Visualizing an Algebra of Objects

C. M. Holt
Chris.Holt@newcastle.ac.uk
Computing Laboratory, University of Newcastle upon Tyne, UK
NE1 7RU

Abstract
Visual languages provide syntactic features to depict graphs explicitly; they offer an alternative to text for the denotation of structures of relations. The language viz was therefore devised to provide a means for visualizing relational specifications and programs. The increased understanding of object-oriented program design makes it sensible to enhance viz to cope with this style, and so provide a single notation for small and large applications. An object is modelled as a graph, linking interfaces (methods) with internal components (the state). Each interface is itself a graph: structure can be introduced into links among objects. Links may be streams of events. An interface may be exclusive, such that one link prevents other connections (as in claiming a resource); or it may be inclusive, with a link creating a copy of the interface (as in a function invocation).

1 Introduction
Geometric diagrams can often provide a clearer insight into the nature of systems of relations than algebraic equations. The use of a language based on sequences of symbols to describe systems that are not naturally linear or tree-like requires the introduction of a kind of redundancy: components must be named at each point of use. The alternative is to denote a component only once, but to connect it where needed, explicitly illustrating structure without adding to the namespace. This approach is complementary to text; a judicious mixture can be clearer than either one used exclusively.

This affects programming language design in that semantic domains can be modelled as attribute graphs (e.g. [1,4,11]). Computational paradigms that are apparently non-graphical rely on an underlying graphical structure: function trees are transformed into graphs by optimizing common sub-expressions, imperative sequences are mapped into DAGs (directed acyclic graphs) connected in control flow graphs, and partial order semantics for concurrent processes define DAGs of events, sequential control paths linked by communications. Furthermore, attempts to combine functional and temporal aspects of programs in an overall framework seem to suggest that projecting programming attributes onto a single, graphical structure can be more flexible and comprehensible than totally separating the attributes, or modifying the structure according to the behaviour being considered [7].

Given a graph domain, the question of clarity arises: what is the best way to represent a graph? The principle of transparency suggests that semantics is clearest when structure is closely reflected in syntax, minimizing translation effort. The implication is that graphs might be manipulated and reasoned about directly as pictures, rather than via a textual language. Historically, pictures have had a rather dubious formality; but this is with respect to "analog" images, in which information is carried by continuous aspects of the components such as size, position, and shading. A "digital" picture holds information in discrete attributes, that have only a few, easily distinguished cases. For example, the number of faces of a polyhedron is unambiguous when it is small, and orientation is clear if it is limited to a very few alternatives, e.g. allowing only 45° angle differences.

The language viz [8] was intended to provide syntactic conventions for denoting graphs, representing programs as systems of relations, following the approaches of [10,13]. It encourages a style of programming that (i) relies on retracts (identity mappings with constrained domains) for typing, as in Scott's domain theory [14]; (ii) uses a three-valued logic, related to that of intuitionism [5]; and (iii) associates false with the empty set of values, as in Martin-LoF's type theory [12]. Some syntactic and semantic conventions of viz are outlined in Section 2, updated to deal with three dimensions.

Attempts have been made to reconcile the various programming paradigms with one another [3,16]; it appears that focussing on graph domains is helpful when dealing with objects. The required extensions include the introduction of time streams and the treatment of interfaces as first class objects; these are considered in Section 3. Time streams consist of sequences of events/values of varying duration, only one of which is accessible in any given spatio/temporal context. Interfaces must allow for the introduction of structure such
that different components may be linked to one another and to internal components of the object. Interfaces may be graphs which themselves have interfaces; visual clutter may be reduced by restricting views in terms of the values they may transmit.

2 Basic Syntax and Semantics

viz provides syntactic conventions for denoting graphs; this leads to a semantic framework of relations that is intended to facilitate reasoning about specifications, programs, and proofs. Related notations include Petri nets (e.g. (2)) and dataflow diagrams (e.g. Show and Tell (9)).

2.1 Volumes

A relation \( R(p_1, p_2, \ldots, p_n) \) is defined by introducing a syntactic template containing formal parameters, and a body of relational applications that use those parameters. A volume is a visual representation of a relation, and so is similar: its surface is a template containing icons and regions, and its contents consists of sub-volumes related to each other and to the surface. More precisely, a surface is a closed structure of sides (flat or curved) upon which icons and regions play the role of textual symbols; their positions (corresponding to the parentheses and commas found in text) must be unambiguous. In the following, a surface is an oriented polygon with each side normal to an axis \((x, y, or z)\). Regions are distinguished on a side by syntactic properties such as shading, shape and elevation.

The contents of a volume may include such things as a specification, an algorithm, and a proof that a algorithm satisfies a specification. These are all graphs, where a graph is a set of nodes and arcs; arcs are undirected and may link any number of nodes. A node is represented as a region, and an arc is represented by linking regions by juxtaposition, by connections with lines, or by icon.

A region has a direction, which is represented by spatial displacement: an inset region is in, an outset region is out, and a flat region is both in and out. A region also has a kind, which restricts the nature of values that pass between the interior and exterior of a volume. Kinds of regions distinguish between parameters and the relation as a whole. In the following, a parameter is associated with a notch (an inset or outset region with straight sides and a flat base), while the relation is associated with a wedge (with angulated sides). For example, all have surfaces consisting of rectangular prisms with regions on their tops (flat, an inward notch, an outward notch, an inward wedge, and an outward wedge). The volume

\[
\begin{array}{c}
\text{airplane} \\
\text{right wing} \\
\end{array}
\]

has a surface which is also a rectangular prism, but which has five regions, in/notch-in/wedge-in/notch on top and out/notch-out/wedge on the bottom (oriented from left to right).

Structures are built by linking regions of volumes with arcs; for example,

\[
\begin{array}{c}
a \\
c \\
d \\
e \\
f \\
b \\
g
\end{array}
\]

has arcs linking \((a.out/notch, b.out/notch, f.(middle in/notch)), (c.out/notch, f.(left in/notch)), etc. The letters through g are icons. An icon is associated with the relation of its volume; it defines an implicit arc, linked with wedge regions of the volumes it is used on. If a surface with an icon has an in/wedge, the relation is being defined, and the icon arc takes that value; otherwise, the icon defines the relation of its volume.

A well-formed view of a structure is one in which each region of each node is visible or may be inferred from visible arcs. Notches and wedges are designed to be visible from the front, but flat regions cannot remain visible when juxtaposed; thus, an airplane structure involving flat regions must be "exploded":

\[
\begin{array}{c}
\text{airplane} \\
\text{right wing} \\
\end{array}
\]

2.2 Mappings

A volume is associated with a value or operation of an algebra by interpreting inputs as arguments (sources) and outputs as results (targets). Thus,

\[
\begin{array}{cc}
3 & 8 \\
5 & 2
\end{array}
\]

\[
\begin{array}{c}
+ \\
- \\
\times
\end{array}
\]

\[
(3+5)*(8-2)
\]

with the overall result associated with the bottom out/notch. This corresponds to the system of relations:
3(a) & 5(b) & 8(c) & 2(d) & +(a,b,e) & -(c,d,f) & *(e,f,g)
where g is the result.

Structures of volumes need not replace text altogether; a volume with the icon "(3+5)*(8-2)" and a single out region could represent the above, though a parser is needed to determine the value associated with the volume by the icon expression.

The same volume can represent different mappings if its regions have different attributes; e.g:

![Diagram](image1)

is subtraction, with the top in/notch being subtracted from the bottom in/notch to yield the top out/notch. This is a consequence of the relational approach, in which +(x,y,z) is the same relation no matter which of the parameters are explicitly instantiated.

An icon may be associated with more than one pattern of regions; icons are polymorphic (many-shaped). For instance, one might define a schema of interfaces for + such that the sum of all of the top regions equals the sum of all the bottom regions; this would allow the structure

![Diagram](image2)

Mixing notches with wedges provides a mechanism for treating relations (programs) as parameters (data), admitting higher-order operations. Whereas x(y(z)) indicates successive application, (x(y))(z) treats the result of x(y) as a function; the two expressions are:

![Diagram](image3) vs. ![Diagram](image4)

For example, the K combinator λx.(λy.x) [15] maps its argument x to λy.x; the structure

![Diagram](image5)

has the overall result 3.

The contents of a volume may be viewed through an out/wedge region, given a means for transforming an arc into a two-dimensional window that allows a view of the arc's value. A syntactic convention is to link an arc with a side of a window; another is to use an icon within a window. E.g. for the K combinator, we have either of the structures

![Diagram](image6) or ![Diagram](image7)

The two-dimensional outlines of regions (parameters) on a window are associated with the corresponding regions of the volume whose value is being displayed. Thus, in the above, the input of K is linked with the output of the volume seen in the window, and the output of K is the value of that volume.

A "transparent" window is a universal identity mapping (I); it allows any values to pass through. Non-transparent windows have more complicated structure, that may be viewed through another window by linking a two-dimensional out/wedge with another window. An alternative is to specify a window's value using an in/wedge. For instance, the two relations

![Diagram](image8) indicate that the window used to view the contents of K is the identity mapping I (λx.x); the first provides a view of the window's structure, while the second specifies that it must be I.

A window is an interface between a volume and an arc; it has the properties of a region. Regions may be generalized so that they may have structure, restricting or modifying the values that they allow to pass. For example, one definition of a mapping is a triple <Domain, Codomain, Pairs>; this can be represented by a volume whose input region only allows values of the domain, whose contents accepts pairs, and whose output region accepts values of the codomain. The syntax is similar to that for windows; since a region is two-dimensional, its value may be displayed through two-dimensional wedges. The pattern for a mapping is

![Diagram](image9)

where "" represents an operation for pair formation, and the link between "" and "" represents a unification that results in the generation of the second element of a pair as the output, if it exists.

Linear recursive mappings generally satisfy the pattern
"size > 1" and "size = 1" are two mappings sharing the same input; whichever one succeeds returns its argument. "partition" returns two results, one to a recursive call and the other to "step". "or" returns its successful argument. The technique of using identity mappings that place constraints on their arguments recurs with this style, as does the tendency to use mappings that return multiple results.

Various syntactic conventions have been developed for icons that are associated with common mathematical structures. Triangles \( \bigtriangleup \) are associated with trees; rectangles \( \square \) with sequences, i.e. trees with associativity; semi-ellipses \( \mathcal{O} \) with bags, i.e. sequences with commutativity; and ellipses \( \circ \) with sets, i.e. bags with idempotence. Dyadic constructors that generate new values from old ones consist of two such symbols, linked by an underline (e.g. \( \bigtriangleup \mathcal{O} \) constructs a tree from two other trees). Identity elements are distinguished symbols, e.g. filled with black (\( \bullet \) is the empty set). Additional domains (e.g. singletons, or the general domain without identity element) can be indicated by various decorations on the basic symbols.

### 2.3 Logic

A link between two output regions is an assertion that they produce the same value. If the assertion holds the arc has that value; if the assertion fails, the arc has the empty, bottom value \( \varnothing \). The identification of two "distinct" values causes a domain collapse; if \( 1=0 \), then all consequences can be proved, and all values are the same.

The empty bottom value \( \varnothing \) is the identity element for the set constructor \( \bigtriangleup \mathcal{O} \), which generates the power domain of values; this is a boolean algebra, when taken with intersection \( \mathcal{O} \) and complement \( \neg \). An element of the domain is constructable if it can be derived from a finite number of applications. For example, if \( 3 \) is a value in the domain, then its complement \( \neg 3 \) contains every element other than \( 3 \). However, a test for whether a given value \( x \) is in the set requires the successful evaluation of \( x \). The set paradox is sidestepped by requiring the construction of elements; the set of all sets containing themselves cannot be constructed in this way.

An application is evaluated within given resources. If \( \varnothing \) is returned the application fails; if a value is calculated, the application succeeds; and if neither of the above holds, the result is undecided. Strong negation is defined in terms of complement; it maps success to failure and vice versa, while mapping undecided values to undecided:

\[
\begin{array}{c|c|c|c}
\text{not} & T & F & U \\
\hline
T & F & U & F \\
F & U & F & T \\
U & F & T & U \\
\hline
\end{array}
\]

For example, not(3) complements 3, and checks the assertion that the result is equal to the top value \( T \) (the complement of the bottom value). Since \( \neg 3 \neq T \), the assertion fails, and the result is the bottom empty value. "not" is used to define "and", that returns the union of its arguments iff they are both true; "or", that returns exactly one of its arguments, preferring a true result if such is available; and "with", that terminates with a non-U result only if both arguments are non-U.

Evaluation can be modeled in terms of reducing possible truth values associated with arcs. The least knowledge about a possible result is knowing that it will succeed, fail, or be undecided. This is refined by ruling out alternatives; proof of termination rules out the chance of an undecided result, and strictness analysis can rule out failure. The lattice generated by \( T, U, \) and \( F \) is an extended truth domain used to reason about these cases; union means alternatives among truth values:

\[
\begin{array}{c|c|c|c|c}
T & U & F & U \cup F \\
\hline
T & U & F & U \cup F \\
F & T & U & F \\
U & F & T & U \\
\varnothing & \varnothing & \varnothing & \varnothing \\
\hline
\end{array}
\]

The bottom value \( \varnothing \) refers to the state in which no ordinary truth value is possible: it means inconsistency, i.e. systems of equations that have no stable solution. A weak negation \( \neg \) is defined as a complement over this lattice, indicating states that are disallowed; this can be helpful in deontic reasoning and specifying authority. A property that must be true is \( T \); its strong negation is not(T)=F, i.e. it must be false; and its weak negation is \( \neg T=(U \cup F) \), i.e. it must not be true (but it might be \( U \)).
3 Objects

An object is a program module that contains data accessible only through its interfaces. It may have an internal state, and it may be defined by combining other objects. The use of volumes to represent objects requires: (i) the introduction of time streams to reason about state; (ii) the structuring of regions to allow the definition of methods; and (iii) operations on windows to allow the contents of volumes to be used to generate new volumes.

3.1 Time Streams

A time stream is an associative structure like a sequence, except that only one component can be accessed in any given "temporal" context; it always looks like a single value. It undergoes a transition from one value to another when given an appropriate trigger, i.e. the (re)evaluation of a volume that has an output region linked to the stream. Such an evaluation is in turn triggered by a temporal change elsewhere.

Many operations on time streams are implicit: a mapping or relation designed for time-free links works on time streams, with transitions leading to re-evaluation and transition propagation. Temporal properties may be explicitly constrained by referring either to the time of an absolute, global clock, or to the time of a local clock; the resolution of a local clock is finer than that of the global.

The use of a local clock implicitly refers to space, i.e. where the clock is; spatial properties may be explicitly constrained by referring either to an absolute, global position, or to a relative, local position. As with time, the resolution of a local position is likely to be finer than that of the global.

A relation may contain purely functional, purely temporal, or purely spatial information; or it may relate these properties to each other. For instance, a volume may represent a signal wire, that changes the position and a given granularity.

\[ x \text{ overlaps } y \iff \exists z. (z \preceq x) \land (z \preceq y); \]

\[ x \prec y \iff \neg(x \text{ overlaps } y); \]

\[ x \text{ precedes } y \iff x \prec y \land \neg(\exists z. x < z \land y < z) \]

etc.

A programming variable is an object with two interfaces, each of which is a time stream; one is used for modifying the value of the variable, and the other is used for accessing the value. The effect of changing the modification stream is that in the immediately following interval, the access stream is also changed; it then maintains its value until the next modification.

An object has a state at any given time consisting of the values of its links. A state changes when a time stream changes value. With changing state comes the possibility of changing link connections; dynamic reconfiguration can occur when an object links together two external links. This kind of structural modification arises when the contents of a volume are being edited, and the contents of a volume becomes the value of a time stream; it also arises in telephone switching circuits.

3.2 Links through Regions

The values displayed through windows can contain structure; that structure must be separable, linking components with arcs that pass through a window. Icons can carry values through windows, if one defines the scope of an icon to be its enclosing volume, together with any volume that can view the icon’s scope; this may be extended to explicit links through pattern matching.

When dealing with a function that returns a structured result, selection functions can obtain various components. For example, two parts of an expression e might be accessible using first(e) and second(e). Another approach is to construct a pattern with appropriate variables to match the structure. Then equation e = (x, y) is evaluated, binding x and y to the two respective integers.

In window terms, the equivalent of selection functions is like "partition" in the linear recursion template; e.g.

\[ \text{partition} \]

Pattern matching requires being able to select elements of the structure in the window directly, extending arcs through the window to arcs and volumes.
This requires being able to specify arcs and volumes in windows unambiguously, preferably in terms of their position. Problems include how to represent a link through a window when that window is directional; ghost arcs are required that can ensure connection while being unable to affect the contents being linked. It should be possible to leave a volume present; again, some kind of ghost representation seems plausible. It should be possible to leave a volume available for use, while not allowing it to be copied or have its value taken through a window; this requires being able to specify arcs and volumes in windows unambiguously, preferably in terms of their directions.

3.3 Inheritance

A value can be accessed any number of times when it is in a link, simply by creating a new arc to the link. This provides a mechanism for copying volumes: convert the volume to a value via a wedge, link with that value, and inject it into another volume via a wedge. [This underlies the semantics of icons.] Inheritance is concerned with creating an object that copies aspects of other objects.

Regions can be manipulated by expanding them into windows or into volumes. The "methods" generally associated with objects can be understood as regions; so, to create an object that contains all the methods of two other objects, one can obtain the values of those objects and inject them into components held within a window. These components may be volumes, in which case the injection is as above; or they may be windows. This leads to the idea that a window may contain other windows. This containment is like a union operation, except that position may be used to select components.

Other operations can be performed on windows. One is a kind of asymmetric union, in which one component window is partially laid over another; the obscured part of the second window is then not accessible. There is also a kind of intersection, in which a window \( x \) defines the set of methods that may be transmitted through a related window \( x' \); \( x' \) is then used as a view to another window \( y \); it transmits only the components of \( y \) specified by \( x' \).

Finally, when combining windows or volumes, various components of them may be interconnected; such connections may or may not be made accessible to that which views the structure. This provides a way to relate different links one to another [6], allowing them to communicate in various ways.

4 Conclusions

A data-flow visual language denotes graphs representing systems of relations. It can be extended to cope with objects, through the introduction of time streams, state, window structures and inheritance. This supports the idea that three-dimensional structures with two-dimensional windows provide a natural way to reason about programs.

Acknowledgements

SERC project GR/F 92121, IED/SERC project GR/F 98055/4/1/2083, BAe DCSC.

References