Chapter 5

Applying the Lattice Model to Recursive Data Type Definitions

In Section 3.1 we showed that a function $D : U \rightarrow V$ satisfying the expressiveness conditions must be a lattice isomorphism. In Section 3.4 we applied this result to specific lattice structures defined for scientific data and display models. However, this result can be applied to any complete lattices and it is natural to apply this result to other lattice structures for data and display models. The motive for new lattice structures must be new data models, since display models are themselves motivated by the need to visualize data. The data model defined in Section 3.2 includes tuples and arrays as ways of aggregating data, but does not include linked list structures defined in terms of pointers. In this chapter we describe several issues in extending our lattice theory to data types appropriate for handling objects with pointers.

5.1 Recursive Data Types Definitions

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

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

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

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

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

5.2 The Inverse Limit Construction

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

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

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

etc.

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

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

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

The inverse limit construction defines the set of data objects of a particular data type that solves a particular recursive domain equation. However, our approach in Section 3.2 was to define a large lattice that contained data objects of many different data types. It would be useful to continue this approach, by defining a lattice that includes all

data types that can be constructed from scalar types as tuples, arrays, and solutions of recursive domain equations. This is the subject of Section 5.3.

5.3 Universal Domains

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

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

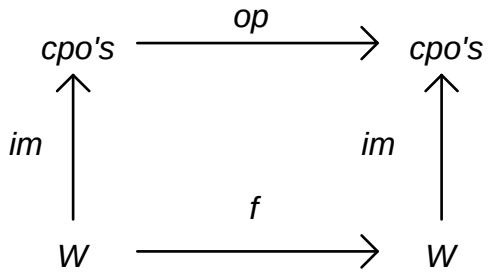

Figure 5.1. The type construction operator *op* is represented by function *f*.

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

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

5.4 Display of Recursively Defined Data Types

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

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

Two graphical analogs of pointers are commonly used in displays:

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

2. Hypertext links. Here the visible contents of a display screen represents one or more nodes in a data object, and an icon embedded in that display screen represents an interactive link to another node or set of nodes. That is, if the user selects the

icon (say by a mouse point and click), new display screen contents appear depicting the display object (and possibly other objects) referenced by the icon.

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

Chapter 6

Conclusions

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

6.1 Main Contributions and Limitations

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

1. Development of a system for scientific visualization that enables a wide variety of visual experiments with scientific computations. This system integrates visualization with a scientific programming language that can be used to express scientific computations. This programming language supports a wide variety of scientific data types and integrates common forms of scientific metadata into the computational and display semantics of data. Any data object defined in a program in this language can be visualized in a wide variety of ways during and after program execution. Displays are controlled by a set of simple mappings rather than program logic. These mappings are independent of data type and separate from a user's scientific programs, which is a clear distinction from previous visualization

systems that require scientists to embed calls to visualization functions in their programs. Furthermore, computation and visualization are highly interactive. In particular, the selection of data objects for display and the controls for how they are displayed are treated like any other interactive display control (e.g., interactive rotation). Previous visualization systems require a user to alter his program in order to make such changes. The generality, integration, interactivity and ease-of-use of this system all enhance the user's ability to perform visual experiments with their algorithms.

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

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

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

6.2 Future Directions

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

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

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

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

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

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

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