightarrow \{\perp, 1\})$ and we interpret fact about displays as functions in $MON(V \rightarrow \{\perp, 1\})$ (however, we limit this to displays less than the

display of the maximal data object). Condition 1 says that for every P there is a Q that makes the diagram in Fig. 10 commute, and Condition 2 says that for every Q there is a P that makes the diagram commute.

We say that a function $D{:}U{\rightarrow}V$ is a *display function* if it satisfies Conditions 1 and 2. In [7] we prove:

Proposition 1. $D{:}U{\rightarrow}V$ is a display function if and only if it is a lattice isomorphism from U onto $\downarrow D(MAX(X))$, which is a sub-lattice of V.

The definition of *display function*, and the proof of this proposition, do not refer to the construction of data and display lattices in terms of scalars (although that construction motivates some of the discussion). The set $MAX(X)$ plays a role in the definition of *display function* and in our proofs, but only as the maximal element of the lattice U. Since any complete lattice has a maximal element (i.e., the *sup* of all its elements), this result is true for any pair of complete lattices U and V.

In the special case that the lattice U and V are constructed from scalars and display scalars as described in Sects. 3.2 and 3.3, display functions can be characterized by simple mappings from scalars to display scalars. Specifically, for a scalar $s \in S$, define an embedding $E_S{:}I_S{\rightarrow}U$ by $E_S(b) = (\perp,...,b,...,\perp)$ (this notation indicates that all components of the tuple are $\perp$ except b) and define $U_S = E_S(I_S) \subseteq U$. Similarly, for a display scalar $d \in DS$, define an embedding $E_d{:}I_d \rightarrow V$ by $E_d(b) = (\perp,...,b,...,\perp)$ and define $V_d = E_d(I_d) \subseteq V$. These embedded scalar objects play a special role in the structure of display functions. In [7] we prove:

Proposition 2. 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 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 (see [7] for the details). 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 as illustrated in Fig. 11. The remarkable thing is that Prop. 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 12 in Sect. 4.4 provides a more detailed example of how a display function is defined by a set of mappings from scalars to display scalars.

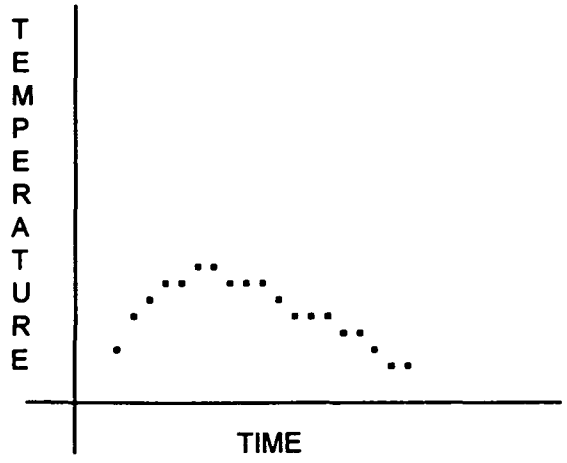

Figure 11. A time series displayed as a graph.

Display functions of the form $D:U \rightarrow V$ are polymorphic in that sense that they can be applied to data objects of any type in T. Furthermore, our lattice results show that we can define such functions in terms of a set of mappings from scalars to display scalars. Just as data flow systems define a user interface for controlling how data are displayed based on the abstraction of the rendering pipeline, we can define a user interface for controlling how data are displayed based on the abstraction of scalar mappings.

4. The VIS-AD Data Model

The VIS-AD (VISualization for Algorithm Development) system was designed to help scientists visualize their computations [5]. The system can be understood in terms of its data model, computational model and display model. The VIS-AD computational model is an imperative programming language of the type familiar to scientists (it is similar to C). The system's data model is based on the data lattice defined in Sect. 3.2. The data types and data objects of the lattice are just the types and objects of the VIS-AD programming language. Furthermore, metadata is integrated into this data model and plays a special role in the semantics of the programming language.

The biggest difference between the VIS-AD data model and the data lattice defined in Sect. 3.2 is the way that users define scalar types in S. A VIS-AD program defines scalar types as *real*, *real2d*, *real3d*, *int* or *string*. The *int* and *string* scalars are discrete scalars, and the *real* scalars are continuous scalars. The *real2d* and *real3d* scalars take pairs and triples of real intervals as values, and were included in the VIS-AD system to simplify the definition of spatial data types (e.g., the scalar *latitude_longitude* defined in the next section is a *real2d* scalar used as an index for 2-D image arrays).

4.1 Examples of User Defined Data Types

Users of VIS-AD can build types as arbitrary hierarchies of tuples and arrays, which provides the flexibility to adapt to scientific applications. VIS-AD's two- and three-dimensional real scalars make it easier to define types for spatial data like satellite images. The VIS-AD programming language provides a simple syntax for defining data types, as illustrated by the following examples taken from an algorithm for discriminating clouds in time series of multi-channel GOES (Geostationary Operational Environmental Satellite) images.

```
type ir_radiance = real;
type vis_radiance = real;
type latitude_longitude = real2d;
type time = real;
type image_region = int;
type count = int;
type goes_image =
  array [latitude_longitude] of
    structure {
      .im_ir = ir_radiance;
      .im_vis = vis_radiance;
    }
type goes_partition = array [image_region] of goes_image;
type goes_sequence = array [time] of goes_partition;
type histogram = array [ir_radiance] of count;
type histogram_partition =
  array [image_region] of
    structure {
      .hist_loc = latitude_longitude;
      .hist_hist = histogram;
    }
```

In these examples, a *goes_image* data object is an array of pixels indexed by *latitude_longitude* values, where each pixel is a structure containing infrared and visible radiances. A *goes_partition* object divides an image into regions, and includes a *goes_image* object for each value of *image_region*. A *goes_sequence* object is a *time* sequence of *goes_partition* objects. A *histogram* data object provides a frequency *count* of the number of occurrences of each *ir_radiance* value in an *image_region*, and a *histogram_partition* object associates a *histogram* object and a *latitude_longitude* value with each *image_region*.

4.2 Integrating Metadata with Programming Language Semantics

Unlike the situation in other programming languages, VIS-AD's arrays may be indexed by real values, or even by two- or three-dimensional real values. This is because VIS-AD's array types are defined as finite samplings of functional

relations from variables (i.e., from scalar types) to other data types. Thus metadata about the sampling of values is built into the semantics of the VIS-AD programming language. This has important consequences for the way that scientific data are manipulated and displayed. For example, an Earth satellite image is really a finite sampling of a continuous radiance field. If the pixels of an image are stored in an array in an ordinary language, the pixels are indexed in the array by integers, and the Earth locations of pixels must be managed separately. Thus the programming language has no information about the association between pixel values and their locations. However, if this satellite image is stored in a *goes_image* object, then the pixels are indexed with *latitude_longitude* values, and the programming language does have access to the locations of pixels. This enables the system to display a *goes_image* object in an Earth based frame of reference. If images from different sources (each with its own Earth projection) are overlaid in a display, the system can use the information about pixel locations to geographically register these images.

In the VIS-AD data model, all scalar values are managed in terms of finite samplings of infinite value sets. In addition to determining the values of array indices, this also determines the sampling and accuracy of values in arrays and tuples. For example, if a satellite sensor generates radiances as 8-bit quantities, then pixel values are really indices into a set of 256 samples of real radiance values. The scale of these real values may be a standard radiance, in which case the set of 256 values encodes the calibration of the satellite's sensor. Thus VIS-AD's management of sampling information can be used to encode satellite navigation and calibration information. Furthermore, sensor systems are fallible, so it is often the case the no value is defined for some pixels. In the VIS-AD data model, any data object or sub-object may take the special value *missing*, indicating the absence of information. Because missing values are part of the data model, they can be part of programming language semantics and display semantics.

We will use a satellite image example to illustrate how sampling information and missing data indicators are integrated with programming language semantics. In this example, we calculate the difference between images generated by different satellites. Let *goes_east*, *goes_west* and *goes_diff* be data objects of type *goes_image*, and let *loc* be a data object of type *latitude_longitude*. Assume that the *goes_east* and *goes_west* images were generated by GOES satellites at East and West stations over the U.S., so that they have different Earth perspectives. Then the following program calculates the difference between these images:

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

The first line specifies that *goes_diff* will have the same sampling of array index values (i.e., of pixel locations) that *goes_east* has. The *foreach* statement provides a way to iterate over the elements of an array. In this case it iterates *loc* over the

pixel locations of the *goes_east* image. The expression *goes_west*[*loc*] resamples the *goes_west* image at the Earth location in *loc*. If *loc* falls in an area where there are no *goes_west* pixels, then *goes_west*[*loc*] evaluates to *missing*. VIS-AD's arithmetic operations evaluate to *missing* if any of their operands are *missing*, so if *goes_west*[*loc*] is *missing* then the difference *goes_diff*[*loc*] is also set to *missing*.

The VIS-AD programming language provides vector operations, so this little program can also be expressed as:

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

The resampling of *goes_west* index values, and the evaluation to *missing* where there are no *goes_west* pixels, are implicit in this statement.

Users can access metadata about sampling and missing data explicitly. For example, the statement:

```
foreach (loc in goes_east) { ... }
```

iterates *loc* over the samples of the *goes_east* array. Missing data indicators may be explicitly accessed using these statements:

```
if (goes_east == missing) { ... }
goes_east[loc].im_ir = missing;
```

However, because of the special role of metadata in the semantics of the VIS-AD programming language, users rarely need to do explicit calculations with this metadata.

The integration of sampling information and *missing* data is generic, rather than specific to images. Thus the techniques illustrated in this satellite image example can be applied to any user-defined data types. As our simple programming example shows, this can relieve users of the need to explicitly keep track of missing data, the need to manage the mapping from array index values to physical values, and the need to check bounds on array accesses. The key to these advantages is that metadata is integrated into the data semantics of a programming language.

We can summarize the kinds of metadata that are integrated with the VIS-AD data model. They are:

1. Sampling information; every value in a data object is taken from a finite sampling of primitive values.
2. Missing data indicators; any value or sub-object in a data object may take the special value *missing* which indicates the lack of information.
3. Names for values; every primitive value occurring in a data object has a scalar type, and hence a name (i.e., the name of the scalar type).

Because these kinds of metadata are integrated with the data model, they are part of the computational and display semantics of the VIS-AD system. Note that the

VIS-AD programming language semantics do not integrate the accuracy information of interval values. However, this accuracy information could be integrated using interval arithmetic [10].

4.3 Other Types of Metadata

There are a great variety of kinds of metadata that scientists use to interpret their data. While some of these are integrated with the VIS-AD data model, the flexibility to define data types gives users a means to include other kinds of metadata in their data objects. For example, users of satellite images may want to manage the following kinds of information with their images:

1. Sensor identification. Satellites often have redundant sensors for measuring the same radiances, each with slightly different characteristics. Scientists sometimes need to know which sensor was used to generate a particular image.
2. Satellite sub-point. This is the Earth location (i.e., *latitude_longitude*) directly under the satellite, and is useful as a rough guide to image coverage.
3. Pixel scan rate. Images are often scanned over a significant time interval, and the scan rate in pixels per second can help assign precise times to pixel radiances.
4. Various measurements of the sensor systems, like voltages, temperatures and pressures. These are often used to diagnose problems with image quality.

We can create a new image type that includes these kinds of information, as follows:

```
type ir_radiance = real;
type vis_radiance = real;
type latitude_longitude = real2d;
type pixel_rate = real;
type sensor_id = string;
type temperature = real;
type voltage = real;
type annotated_goes_image =
 structure {
   .image_sensor = sensor_id;
   .image_subpoint = latitude_longitude;
   .image_pixel_rate = pixel_rate;
   .image_sensor_temp = temperature;
   .image_sensor_cathode = voltage;
```

```
.image_data =
  array [latitude_longitude] of
    structure {
      .im_ir = ir_radiance;
      .im_vis = vis_radiance;
    }
}
```

While these kinds of metadata are not part of the semantics of the programming language, they are part of data objects and can be accessed by users' programs.

4.4 Data Display in VIS-AD

The VIS-AD display model is similar to the display lattice V described in Sect. 3.3, and is realized as a set of interactive, animated, 3-D voxel volumes. It is defined in terms of a set of display scalars that include:

- x, y and z coordinates of graphical marks in a 3-D volume
- *color* values of graphical marks
- a set of *contour* values; for each *contour* display scalar iso-surfaces and iso-lines are interpolated through the graphical marks in the 3-D volume
- an *animation* value; graphical marks whose *animation* value overlaps an animation index are selected for display
- a set of *selector* values, used to model abstract user control over display contents; the user selects a set of values for each *selector* display scalar, and only those graphical marks that overlap that set are displayed

Figure 12 illustrates the way that user's of VIS-AD control how their data types are displayed. An *image_sequence* data object is a time sequence of images with two spectral channels called *ir* (infrared) and *vis* (visible). Image pixels are indexed by pairs of real numbers specifying their Earth locations. Users define mappings from the scalar types of their application to the display scalar types that define the VIS-AD display model. The mappings indicated by arrows in Fig. 12 will cause an *image_sequence* data object to be displayed as an animated sequence of colored terrains, where the *ir* channel will determine the height of the terrain and the *vis* channel will determine its color. Users can interactively change the mappings from scalars to display scalars (e.g., change the mappings in Fig. 12 by mapping *ir* to *color* and mapping *time* to y - this will create a time series of images stacked up along the y axis). They can also interactively control the functions by which scalar values determine display scalar values (e.g., by adjusting color tables for the mapping of *vis* to *color* in Fig. 12). Data objects may be displayed according to multiple sets of mappings simultaneously.

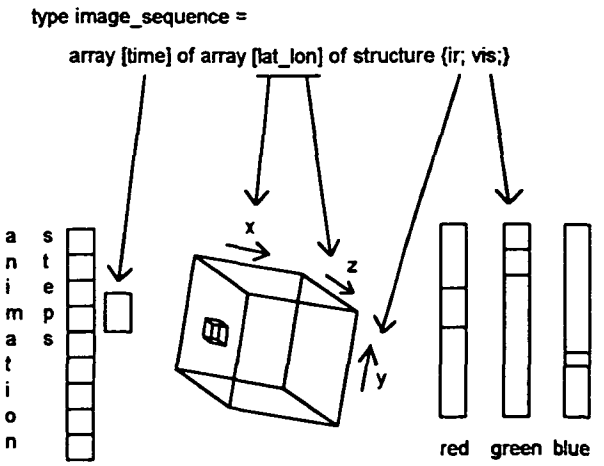

Figure 12. Users of VIS-AD control how data are displayed by defining mappings from the scalar types used to define complex data types to the display scalars used to define the VIS-AD display model.

The interface for controlling displays, consisting of the definitions of scalar mappings, is de-coupled from the VIS-AD programming language. This is important, because it allows users to control the display of their data objects without embedding explicit calls to display functions in their programs.

The VIS-AD system is available by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/visad directory. The README file contains instructions for retrieving and installing the system.

5. Extending Data Lattices to More Complex Data Models

In Sect. 3.4 we described how a function $D:U \rightarrow V$ satisfying the expressiveness conditions must be a lattice isomorphism. Although we motivated this result in the context of a specific lattice structure for U and V (i.e., their members are sets of tuples of scalar values), the proof of this result only depends on U and V being complete lattices. Thus it is natural to seek to apply this result to other lattice structures for data and display models. The motive for new lattice structures must be new data models, since display models are themselves motivated by the need to visualize data. The data model defined in Sect. 3 includes tuples and arrays as ways of aggregating data. We will describe the issues involved in extending data lattices to data types defined by recursive domain equations, to abstract data types, and to the object classes of object-oriented programming languages. This discussions of this section are somewhat speculative.

5.1 Recursive Data Types Definitions

The denotational semantics of programming languages provides techniques for defining ordered sets whose members are the values of programming language expressions [4, 13, 14, 15]. An important topic of denotational semantics is the study of *recursive domain equations*, which define *cpos* (defined in the next paragraph) in terms of themselves.

First, we present some definitions used in denotational semantics. A *partially ordered set* (*poset*) is a set D with a binary relation $\leq$ on D such that, $\forall x, y, z \in D$

$x \leq x$	"reflexive"
$x \leq y \,\&\, y \leq x \Rightarrow x = y$	"anti-symmetric"
$x \leq y \,\&\, y \leq z \Rightarrow x \leq z$	"transitive"

A subset $M \subseteq D$ is *directed* if, for every finite subset $A \subseteq M$, there is an $x \in M$ such that $\forall y \in A.\ y \leq x$. A *poset* D is *complete* (and called a *cpo*) if every directed subset $M \subseteq D$ has a least upper bound $sup(M)$ and if there is a least element $\perp \in D$ (i.e., $\forall y \in D.\ \perp \leq y$). If D and E are *posets*, a function $f{:}D \rightarrow E$ is *monotone* if $\forall x, y \in D.$ $x \leq y \Rightarrow f(x) \leq f(y)$. A function $f{:}D \rightarrow E$ is *continuous* if it is monotone and if $f(inf(M)) = inf(f(M))$ for all directed $M \subseteq D$. If D and E are *cpos*, a pair of continuous functions $f{:}D \rightarrow E$ and $g{:}E \rightarrow D$ are a *retraction pair* if $g \circ f \leq \mathbf{id}_D$ and $f \circ g = \mathbf{id}_E$. The function g is called an *embedding*, and f is called a *projection*.

We take the following example of a recursive domain equation from [12]. A data type for a binary tree may be defined by:

$$Bintree = (Data + (Data \times Bintree \times Bintree))_{\perp} \qquad (2)$$

Here "+", "×" and "$(.)_{\perp}$" are type construction operators similar to the tuple and array operators we discussed in Sect. 3.1. The "+" operator denotes a type that is a choice between two other types (this is similar to the *union* type constructor in the C language), "×" denotes a type that is a cross product of other types (this is essentially the same as our tuple operator, so that (*Data* × *Bintree* × *Bintree*) is a 3-tuple), and the "$\perp$" subscript indicates a type that adds a new least element $\perp$ to the values of another type. Equation (2) defines a data type called *Bintree*, and says that a *Bintree* data object is either $\perp$, a data object of type *Data*, or a 3-tuple consisting of a data object of type *Data* and two data objects of type *Bintree*. Intuitively, a data object of type *Bintree* is either missing, a leaf node with a data value, or a non-leaf node with a data value and two child nodes.

The obvious way to implement binary trees in a common programming language is to define a record or structure for a node of the tree, and to include two pointers to other tree nodes in that record or structure. In general, the self references in recursive type definitions can be implemented as pointers. Thus, recursive domain equations correspond to defining data types with pointers.

5.1.1 The Inverse Limit Construction

The equality in a recursive domain equation is really an isomorphism. As explained quite clearly by Schmidt in [12], these equation may be solved by the *inverse limit construction*. This construction starts with $Bintree_0 = \{\perp\}$, then applies (2) repeatedly to get

$$Bintree_1 = (Data + (Data \times Bintree_0 \times Bintree_0))_\perp$$
$$Bintree_2 = (Data + (Data \times Bintree_1 \times Bintree_1))_\perp$$
etc.

The construction also specifies a retraction pair $(g_i, f_i):Bintree_i \leftrightarrow Bintree_{i+1}$ for all i, such that g_i embeds $Bintree_i$ into $Bintree_{i+1}$ and f_i projects $Bintree_{i+1}$ onto $Bintree_i$. Then *Bintree* is the set of all infinite tuples of the form $(t_0, t_1, t_2, ...)$ such that $t_i = f_i(t_{i+1})$ for all i. It can be shown that *Bintree* is isomorphic with $(Data + (Data \times Bintree \times Bintree))_\perp$, and thus "solves" the recursive domain equation. The order relation on the infinite tuples in *Bintree* is defined element-wise, just like the order relation on finite tuples defined in Sect. 3.1, and *Bintree* is a *cpo*. We note that the inverse limit construction can also be applied to solve sets of simultaneous domain equations.

One way of extending our data lattices would be to show how to apply the inverse limit construction to recursive equations involving our tuple and array type constructors. Our tuple constructor is equivalent to the cross product operator "×". While our array constructor is similar to the function space operator "→" used in denotational semantics, it is not the same. $(A \rightarrow B)$ defines the set of all functions from A to B, while our array constructor (*array* [A] *of* B) defines the set of functions from finite subsets of A to B. Thus we would need to show how to apply the inverse limit construction to equations involving the constructor (*array* [A] *of* B). The *cpos* defined by the inverse limit construction are generally not lattices, but can always be embedded in complete lattices. Specifically, the Dedekind-MacNeille completion, described in [2], shows that for any partially ordered set A, there is always a complete lattice U such that there is an order embedding of A into U.

Note that the set of *Bintree* objects defined by the inverse limit construction includes infinite trees. This is because this set is complete and infinite trees are limits of infinite sequences of finite trees. The development of denotational semantics was largely motivated by the need to address non-terminating computations (the unsolvability of the halting problem showed that there was no way to separate terminating from non-terminating computations), and non-terminating computations may produce infinite trees as their values. Since our result that display functions are lattice isomorphisms depends on the assumption that data and display lattices are complete, it is likely that any extension of our data lattice to include solutions of recursive domain equations must include infinite data objects.

The inverse limit construction defines the set of data objects of a particular data type that solves a particular recursive domain equation. However, our approach in Sect. 3.2 was to define a large lattice that contained data objects of many different data types. It would be useful to continue this approach, by defining a lattice that includes all data types that can be constructed from our scalar types as tuples, arrays, and solutions of recursive domain equations. This is the subject of Sect. 5.1.2.

5.1.2 Universal Domains

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

Scott developed an elegant way to solve recursive domain equations by applying the fixed point theorem [4, 14]. In a sense, the solution of a recursive domain equation is just a fixed point of a function that operates on *cpos*. Scott defined a *universal domain* U and a set R of retracts of U (this may be the set of all retracts on U, the set of projections, the set of finitary projections, the set of closures, or the set of finitary closures - note that these terms are defined in [7]). Then he showed that a set OP of type construction operators (these operators build *cpo*'s from other *cpo*'s) can be represented by continuous functions over R, in the sense that for $o \in OP$ there is a continuous function f on R that makes the diagram in Fig. 13 commute.

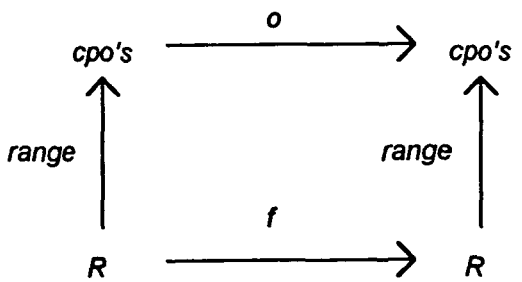

Figure 13. The type construction operator o is represented by function f.

Note that $range(w) = \{w(u) \mid u \in U\}$. For unary $o \in OP$ this is $range(f(w)) = o(range(w))$. Similar expressions hold for multiary operators in OP. Then, for any recursive domain equation $D = O(D)$ where O is composed from operators in OP, there is a continuous function $F:R\rightarrow R$ that represents O. By the fixed point theorem, F will have a least fixed point $fix(F)$, and $O(range(fix(F))) = range(F(fix(F))) = range(fix(F))$, so $range(fix(F))$ is a *cpo* satisfying the recursive domain equation $D = O(D)$. The solution of any domain equation (or any set of simultaneous domain equations) involving the type construction operators in OP

will be a *cpo* that is a subset of the universal domain U. Thus this approach is similar to the way that the data types of our data model define sets of data objects that are embedded in a single data lattice.

Universal domains and representations have been defined for sets OP that include most of the type constructors used in denotational semantics, including "+", "×", "→", and "$(.)_\perp$". In order to apply universal domains to extend our data model to include recursively defined types, we would need to show how our tuple and array type constructors can be represented over some universal domain.

A common example of a universal domain is the set $POWER(\mathbf{N})$, which is just the set of all subsets of the natural numbers **N** (i.e., non-negative integers). $POWER(\mathbf{N})$ is a complete lattice. However, it does not include natural embeddings of our scalar data objects. Furthermore, the embeddings of mathematical types into universal domains, as defined by papers in denotational semantics, are not suitable for our display theory. For example, a simple integer and a function from integers to integers are embedded to the same member of $POWER(\mathbf{N})$. A display function applied to the lattice $POWER(\mathbf{N})$, with these embeddings, would produce the same display for the integer and the function from integers to integers. Since the goal of visualization is to communicate information rather than to make it obscure, other embeddings of types into universal domains must be developed. Specifically, an extension of our display theory to recursively defined data types should include a universal domain with natural embeddings of our scalar data types, and should include representations of our tuple and array type constructors that will produce natural embeddings of constructed types.

5.1.3 Display of Recursively Defined Data Types

Since the goal of visualization is to communicate the information content of data to users, an extension of our theory must focus on the data lattice U. However, since a display function D is a lattice isomorphism of U onto a sub-lattice V, we should be able to say some things about the structure of V. If a subset $A \subseteq U$ is the solution of a recursive domain equation, then $D(A) \subseteq V$ is isomorphic to A and must itself be a solution of the recursive domain equation.

For example, if the set A is the solution of (2) for *Bintree*, then the set $D(A)$ must also solve this equation. The isomorphism provides a definition of the operators "+", "×" and "$(.)_\perp$" in $D(A)$ and thus also defines a relation between objects and their "subtree" objects in $D(A)$. The isomorphism does not tell us how to interpret these operators and relations in a graphical display, but it does tell us that such a logical structure exists. Given the complexity of this structure, it seems likely that display objects in $D(A)$ will be interpreted using some graphical equivalent of the pointers that we use to implement data objects in A.

Two graphical analogs of pointers come to mind immediately:

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

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

In order to extend our display theory to data types defined with recursive domain equations, we need to extend our display lattice V to include these graphical interpretations of pointers. The most interesting problem is to find a way to do this that produces a display lattice complex enough to be isomorphic to a universal domain as described in Sect. 5.1.2.

5.2 Abstract Data Types and Object Classes

Abstract data types and the object classes of object-oriented programming are ways of defining data types that hide the internal structures of data objects from the programs that use those data objects. Definitions of abstract data types and object classes include definitions of *member functions* for basic operations on data objects. Data objects are accessed by applying these member functions, rather than by selecting their primitive sub-objects. In fact, the hidden implementation of data objects may not include sub-objects at all, but may be purely functional. For example, an array data object may be implemented either by explicitly storing elements of the array, or by a function for computing the elements of the array as they are accessed.

5.2.1 Abstract Data Types

In an algebraic setting [17], abstract data types are specified by a signature $\Sigma = (T, F)$ and a set of logical conditions E. T is a set of types and F is a set of member functions among the types in T (that is, the types of the member functions, and the numbers and types of their arguments, are specified). E is a set of first order statements involving quantifiers, equality, the member functions of F, and variables with types in T. It is undecidable whether the statements in E are satisfiable (i.e., no algorithm exists which can tell, for given E, whether any set of data objects and functions satisfy E), so there are no compilers that produce implementations from T, F and E. However, abstract data types are used as the basis of programming methods. System designers use heuristic methods to derive conditions in E by analyzing system requirements, and use these conditions as a guide for implementing the functions in F [9].

Because of the generality of the abstract data type formalism, it can be used to express our lattice theory of display. To see this, define $T_{LAT} = \{U, V\}$, $F_{LAT} = \{inf_U: U\times U\rightarrow U, sup_U: U\times U\rightarrow U, inf_V: V\times V\rightarrow V, sup_V: V\times V\rightarrow V, D: U\rightarrow V\}$ and $E_{LAT} =$ {lattice axioms for U and V, expressiveness conditions on D}. That is, the data types of T_{LAT} are the data and display lattices U and V, the functions in F_{LAT} are the lattice operations on U and V plus the display function D, and the logical

conditions in E_{LAT} are the axioms defining U and V as lattices plus the expressiveness conditions. Expressiveness Conditions 1 and 2, as defined in Sect. 3.4, quantify over $MON(U \rightarrow \{\perp , 1\})$ and are thus second order statements whereas E_{LAT} is supposed to consist of first order statements. However, as shown by Prop. 1, Conditions 1 and 2 are equivalent to conditions that can be expressed as first order statements. There are obviously many different sets of lattice U and V satisfying the abstract data type definition in T_{LAT}, F_{LAT} and E_{LAT}.

In Sect. 5.2.3 we will discuss the issues involved with extending our lattice theory to display the data objects of abstract data types.

5.2.2 Object Classes

Like abstract data types, the object classes of object-oriented programming languages define access to data objects in terms of a set of member functions. However, rather than defining logical conditions that the member functions must satisfy, object classes define these functions explicitly as programs. An *object class* in C++ defines members as both data structures and functions, which are divided into private and public parts [3]. The private members are only accessible from member functions defined as part of the class (i.e., they are not accessible from outside the class definition).

In addition to hiding the implementation of object classes, object-oriented languages provide two mechanisms called polymorphism and inheritance that provide a novel style of programming compared to traditional procedural languages. *Polymorphism* means that the same member function name may be defined in the public parts of multiple object classes. Calls to member functions are bound to the appropriate function definitions at run time, determined by the classes of the objects passed as arguments to the functions. *Inheritance* allows an object class to be defined in terms of another. The new class "inherits" the members of the old class, except where those data members are explicitly redefined.

Object-oriented visualization systems define polymorphic display functions. These systems partition their data models into a number of object classes, and their polymorphic display functions may be called to display any data object in their data models. Thus the object oriented approach and our lattice approach both define display functions that can be applied to data objects of any type. However, where the object oriented visualization systems require constructive definitions of display functions as programs, our approach defines data and display lattices and defines display functions as any function satisfying the expressiveness conditions. Thus it is still interesting to investigate how our lattice theory can be extended to the object classes of an object-oriented language.

The private and public members of class definitions include data structures. These may be displayed using the techniques that we developed in Sect. 3, and the techniques for displaying recursively defined data types suggested in Sect. 5.1. However, this approach does not provide a systematic way to display data objects defined by classes, since class definitions include functions as well as data

structures. In the next section we will discuss issues involved in extending our lattice model of display to the member functions of object classes.

5.2.3 Lattice Models for Abstract Data Types and Object Classes

Because of the similarity between abstract data types and object classes, we will discuss them together, using the notation of abstract data types. We let T denote a set of types and let F denote a set of member functions. The types in T may be abstract data types or may be object classes, and the functions in F may be defined by a set E of logical conditions or by a set of programs. The important point is that the functions in F define relations among data objects, and that these relations take the place of a definition of data objects in terms of their internal structure. In fact, we could say that the relations defined by functions in F are a generalization of the relations between objects and their sub-objects that define the internal structures of data objects. In Sect. 3.1 we defined tuple objects in terms of their element sub-objects, and we defined array objects in terms of their domain and range sub-objects. These relations between objects and their sub-objects are special cases of relations between objects expressed by member functions.

Let A be the union of the sets of data objects of all types in T. We could give A the discrete order (i.e., no object is greater than any other object), but this would lead to a very boring theory of data display. In order to define a more interesting order relation on A, we can start with the data lattice that we defined in Sect. 3.2 (here we call it U_0). If T is a set of abstract data types, then we let the objects of U_0 serve as constants in the logical conditions of E. If T is a set of object classes, then we take the continuous and discrete scalar types of U_0 as the primitive values of an object oriented programming language.

If we also assume that the member function in F are monotone, then we can use these functions to derive order relations between objects in A from the order relations between objects in U_0. However, there is no guarantee that there is a order relation on A that is consistent with the assumption that the functions in F are monotone. For example, while it is easy to define monotone arithmetic operators on the scalar types of U_0, there is no reasonable way to define monotone logical and comparison operators on the scalar types of U_0 (we run into inconsistencies assuming either that *false* $\leq$ *true* or that *true* and *false* are not ordered). This suggests that a monotonicity assumption is a severe restriction on the member functions in F.

However, in order to define an interesting order condition on A we may assume that member functions are monotone. In this case, we need to verify that the monotone functions of F are consistent with an order relation on A, although this appears to be a difficult problem. If T is a set of object classes, and if member functions are implemented in a programming language that includes logical and comparison operators, then it is generally undecidable whether functions defined among the objects of U_0 are monotone. However, we may be able to design a restricted programming language for member functions that allows us to verify that monotone member functions are consistent with an order relation on A. If T is a set

of abstract data types, then the monotonicity requirement must be added to E as a set of logical conditions on the member functions in F (along with conditions that define order relations on the types in T). This may cause a set of satisfiable conditions to become unsatisfiable, and it is generally undecidable whether the addition of monotonicity conditions causes a set E of conditions to become unsatisfiable. Of course, this situation is no worse than without monotonicity assumption, since the question of whether a set of logical conditions is satisfiable is generally undecidable.

Given an ordered set A of data objects (the union of the sets of data objects of each type in T), we can use the Dedekind-MacNeille completion to embed A in a complete lattice U. In order to apply our display theory we would need to construct a display lattice V such that isomorphisms from U onto sub-lattices of V exist, and develop interpretations of display objects in V in terms of a physical display device. Since the display function $D:U \rightarrow V$ is an isomorphism between the set of data objects and a subset of the set of display objects, and since the relations between data objects expressed by the member functions in F (and subject to the logical conditions in E) are a generalization of the hierarchical relations between objects and sub-objects in the data model defined in Sect. 3.1, it is natural to seek an interpretation of display objects in terms of relations between display objects that generalizes the relation between display objects and graphical marks as described in Sect. 3.3. For example, we may represent data objects by icons in a display, and let users interactively explore the relations between those icons as defined by the functions in F. Finding a systematic way to interpret displays of abstract data types seems like a very open ended and interesting problem.

It is interesting to note that in the case of abstract data types, we can use the generality of the framework to add our display model to an existing set of abstract data types defined by T, F and E. Take T_{LAT}, F_{LAT} and E_{LAT} as defined in 5.2.1, and define

$$T' = T \cup T_{\mathrm{LAT}}$$
$$F' = F \cup F_{\mathrm{LAT}} \cup \{\text{embeddings of the types in } T \text{ into } U\}$$
$$E' = E \cup E_{\mathrm{LAT}} \cup \{\text{monotonicity conditions on the embeddings from } T \text{ into } U\}$$

Of course, there is no algorithm for deciding if the conditions in E' are satisfiable or for constructing the lattice U and V if they are. Furthermore, this tells us nothing about how display objects in V are interpreted in terms of a physical display device.

6. Conclusions

The design of the VIS-AD data model is tightly integrated with a computational model and a display model. The result is a data model whose data objects can be uniformly visualized using polymorphic display functions, and which has the flexibility to adapt to scientists' computations. Several kinds of metadata are integrated with this data model, providing a novel and useful programming

language semantics, and also providing the capability to display multiple data objects in common frames of reference.

The VIS-AD data model is based on lattices. These lattices may be applied to models of both data and displays. This provides an interesting context for analyzing visualization processes as functions from data lattices to display lattices. There are also interesting prospects for extending the lattice theory of visualization to more complex data models that involve recursively defined data types, abstract data types, and the object classes of object oriented programming languages.

References

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

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

[3] Gorlen, K. E., S. M. Orlow and P. S. Plexico, 1990; Data Abstraction and Object-Oriented Programming in C++. John Wiley & Sons.

[4] 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.

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

[6] Hibbard, W., C. Dyer and B. Paul, 1993; A lattice theory of data display. Submitted to IEEE Visualization '94.

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

[9] Mitchell, R., 1992; Abstract Data Types and Modula-2: a Worked Example of Design Using Data Abstraction. Prentice Hall.

[10] Moore, R. E., 1966; Interval Analysis. Prentice Hall.

[11] 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.

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

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

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

[15] Scott, D. S., 1982; Lectures on a mathematical theory of computation, in: M. Broy and G. Schmidt, eds., *Theoretical Foundations of Programming Methodology*, NATO Advanced Study Institutes Series (Reidel, Dordrecht, 1982) 145-292.

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

[17] Wirsig, M., 1990; Algebraic specification. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 675-788.