AUTOMATED DISPLAY OF GEOMETRIC DATA TYPES

by

William L. Hibbard
and
Charles R. Dyer

Computer Sciences Technical Report #1015

March 1991

Automated Display of Geometric Data Types

William L. Hibbard[1,2] and Charles R. Dyer[2]
[1]Space Science and Engineering Center
[2]Computer Science Department
University of Wisconsin-Madison
Madison, Wisconsin 53706

ABSTRACT

Systems for visualizing data and systems for visualizing algorithm behavior both face the problem of producing displays from user-definable geometric data structures. We present a technique for solving this problem based on a data type definition language and an algorithm for producing displays from any definable data type. The data types are constructed as scalars, tuples and arrays. We define a data type for a display window in terms of dimensions for space, animation, interactive selection and color. The user defines mappings from the dimensions of the application to the dimensions of the display, and the display algorithm transforms application data types to match the display data type, based on these mappings.

CR Categories: I.3.2 [Computer Graphics]: Picture/Image Generation - Display algorithms; I.3.6 [Computer Graphics]: Methodologies and Techniques - Languages.
Additional Keywords: Program visualization, image processing.

ACKNOWLEDGEMENTS

We wish to thank John Benson and Robert Krauss for their help. This work was funded by NASA/MSFC (NAG8-828) and the Gigabit Testbed Project managed by CNRI.

1. INTRODUCTION

Interactive visualization is being applied to increasingly complex information and tasks. These include visualizing data sets produced by observation and simulation, and visualizing ongoing processes such as communication systems, weather models or programs being debugged.

Systems like ConMan [3], AVS [9] and apE [2] allow their users to build data flow networks of processing modules into applications for analyzing and visualizing complex data. These systems manage their data in a variety of geometric data structures, such as images, multi-dimensional arrays, and lists of polygons and vectors. Data are passed between modules in these structures. These systems typically provide a library of modules implementing commonly used data transformations. Some of these produce graphical output, allowing the user to build visualization applications as networks of modules.

The Khoros system [8] provides a library of image processing modules that can be composed into data flow networks. The system can display images computed at any point in a module network to help algorithm designers understand the behavior of their algorithms. Thus, while the module networks of Khoros are similar to those of ConMan, AVS and apE, Khoros differs from those systems in the intended relation between the module network and the visualization. Khoros exploits visualization to support the creation and refinement of algorithms, whereas the other systems emphasize the module network as an application for producing analyses and visualizations as the end product.

The Powervision system [7] uses an object-oriented language to support interactive development of image processing algorithms. Algorithms are expressed in a procedural programming language rather than as data flow networks. The system defines several generic functions for accessing data objects, which must be included in the definition of every object class in the system. Powervision includes several predefined display types such as image, curve and region, which are defined in terms of generic access functions. Thus when the definitions of new object classes include the generic functions, their display is trivial.

The BALSA system [1] allows students to control the execution of programs and to see depictions of their changing data structures. Algorithms to be visualized with BALSA start as programs written in a procedural language such as Pascal or C. These are adapted to BALSA by designing useful displays of their data structures, and adding calls to BALSA library routines for generating these displays and for interactive execution control.

All of these systems are limited in their extensibility to new data structures. Applying BALSA to a new algorithm requires the design of displays particular to its data structures. The object-oriented language of Powervision reduces the amount of code required for the display of new object classes, but does not eliminate it. The designer of a new object class in Powervision must implement the required generic functions for data object access, and a particularly novel object class may require the definition of a new display type. The displays of ConMan, AVS, apE and Khoros are limited to data structures for which display modules have been previously implemented. There is a need for visualization that can be applied automatically to user-definable data structures.

We are designing the AVID (Algorithm Visualization and Interactive Development) system as an environment for analyzing and visualizing complex data and for developing algorithms for processing images and other scientific data. AVID will provide a library of processing modules, and tools for interactively writing and executing algorithms using these modules. It will also be capable of generating graphical depictions of data computed at any point in a module network. The principal problem in this effort is the design of extensible data structures and automatic display generation for these data structures. In this report we will define a data type definition language that supports complex geometric data structures, and describe an algorithm for automatically generating displays from these data types.

2. RELATED WORK

There is currently a limited amount of work on automated display of complex data. Three examples are the University of Washington Illustrating Compiler (UWPI) [4], the APT (A Presentation Tool) system for automated design of graphical presentations of relational information [6], and the COOL (COnstraint-based Object Layout) system for layout of objects and relations [5].

The UWPI system is a Pascal compiler that analyzes the data structures and procedural logic of a program to detect higher level structures such as stacks and queues. The execution state of the program is then illustrated at run time using standard graphical depictions of these higher level structures. The basic display elements of UWPI are numbers, which are simply printed as digits; arrays, which are displayed as histograms; and directed graphs, which are depicted with nodes and links. The system attempts to combine multiple data structures into a coherent structure showing the relations between variables. The UWPI system provides an automated approach to some of the display design problems posed by the BALSA system, although UWPI provides useful displays for only a limited set of data structures. Furthermore, as the creators of UWPI observe, algorithms using high level data structures may be written in languages where these structures may be coded explicitly as abstract data types, rather than being deduced implicitly through compiler analysis.

The APT system generates static 2-D graphical designs for data from a relational data base, where the designs are represented as sentences in a graphics language. The system defines criteria for expressiveness and effectiveness of graphical designs as predicates involving data relations and graphics languages. The APT system uses backtracking search to find a graphical language representation for a set of relations that satisfies the expressiveness criteria and optimizes the effectiveness criteria. The formality of the APT approach may make it applicable to a wide variety of graphical design problems. The creators of APT note that it could be extended to dynamic and 3-D displays, and to finer grained graphics primitives in its graphics languages. However, these extensions will pose efficiency problems for the backtracking search.

The COOL system, with its associated TRIP (TRanslation Into Pictures) system, automatically translates textual descriptions of objects and their relations into two-dimensional images composed of simple shapes describing the same information. The text descriptions are first translated using YACC to a PROLOG representation. Users specify the mapping from objects and relations to geometric shapes as a PROLOG program, and this translates the PROLOG representation into a set of geometric shapes with constraints. The COOL kernel solves the constraint system and generates a picture layout, which is rendered by TRIP. This system has been applied to generate images from kinship diagrams, list structures, program structures and entity-relationship diagrams.

Most physical science data are defined in terms of space, time, and continuous numerical variables. Such data require a geometrical data model. The UWPI, APT and COOL systems described above are concerned with the geometry of the display, but do not provide a geometrical model of data. The UWPI system uses primarily the complex linked data structures of systems programming. The APT system uses a relational data model and the COOL system uses an object and relation model, both appropriate to business and social science. None of these data models efficiently support the geometric nature of most physical science data.

3. DATA TYPES

Our goal is to provide a geometric data model based on real and integer values, which may be built into more complex structures using tuples and arrays. In fact, we allow the users to define data types appropriate to their applications as arbitrary hierarchies of tuples and arrays. In this section we define a data type definition language and how it describes data objects, and we give some examples of type definitions.

3.1 Data Type Definition Language

Data types are constructed from **atoms**, which correspond to the basic dimensions and quantities in an application. An atom declaration consists of an atom class name, which may be REAL, REAL2D, REAL3D, INTEGER, LIST or STRING, followed by a list of atom names. REAL2D and REAL3D atoms take pairs and triples of real numbers as values. LIST atoms take integer values that do not have physical significance in the application. Rather, LIST atoms serve as the domains of variable length arrays used to implement list data structures. STRING atoms take text string values.

Types are constructed as scalars, tuples and arrays over the atoms. The syntax of a type declaration is:

```
type_declaration := TYPE type_name type
type := atom_name                  /* scalar */
type := ( atom_name → type )       /* array: domain atom → range type */
type := ( element_list )           /* tuple: list of element types */
element_list := type , element_list
element_list := type
type := type_name                  /* type reference */
```

The recursive type constructions for array and tuple allow very complex types to be defined in this simple syntax.

3.2 Data Objects

Every **data object** is declared with a data type and holds a value of that type. Every atom occurrence in a data type has an associated **state set**, which is the finite set of possible values for that atom. The state set always includes the distinguished value MISSING, used to indicate the lack of information. A data object of a **scalar** type holds a single value from the state set of its atom. A data object of a **tuple** type holds one value for each of its element types. A data object of an **array** type associates one value of its range type with each value in the state set of its domain atom. Intuitively, the domain atom of an array is an index into a collection of data objects of the range type. An entire array or tuple data object may have the single value MISSING.

In most programming languages, the set of possible values of an atomic data object is fixed by the type definition. Here, however, the state set specifications for atom occurrences are part of the data object. In a full implementation of the data type definition language, occurrences of atom_name in type declarations will be elaborated to specify how the state sets of atoms vary over array domains and between different occurrences of an atom. However, this is not necessary for a description of our display algorithm. Here we will assume that all occurrences of an atom have the same state set, except for LIST atoms. The state set of a LIST atom varies freely between occurrences of the atom in a data object, so that there is no relation between the lengths of different lists.

The value of a **type reference** in a data object is a pointer to another data object of the referenced type. Circular chains of type references are not allowed, so a type declaration may not reference itself, and may not reach itself by following a chain of type references. Thus type references may be resolved at run time by replacing pointers by the referenced data objects. In the context of our display algorithm, this is equivalent to replacing type references in a type declaration by the text of the referenced declarations. We will assume that such text replacements are made and not explicitly consider type references in the display algorithm.

3.3 Examples of Data Types

Examples of atom and type declarations are:

REAL2D location

REAL brightness

INTEGER region, frequency

LIST line_list

TYPE image (location→brightness)

TYPE partition (location→region)

TYPE boundary (line_list→(location,location))

TYPE boundaries (region→boundary)

TYPE histograms (region→(brightness→frequency))

The *image* type associates a brightness with each of a finite set of locations in the real plane. The *partition* type associates a region with each location. The *boundary* type is a list of location pairs (the end points of line segments). The *boundaries* type associates a *boundary* with each region. The *histograms* type associates a frequency with each brightness, for each region.

Because the state sets of atoms are specified as part of the data objects, the *image* type is indexed by a REAL location which may, for example, be earth latitude and longitude. The locations of pixels become part of the *image* data object. Thus this single *image* data type may be used to hold data from a variety of different sensing instruments each with a different map projection. Similarly, the brightnesses of pixels are REAL values that may be objective physical quantities, with the sampling of this quantity specified as part of the *image* data object. The usefulness of scientific data depends on attached auxiliary data for location and sampling. By specifying the state sets of atoms as part of the data objects, AVID builds this function into its data management system.

4. DISPLAY OF DATA TYPES

4.1 The Display Data Type Template

Our display technique assumes that the user sees a three-dimensional volume projected onto the display window, that the contents of that volume may be animated, that the contents may be altered by widget selections, and that the contents are defined by a variety of rendering parameters attached to voxels. The type definition language includes intrinsic definitions of atoms for the basic dimensions and quantities of the display. The REAL3D atom P3 defines a three-dimensional space that is projected onto the display window. The state set of P3 is the orthogonal product of the state sets of the REAL atoms PX, PY and PZ. PXY, PXZ and PYZ are their REAL2D products. The REAL atom T defines a sequence of times that may be animated in the display. The atoms Si, i=1,...,n, may be REAL, REAL2D, REAL3D, INTEGER or STRING, and define widget selectors of display contents. The atoms Ri, i=1,...,m, may be REAL, REAL2D or REAL3D, and define color and other rendering parameters attached to voxels. The special atom TWO has states MEMBER and MISSING and is used in an array range to indicate a subset of the array domain state set.

The **display type template** is defined as:

(S1→(S2→...(Sn→(T→(P3→(R1,...,Rm,TWO))))...))

A data type matches this template if the template can be made identical to the type by removing any of the display domain atoms Si, T or P3, by replacing P3 by any of its factors PX, PY, PZ, PXY, PXZ or PYZ, or by removing any of the display range atoms Ri or TWO.

Any type X that matches the display type template can be displayed in a straightforward way. Appropriate widgets can be created allowing the user to select a value from the state set of each atom Si in X. If the atom T occurs in X, values from its state set may be selected or sequenced for animation under user control. The display is defined as a sequence of frames, with each frame generated from a data object indexed by the selected values for the atoms Si and T, and matching the template: (P3→(R1,...,Rm,TWO)). This data object is a zero-, one-, two- or three-dimensional array of voxels that are projected onto the screen. The voxels are rendered according to the values of the atoms Ri and TWO. Each atom Ri present in X is linked to color or some other parameter for voxel rendering, to a scalar quantity to be rendered with iso-level contour surfaces or lines, or to a two- or three-dimensional quantity to be plotted using vectors. If TWO is present in X, the MEMBER value is rendered as monochrome. The MISSING value of Ri and TWO is rendered as invisible.

4.2 Mapping Application Atoms to Display Atoms

Designing a display for a data structure generally involves mapping its dimensions and quantities to the dimensions of the display screen, to animation, to widget selections and to colors. Our technique transforms data types into the form of the display type template based on user-defined **mappings** from the atoms declared in the application to the display atoms P3, PX, PY, PZ, PXY, PXZ, PYZ, T, Si and Ri. Each application atom is either mapped to one display atom or is unmapped, and more than one application atom may be mapped to the same display atom. Given a type to be displayed, its application atoms are replaced by display atoms based on the user-defined mappings. Then the new type is transformed to match the display type template through a sequence of basic transforms that eliminate unmapped application atoms and rearrange display atoms into the template form.

Our technique requires that numerical atoms (REAL, REAL2D, REAL3D, INTEGER and LIST) may only be mapped to numerical atoms of the same dimension, with no guarantee that state set values are mapped to state set values. STRING atoms may only be mapped to STRING atoms with no change to text values in data objects. No atom may be mapped to TWO.

Replacing application atoms by display atoms requires adjustment of data objects because of the mappings of numerical atoms. The value of a scalar data object, whose atom is mapped to a numerical display atom D, is replaced by the value in the state set of D nearest the mapped value. The value of a tuple data object is adjusted by adjusting the values of its element data objects. The value of an array data object, assuming that the atom C is its array domain, is adjusted according to:

```
if C is mapped to a numerical display atom D by function F
  for each value d in the state set of D
    find the value c in the state set of C such that F(c) is
    nearest to d;
    if F(c)-d is sufficiently small
      let Z be the data object associated with c in the array;
      adjust the value of Z;
      associate Z with d;
    else
      associate MISSING with d;
    end if
  end for
else
  for each value c in C
    let Z be the data object associated with c in the array;
    adjust the value of Z;
  end for
end if
```

When C is mapped to a numerical display atom, this procedure resamples the array to the nearest value mapped from C. Alternatively, it would be straightforward to interpolate from a set of neighboring values mapped from C.

4.3 Transforming Data Types to the Display Type Template

The display algorithm applies a sequence of basic transforms to alter a data type and its data objects to the form of the display type template. The algorithm works recursively on the array and tuple type constructions. Tuple and array types are transformed by first transforming their element and range subtypes, and then applying basic transforms based on the assumption that their element and range subtypes match the display type template. Here we present the general outline of the display algorithm and its basic transforms. See the appendices for details.

We define each **basic transform** using the format:

TYPE_TEMPLATE **-transform_name->** TYPE_TEMPLATE

We let B, C and D represent atoms and X, Y and Z represent types.

A scalar data type is defined by a single atom, which must either be unmapped, mapped to one of the display domain atoms Si, T or a factor of P3, or mapped to one of the display range atoms Ri or TWO. An unmapped atom C is transformed by

C **-null->** NULL

where NULL is a special type that matches the trivial display type template with all atoms omitted. An atom mapped to a display domain atom B is transformed to match the display type template by the transform

B **-set_scalar->** (B->TWO)

where (B->TWO) represents the subset of B containing the single data value from the scalar B. An atom mapped to a display range atom already matches the display type template and is not transformed.

An array data type (C->X) has a domain atom C and a range type X. A recursive call is made to the algorithm to transform X to match the display type template. As with a scalar type, the domain atom C must either be unmapped, mapped to a display domain atom, or mapped to a display range atom. In all three cases, the transform

(C->(B->Y)) **-permute->** (B->(C->Y))

may be applied to move the domain atom C into the nested arrays of the range type X. This is similar to permuting array indices in Fortran. If C is an unmapped atom, *permute* is applied to bring it inside the nested arrays of X until one of the following transforms may be applied:

(C->(D->TWO)) **-union->** (D->TWO)

(C->Ri) **-composite_scalar->** Ri

(C->(R1,...,Rn)) **-composite_tuple->** (R1,...,Rn)

Intuitively, these are projection operations, collapsing a dimension by summing over it. If C is a display domain atom, *permute* may be applied to bring it to its correct place among the array domains of X, or, if an identical array domain already exists in X, then the transform

(C->X) **-substitute->** (Si->X)

may be applied, replacing C by a newly created selection atom Si. Non-overlapping factors of P3 are replaced by their product by the transform

(Pi→(Pj→X)) **-merge→** (Pij→X)

If C is a display range atom, *permute* may be applied until the transform

(C→TWO) **-composite_set→** C

can be applied. If this is impossible, the *substitute* transform may be applied to replace C by a selection atom. Note that the *composite_set* transform sets the C data object to the value MISSING if the subset (C→TWO) is empty.

A tuple data type (X1,...,Xn) is defined by a list of element data types Xi. Except for two special forms of tuples, recursive calls are made to the algorithm to transform each element type Xi to match the display type template. The **position** and **embed** transforms are then applied to the tuple. The *embed* transform forces all the element types Xi to have the same sequence of nested array domains. The *embed* transform forms the set S of array domains occurring in all the tuple elements, and then adds domains to each element Xi until they have all the domains in the set S. When the *embed* transform adds a domain, for example replacing Xi by (B→Xi), the (B→Xi) array data object associates the value of the Xi data object with every domain value of B, unless B is a spatial domain (factor of P3). In that case, the value of the Xi data object is only associated with one value of B, and MISSING is associated with other values of B. Thus the *embed* transform extends data objects to constants over animation and selection atoms, but avoids pathologies like extruding 2-D graphs into cylindrical objects in space. The *position* transform uses values of scalar tuple elements that are display domain atoms to locate other tuple element data objects within those display domains. The *position* transform embeds tuple elements in arrays whose domains are display domain atoms that occur as other elements of the tuple. When the *position* transform replaces Xi by (B→Xi), the value of the Xi data object is associated with the scalar value of B occurring as an element of the tuple, and MISSING is associated with other values of B.

After the *position* and *embed* transforms are applied, the transform

((B→Y1),...,(B→Yn)) **-distribute→** (B→(Y1,...,Yn))

may be applied successively to the nested arrays of the tuple elements Xi, to unify their array domains. This results in the type (B1→...(Bm→(Z1,...,Zn))...), where each Zi is a tuple of display range atoms. The transform

(Z1,...,Zn) **-composite→** Z

combines the Zi into a single tuple Z of display range atoms.

The first special tuple form has N identical scalar elements, all spatial atoms (P3 or one of its factors). This is interpreted as a convex line segment, polygon or polyhedron. The transform

(D,...,D) **-scan→** (D→TWO)

produces the subset of D values lying in or near the convex closure of the scalar tuple elements. Intuitively, this corresponds to the scan conversion operation.

The second special tuple form has only scalar elements, at least one of which is a display domain atom, and is transformed by

(C1,...,Cn) **-set_tuple->** (C1->...(Cn->TWO)...)

The nested arrays represent the subset of the product of the state sets of the Ci's that contains the single tuple element data object.

This display algorithm is described in more detail in the appendices. It can deterministically transform any data type to match the display type template. This is shown by induction, following the form of the recursive algorithm given in Appendix B. Scalar types are transformed to match the template, and an array type can be transformed to match the template if its range type can be. The *-scan->* tuple case is transformed to match the template. The *-set_tuple->* tuple case can be transformed to match the template if the derived nested array type can be. The transform of each nested array depends on its range, which is either another array or the scalar TWO, so by induction the derived nested array type can be transformed to match the template. In the general tuple case the algorithm adds containing arrays to each element by the *-position->* and *-embed->* transforms and then invokes itself recursively on the altered elements. However, the successful transform of the containing arrays depends, by induction, on the successful transform of the original elements. Thus the general case tuple can be transformed to match the template if its elements can be. Since scalars can be transformed to match the template, and arrays and tuples can be if their range and element types can be, all types can be transformed to match the display type template.

The display algorithm is based on a few intuitive ideas of how scalars, arrays and tuples may be displayed. We have elaborated these ideas into a recursive procedure which transforms any hierarchical structure of scalars, arrays and tuples into displayable form. Several of the basic transforms reflect a non-trivial decision about data interpretation or display philosophy, as discussed below.

A) *Substitute* - this transform changes the mapping of an application atom occurrence to a selection atom to avoid identical nested array domains that have no intuitive depiction.

B) *Union*, *composite*, *composite_set*, *composite_scalar* and *composite_tuple* - these transforms merge data that occur at the same display location. The logic of the composite transforms may be implemented as user controllable. These transforms could be eliminated from the algorithm and replaced by *substitute* (a variant of *substitute* would be required for *composite*).

C) *Scan* - this transform assumes an interpretation of a tuple as a convex set. It could be eliminated from the algorithm and its special case tuples would be handled as general case tuples.

D) *Position* - this transform assumes an interpretation of some tuple elements as locations for other tuple elements. It could be eliminated from the algorithm.
E) *Embed* - this is an intuitive rearrangement, except for the interpretation of embedding in spatial domains as restricted to a single domain value rather than constant over the domain.

4.4 Examples of Type Transforms

The display algorithm may be illustrated using the example data types declared above for *image, partition, boundaries* and *histograms*. Assume that the user defines mappings from application atoms to display atoms as follows:

location to PXY
brightness to PZ
region to R1
frequency to PX.

The *image* type is declared as
TYPE image (location→brightness)
The first stage of the type transformation is to replace application atoms by their corresponding display atoms, so that the *image* type becomes
→ (PXY→PZ)
A data object of this type is an array providing a Z coordinate for each pair of X and Y coordinates. Since it is an array, the transform algorithm will first transform its range type PZ. PZ is a display domain scalar so the algorithm next applies
PZ *-set_scalar→* (PZ→TWO), and substituting this into the type expression above gives
→ (PXY→(PZ→TWO))
This consists of two nested arrays, whose domains PXY and PZ are non-overlapping factors of P3, so the algorithm finally applies
-merge→ (P3→TWO)
This matches the display type template so the overall transform of the *image* type is
(location→brightness) → (P3→TWO)
A data object of the transformed type represents a set of voxels in P3 lying on a surface showing brightness in the Z direction as a function of location over X and Y.

The *location* type is declared as
TYPE partition (location→region)

The first stage of the type transformation is to replace application atoms by their corresponding display atoms, so that the *location* type becomes

→ (PXY→R1)

Since this is an array, the transform algorithm will first transform its range type. However, R1 is a display range scalar and does not need to be transformed. Then the transform algorithm recognizes that the array (PXY→R1) also does not need to be transformed - it already matches the display type template. A data object of this type indicates region by color as a function of location over X and Y.

The *boundaries* type is declared as

TYPE boundaries (region→(line_list→(location,location)))

The first stage of the type transformation is to replace application atoms by their corresponding display atoms, so that the *boundaries* type becomes

→ (R1→(line_list→(PXY,PXY)))

The transform algorithm will be recursively applied to the range of each of these two nested arrays. The inner range is a tuple of identical factors of P3, so the algorithm applies (PXY,PXY) *-scan→* (PXY→TWO), and substituting this into the type expression above gives

→ (R1→(line_list→(PXY→TWO)))

Now the inner array has domain atom *line_list*, which is unmapped, so the transform algorithm applies (line_list→(PXY→TWO)) *-union→* (PXY→TWO), and substituting this into the type expression above gives

→ (R1→(PXY→TWO))

The domain R1 of this array is a display range atom, so the algorithm applies

-permute→ (PXY→(R1→TWO))

Then the transform algorithm applies (R1->TWO) *-composite_set→* R1, and substituting this into the type expression above gives

→ (PXY→R1)

This matches the display type template so the overall transform of the *boundaries* type is

(region→(line_list→(location,location))) → (PXY→R1)

A data object of the transformed type shows the region boundaries as colored polylines over location X and Y. The composite color is used at intersecting boundary points, and the value MISSING is used at non-boundary points. Figure 1 illustrates the transform of *boundaries*.

The *histograms* type is declared as

TYPE histograms (region→(brightness→frequency))

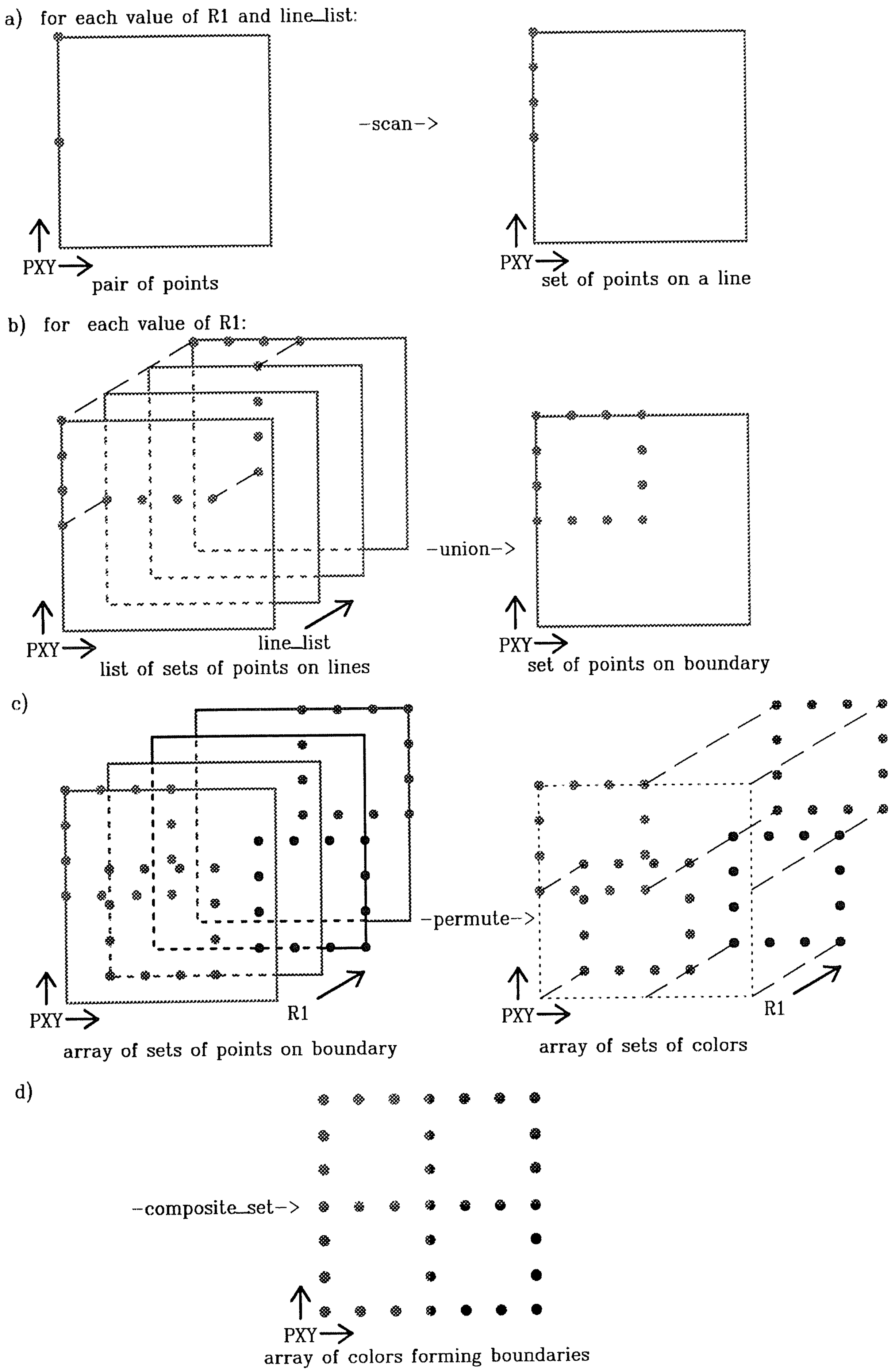
a) for each value of R1 and line_list:
-scan->
PXY
pair of points
PXY
set of points on a line
b) for each value of R1:
line_list
PXY
list of sets of points on lines
-union->
PXY
set of points on boundary
c)
PXY
R1
array of sets of points on boundary
-permute->
PXY
R1
array of sets of colors
d)
-composite_set->
PXY
array of colors forming boundaries

FIGURE 1. Transforming the *boundaries* type.

The first stage of the type transformation is to replace application atoms by their corresponding display atoms, so that the *histograms* type becomes

$\rightarrow$ (R1$\rightarrow$(PZ$\rightarrow$PX))

The transform algorithm will be recursively applied to the range of each of these two nested arrays. The inner range PX is a display domain scalar, so the transform algorithm applies PX *-set_scalar*$\rightarrow$ (PX$\rightarrow$TWO), and substituting this into the type expression above gives

$\rightarrow$ (R1$\rightarrow$(PZ$\rightarrow$(PX$\rightarrow$TWO)))

The transform algorithm is now applied to the inner array (PZ$\rightarrow$(PX$\rightarrow$TWO)). It consists of two nested arrays, whose domains PZ and PX are non-overlapping factors of P3, so the algorithm applies

(PZ$\rightarrow$(PX$\rightarrow$TWO)) *-merge*$\rightarrow$ (PXZ$\rightarrow$TWO), and substituting this into the type expression above gives

$\rightarrow$ (R1$\rightarrow$(PXZ$\rightarrow$TWO))

The domain R1 of this array is a display range atom, so the algorithm next applies

-permute$\rightarrow$ (PXZ$\rightarrow$(R1$\rightarrow$TWO))

Then the transform algorithm applies (R1$\rightarrow$TWO) *-composite_set*$\rightarrow$ R1, and substituting this into the type expression above gives

$\rightarrow$ (PXZ$\rightarrow$R1)

This now matches the display type template so the overall transform of the *histograms* type is

(region$\rightarrow$(brightness$\rightarrow$frequency)) $\rightarrow$ (PXZ$\rightarrow$R1)

A data object of the transformed type shows histograms as colored graphs, with frequency as X versus brightness as Z. The composite color is used at intersecting histogram points, and the value MISSING is used at non-histogram points.

4.5 Efficiency of Type Transforms

The display algorithm is costly in terms of both computing time and memory space. A data object is transformed to the display type template by a sequence of basic transforms, some requiring a pass through the entire data object. Transforms may also increase the size of a data object, particularly those that create new arrays.

An analysis of the computing time of the display algorithm is quite complex, depending on the storage implementation of data objects. Data types and objects are organized as tree structures, with array and tuple constructions at the non-leaf nodes, and atoms at the leaf nodes. The display algorithm rearranges a given tree structure by applying a sequence of transforms, which each act locally to rearrange a node and its children. Assuming an implementation using pointers from nodes to their children, most transforms do not need to process the contents of the data objects below the level of the nodes they rearrange. Furthermore, almost all of the storage space for a data object will be devoted to storing the values of the leaf nodes. Thus applications of transforms at higher levels in the tree will need much smaller amounts of computing time than those at lower levels.

One way to estimate the computing time of the display algorithm is to count the number of times the basic transforms access the contents of all the leaf nodes. Define D(X) as the maximum depth of nesting of the type tree X, and define L(X) as the number of times all the leaf nodes are accessed in transforming type X. Then $L(X) \leq 4*D(X)$; this can be proved by induction, following the form of the recursive algorithm given in Appendix B, as follows:

```
Case: scalar $X = C$
  $D(X) = 1$
  $L(X) = 0$ or $1$
  So in the scalar case, $L(X) \leq 4*D(X)$.
Case: array $X = (C \rightarrow Y)$
  $D(X) = D(Y)+1$
  Case: transform(Y)=NULL
    $L(X) = L(Y)+1$  /* the null_array transform will access all the leaf nodes to free their
    storage */
  Case: C = unmapped
    $L(X) = L(Y)+2$  /* the union, composite_scalar and composite_tuple transforms will each
    access all the leaf nodes, and the lowest level permute transform may also access all the leaf
    nodes */
  Case: C = display range
    Case: composite_set  /* if Y' = TWO */
      $L(X) = L(Y) + 1$  /* the composite_set transform will access all the leaf nodes */
    Case: substitute  /* if $X = (Ri \rightarrow Y)$ */
      $L(X) = L(Y) + (0$ or $1)$  /* the lowest level permute transform may access all the leaf
      nodes */
  Case: C = display domain case
    $L(X) = L(Y) + (0$ or $1)$  /* the lowest level permute transform or the merge transform may
    access all the leaf nodes */
  In the array case, $L(X) \leq L(Y)+2$ and by induction $L(Y) \leq 4*D(Y)$ so $L(X) \leq 4*D(Y)+2 =$
  $4*(D(X)-1)+2 = 4*D(X)-2 < 4*D(X)$.
Case: tuple $X = (Y1,...,Yn)$
  $D(X) = MAX \{D(Yi) | i=1,...,n\} + 1$
  Case: scan
    $L(X) = 1$  /* the scan transform will access all the leaf nodes */
```

Case: *set_tuple*

$L(X) \leq 2$ /* the *set_tuple* transform will generate one leaf node; because it sorts the Yi into an order to avoid any *permute* transforms, the recursive call to transform(X) will access the leaf node at most once with a *composite_set* transform */

Case: general tuple

$L(X) \leq MAX \{L(Yi) | i=1,...,n\} + (0, 1, 2, 3 \text{ or } 4)$ /* because the leaf nodes of differnet tuple elements do not overlap, we use the maximum rather than the sum of the L(Yi); the lowest level *permute* or *merge* transform may access each leaf node once, and these are applied after both the *position* and the *embed* transforms; the lowest level *distribute* transform may access each leaf node, and the *composite* transform will access each leaf node */

In the tuple case, $L(X) \leq MAX \{L(Yi) | i=1,...,n\} + 4$ and by induction $L(Yi) \leq 4*D(Yi)$ so $L(X) \leq 4*MAX \{D(Yi) | i=1,...,n\} + 4 = 4*(D(X)-1)+4 = 4*D(X)$.

This analysis provides a bound on how many times the display algorithm processes the leaf node data in a data object. However, the size of a data object may vary considerably as a data object is transformed, and this variation is difficult to estimate. As we discuss below, there are ways of implementing data objects that limit their growth as they are transformed.

The *scan*, *set_scalar*, *set_tuple* and *embed* transforms create new arrays and can therefore increase the sizes of data objects. However, these transforms do not add any new information. By providing alternate implementations for special-case data objects, the system can reduce redundant information in those data objects.

For example, the *embed* transform creates new arrays as seen below:

(X1,(B→X2)) *-embed→* ((B→X1),(B→X2))

If B is not a spatial atom (i.e., not a factor of P3), the X1 range value is the same for all B domain values. We can avoid an increase in the size of the corresponding data object by providing a special-case array, which stores the range value just once and notes that it is constant over the entire domain state set.

The *set_scalar* transform replaces a single value by an array representing a subset, as seen in the following example:

B *-set_scalar→* (B→TWO)

The array (B→TWO) represents a subset of B containing a single value. We can avoid an increase in the size of the data object by providing a special-case implementation of such arrays, which stores a list of those domain values associated with MEMBER, with the assumption that all other domain values are associated with MISSING.

Arrays derived from subset arrays by the *composite_set* transform also form a special case. For example, the transform:

$(P3 \rightarrow (R1 \rightarrow TWO)) \rightarrow (P3 \rightarrow R1)$
results from a substitution of $(R1 \rightarrow TWO)$ *-composite_set→* R1, and the array $(P3 \rightarrow R1)$ may associate most P3 values with MISSING. We can minimize the size of the corresponding data object by providing a special-case array, which stores a list of (domain value, range value) pairs for those range values which are not MISSING.

Another approach to limiting data object size is to recognize that the data objects are being transformed in order to be displayed, and that each frame of an animated or interactive display will depend on only part of a data object. In cases where computing power is more plentiful than data storage, it may be better to apply the transform algorithm once per frame, but to only transform those parts of a data object that have an effect on the frame contents. A modified version of the transform algorithm may be applied with special treatment for the animation atom T and the selection atoms Si. The modified algorithm would keep track of the currently selected values for the T and Si atoms. Where these atoms occur as array domains, the range values associated with non-selected domain values would be forced to the MISSING value, reducing the storage space needed for those arrays.

One approach to reducing the computing time of the display algorithm is to limit the number of transforms applied. This may be done by
A) Implementing a set of procedures that transform complex data objects to match the display type template in a single pass through the data object, each associated with a template for the types that it will transform. The display algorithm would compare a type against the set of templates, and apply the transform procedure rather than its usual logic if a match was found. Because the algorithm is applied recursively, this would benefit any type with subtypes that match the templates.
B) Simultaneously transforming non-overlapping subtypes of a type. For example, the element data objects of a tuple could be transformed in parallel.
C) Combining sequences of *permute* transforms into a single complex permutation.

Increasing processor performance will make it feasible to apply our display algorithm interactively to data structures of increasing size, and many of the basic transforms can be implemented naturally in parallel, voxel-based architectures.

5. CONCLUDING REMARKS

We have offered a solution to the problem of automatic display of geometric data structures. Our approach is to define a data type definition language that is used to define all the data structures of an application. Our display algorithm then transforms any data type to match a type template for a display window. The transformed data type may then be interactively displayed.

This approach may be adapted to a number of existing visualization systems, to transform data types into a type template for their existing displays.

Because our technique displays data types based on a set of atom mappings, different data types can be overlaid in a common frame of reference. Thus the user may track the relation between different processing stages in an algorithm, or may compare different structures within a data set. The display of multiple data types in a single frame of reference implies a compromise between the most natural frames of reference for different data types. However, an implementation may allow multiple display windows, each with its own set of atom mappings, and thus each most natural to different aspects of the data.

AVID data types are constructed as trees of tuple and array nodes. The AVID display algorithm works by applying a sequence of basic transformations to the nodes of the type tree, starting at the leaves and working up the tree in a deterministic way. This simplicity should help make its computing time and results predictable and intuitive.

AVID places severe restrictions on the use of type references, so that complex linked structures involving circular references cannot be built in AVID. Thus AVID is inappropriate for visualizing the structures of operating systems and other non-geometric domains.

REFERENCES

[1] Brown, M., and R. Sedgewick, 1984; A system for algorithm animation; Computer Graphics 18(3), 177-186.

[2] Dyer, D., 1990; A dataflow toolkit for visualization; Computer Graphics and Applications, 10(4), 60-69.

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

[4] Henry, R., K. Whaley, and B. Forstall, 1990; The University of Washington illustrating compiler; SIGPLAN Notices 25(6), 223-233.

[5] Kamada, T., and S. Kawai, 1990; A general framework for visualizing abstract objects and relations; ACM Transactions on Graphics, 10(1), 1-39.

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

[7] McConnell, C. and D. Lawton, 1988; IU software environments; Proc. IUW, 666-677.

[8] Rasure, J., D. Argiro, T. Sauer, and C. Williams, 1991; A visual language and software development environment for image processing; International Journal of Imaging Systems and Technology, accepted for publication.

[9] Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, 1989; The application visualization system: a computational environment for scientific visualization; Computer Graphics and Applications, 9(4), 30-42.

Appendix A. Basic Transforms of Data Types

This appendix contains a list of the basic transforms of types, in the format:
TYPE_TEMPLATE **-transform_name→** TYPE_TEMPLATE
followed by any necessary conditions not specified in the input TYPE_TEMPLATE, and a description of the corresponding transform of data objects. In the type templates we let L represent any unmapped application atom, P represent P3 or any of its factors, B represent P, T or Si, D represent P, T, Si, Ri or TWO, Qi represent Ri or TWO, C represent any atom, and X and Y represent any type. In the psuedocode sections, statements of the form "**type** X = ..." elaborate the type structure of X and assign symbols to its atoms and subtypes.

L **-unmapped→** NULL

the scalar data object is replaced by the NULL object

(D→NULL) **-null_array→** NULL

the array data object is replaced by the NULL object

(Y1,...,Yi,NULL,Yi+1,...,Yn) **-null_tuple→** (Y1,...Yi,Yi+1,...,Yn)

the NULL object is deleted and the tuple is shortened

(Y) **-unary→** Y

the tuple data object is replaced by its single element data object

(L→(B→Y)) **-permute→** (B→(L→Y))

the range values of the array data object are permuted, as one would reorder the indices of a Fortran array

(Ri→(B→Y)) **-permute→** (B→(Ri→Y))

(B1→(B2→Y)) **-permute→** (B2→(B1→Y))

where (B1 = P and B2 = T or Si) or (B1 = T and B2 = Si) or (B1 = Si and B2 = Sj and i>j)

(L→(D→TWO)) **-union→** (D→TWO)

the array (D→TWO) represents a subset of the state set of D; the new data object is the union of the subsets associated with each L value

(Ri→TWO) **-composite_set→** Ri

the transform rule for data objects can be modified by the user; the default is that the new scalar data object has the state set value nearest the average of the Ri values in the subset represented by (Ri→TWO); if the subset is empty then the new scalar data object has the value MISSING

(L→Qi) **-composite_scalar→** Qi

the transform rule for data objects can be modified by the user; the default is that the new scalar data object has the state set value nearest the average (inclusive-or for Qi = TWO) of the range values of (L→Qi)

(L→(Q1,...,Qn)) **-composite_tuple→** (Q1,...,Qn)

the transform rule for data objects can be modified by the user; the default is that the Qi element for the new tuple data object has the state set value nearest the average (inclusive-or for Qi = TWO) of the Qi element values from the array range

(Y1,...,Yn) **-composite→** (Q1,...,Qm)

where each Yi has the form (Q1',...,Qk') or Q1'; set Q1,...,Qm to the union of the Qj' in all the Yi; the transform rule for data objects can be modified by the user; the default is that the Qj element for the new tuple data object has the state set value which is nearest the average (inclusive-or for Qj = TWO) of the values of all occurrences of Qj among the Yi

(P,...,P) **-scan→** (P→TWO)

where the tuple element types are identical factors of P3; the new data object represents the subset of points in the state set of P which lie in or near the convex closure of the tuple element values

B **-set_scalar→** (B→TWO)

if the value of the scalar data object is b, the array (B→TWO) associates MEMBER with b and MISSING with all other values of B

(C1,...,Cn) **-set_tuple→** (C1'→...(Cn'→TWO)...)

the Ci are reordered as Ci' to avoid the need for applying any permute transforms when the nested arrays are transformed; if the tuple data object is (c1,...,cn) then the array (Ci→...) associates the MISSING value with all domain values except ci; the array (Cn'→TWO) associates MEMBER with cn' and MISSING with all other values of Cn'.

(D→Y) **-substitute→** (Si→Y)

a new selector Si is declared, of the same class as the atom D, and with the same state set; D is replaced with Si and the data object is unchanged

(Pi→(Pj→Y)) **-merge→** (Pij→Y)

where Pi and Pj are non-overlapping factors of P3; the state set of Pij is the product of the state sets of Pi and Pj, and the array range data objects are permuted accordingly

(Y1,...,Yn) **-position→** (Y1',...,Ym')

where at least one but not all of the Yi is a scalar whose atom is an Si, T or factor of P3, and each Si, T and factor of P3 occurs at most once among the scalar Yi; each of these Yi is deleted as a tuple element; the remaining Yj are assumed to match the dislay type; do the following:

```
for each remaining Yj
  type Yj = (B1→...(Bn→Y)...);
```

```
        for each deleted Yi, ordered to avoid permute transforms
          type Yi = B;
          delete the B element value from the tuple data object and
          set b = the deleted B value;
          if (B is not a factor of P3 and B does not occur among the
          Bj) or (B is a factor of P3 and no overlapping factor of P3
          occurs among the Bj)
            modify Yj to (B->(B1->...(Bn->Y)...));
            set the data object for this new array to associate the
            old value of the Yj data object with b, and associate
            MISSING with all other values of the array domain B;
          end if
        end for
      end for
(Y1,...,Yn) -embed-> (Y1',...,Yn')
    where the Yi match the dislay type; type Yi = (Bi1->...(Bin->Yi")...); let B1',...,Bm' be the
    union of the Bij in all the Yi, including one Bj' which contains the P3 factors occurring in all the
    Yi; do the following:
    for each Yi
      type Yi = (B1->...(Bn->Y)...);
      for each Bj', ordered to avoid permute transforms
        if Bj' does not occur among the Bi
          if Bj' is not a factor of P3 /* Bj' is T or Si */
            modify Yi to (Bj'->(B1->...(Bn->Y)...));
            set the data object for this new array to associate the
            old value of the Yi data object with each b in the
            state set of Bj';
          else /* Bj' is a factor of P3 */
            Pk = Bj';
            if any Pl occurs among the Bi
              /* because Bj' does not occur among the Bi and Bj'
              contains all the P3 factors occurring among the Bi, Pl
              must be a proper factor of Bj' (=Pk) */
              set Bj" = Pk/Pl;
            else
              set Bj" = Pk;
            end if
            modify Yi to (Bj"->(B1->...(Bn->Y)...));
            set the data object for this new array to associate the
            old value of the Yi data object with the minimal b in
            the state set of Bj', and to associate MISSING with all
            other values;
          end if
        end if
      end for
    end for
((B->Y1),...,(B->Yn)) -distribute-> (B->(Y1,...,Yn))
    the new array data object associates the tuple (y1,...yn) with each value b from the state set of B,
    where yi is associated with b in (B->Yi)
```

Appendix B. The Data Type Transform Algorithm

The transform algorithm is given below by a pseudocode description of the recursive function **transform**. Its argument is a type, which it modifies. The applications of basic type transforms are specified using input and output type templates, which bind to the current type and its subtypes according to the atom and type names used. The type argument is assumed to have an associated data object which is also modified. If the argument is an element of a tuple, the modified type and object are replaced in the containing tuple. If the argument is an array range, the modifed type and the values of all the range data objects are replaced in the array. This appendix continues the symbol naming conventions of Appendix A.

```
procedure transform(X)
{
 if X is a scalar
   type X = C;
   if C is unmapped
     C -unmapped-> NULL; /* matches the display type template */
   else if C  = P, T or Si
     C -set_scalar-> (C->TWO); /* matches the display type
     template */
   /* the case C = Qi already matches the display type
   template */
   end if
 else if X is an array
   type X = (C->Y);
   transform(Y); /* now Y matches the display type template */
   if Y = NULL
     (C->NULL) -null_array-> NULL;
   else if C is unmapped
     while Y is an array and (type Y = (B->Y')) != (B->TWO)
       (C->(B->Y')) -permute-> (B->(C->Y');
       let Y refer to Y';
     end while
     if Y = (B->TWO)
       (C->(B->TWO)) -union-> (B->TWO);
     else if Y = Ri
```

```
    (C→Ri) -composite_scalar→ Ri;
   else /* now Y = (Q1,...,Qn) */
    (C→(Q1,...,Qn)) -composite_tuple→ (Q1,...,Qn);
   end if
  else if C = Ri
   type X = (Ri→(B1→...(Bn→Y')...));
   if Y' = TWO
    for k = 1 to n /* permute Ri to the innermost nested
    array domain of X */
     (Ri→(Bk→Y")) -permute→ (Bk→(Ri→Y");
    end for
    (Ri→TWO) -composite_set→ Ri;
   else /* now X = (Ri→Y) */
    (Ri→Y) -substitute→ (Sk→Y);
    apply -permute→ wherever possible to order Sk and
    the Bj;
   end if
  else /* now C = Si, T or P */
   type X = (B→(B1→...(Bn→Y')...)); /* note B = C */
   if any Bi = B or B and any Bi are overlapping factors of P3
    (B→Y) -substitute→ (Sk→Y);
    apply -permute→ wherever possible to order Sk and
    the Bj;
   else
    apply -permute→ wherever possible to order B and the Bj;
    if B permutes to a position (B→(Bi→Y")) and B and Bi
    are factors of P3
     Pj = B; Pk = Bi;
     (Pi→(Pk→Y')) -merge→ (Pik→Y');
    end if
   end if
  end if
 else if X is a tuple
  type X = (Y1,...,Yn);
  if Y1 = ... = Yn = P
   (P,...,P) -scan→ (P→TWO);
```

```
else if all the Yi are scalars and at least one is Si, T or P
 (Y1,...,Yn) -set_tuple-> (Y1'->...(Yn'->TWO)...);
 /* now X = (Y1'->...(Yn'->TWO)...) */
 transform(X);
else
 if at least one but not all of the Yi is a scalar whose
 atom is an Si, T or P and each Si, T and factor of P3
 occurs at most once among the scalar Yi
  for each element Yi which is not a scalar of an
  Si, T or P
   tranform(Yi); /* now Yi matches the display type
   template */
  end for
  (Y1,...,Yn) -position-> (Y1',...,Ym');
  for each element Yi'
   type Yi' = (B1->...(Bk->Yi")...)
   apply -permute-> wherever possible to order the Bj;
   apply -merge-> if possible to combine factors of P3;
  end for
 else
  for each element Yi
   tranform(Yi); /* now Yi matches the display type
   template */
  end for
 end if
 type X = (Y1,...,Yn);
 while n > 1 and some Yi = NULL
  (Y1,...,Yi,NULL,Yi+1,...,Yn) -null_tuple->
  (Y1,...Yi,Yi+1,...,Yn);
  type X = (Y1,...,Yn);
 end while
 if n = 1
  (Y1) -unary-> Y1;
 else
  (Y1,...,Yn) -embed-> (Y1',...,Yn');
  /* now X = (Y1',...,Yn') */
```

```
        for each Yi'
          type Yi' = (B1->...(Bm->Yi")...);
          apply -permute-> wherever possible to order the Bj;
          apply -merge-> if possible to combine factors of P3;
        end for
        type Yi' = (B1->...(Bm->Yi"); /* now all the Yi' have the
        same sequence of Bj, and X = ((B1->...),...,(B1->...)) */
        for k = 1 to m /* apply distribute to each Bk */
          ((Bk->...),...,(Bk->...)) -distribute->
          (Bk->((Bk+1->...), ... ,(Bk+1->...)));
        end for
        /* now X = (B1->...(Bm->(Y1",...,Yn"))...) */
        (Y1",...,Yn") -composite-> (Q1,...,Ql);
      end if
    end if
  end if
}
```