Language Spaces

C. M. Holt

Dept of Computing Science, U of Newcastle upon Tyne, NE1 7RU, UK
chris.holt@ncl.ac.uk http://www.cs.ncl.ac.uk/chris.holt/

Abstract

Visual programming gives vast syntactic freedom to both language designers and applications programmers. Unless a flexible framework is found within which widely varying programming styles and paradigms can be embedded, all the problems encountered in creating interfaces between modules in different textual languages will be repeated, exacerbated by syntactic incompatibility. This paper looks at what such a framework might consist of, expressed informally as a higher-order universal algebra of relations; examples are drawn from temporal logics.

1 Introduction

The relationship between semantics and syntax that has developed over the past few decades seems to have the following structure: (i) a semantic domain is described mathematically, such that elements (programming operations and data types) have formal definitions; (ii) an abstract syntax is devised, together with a "meaning" that maps its elements to those of the semantic domain; and (iii) a concrete syntax is invented, grounded in the physical medium to be manipulated, that is mapped to the abstract syntax. Of course, the development process is not usually so tidy; but this is what is generally understood as being required for a language to be well-defined.

The advent of visual language design has confused this three-level hierarchy; the increased expressiveness of concrete syntax has decreased the perceived need for an abstract syntactic layer separating the representation from the underlying meaning. When a graph is denoted "directly" as points linked by arcs, there seems to be little need for a mediating level separating "what you see" from "what you get". Unfortunately, this vision in which a programmer directly manipulates the objects of a program is misleading to the extent that it implies a kind of privilege for the relationships depicted explicitly. Programmers are used to directory listings that offer a variety of ways in which files can be organized (e.g. by size, name, kind, or date); they are less used to this kind of choice when dealing with program structures. It is argued here that maintaining the notion of abstract syntax, albeit modified to deal with the requirements of visual programming, can make multiple views of programs easier to understand.

This depends on an idea of abstract syntax slightly different from that commonly employed in dealing with textual languages. The basic idea is that an entity of a semantic world consists of a number of objects organized into a structure via several relations. A view of such an entity is constructed by selecting one of these relations, using it to generate a space. Then, the objects are depicted as residing in that space, with the spatial structure "denoting" the given relation. As different relations are selected, leading to different spatial representations, a variety of views of the underlying entity can be generated and compared against one another. This leads to a notion of abstract syntax as being the reification of a chosen relation; it is the result of a kind of projection operation applied to a semantic domain. Once a spatial structure has been selected, it is necessary to decide how to represent that structure to an observer; this is the process of refining an abstract syntax to a concrete syntax.

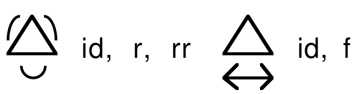
Implicit in the approach suggested above is the notion of a *language space*, a mathematical structure (space) used to describe something (hence "language"). A language space is representative to the extent that its structure matches that of the space it refers to. Elements of such a space have attributes which signify: they refer via a meaning operation to values in a different domain. There is intensionality in that the way something is said (its sense, or "grammatical" structure) carries information about what is meant. Although language spaces are invoked here to explain and justify a visual abstract syntax, examples can be found to illustrate that the basic principles are applicable throughout the scale among semantic domains and among abstract and concrete syntactic domains.

The following section tries to give a feel for what a language space is by presenting both semantic and syntactic examples. The character of such a space depends on both the nature of the space and on the associated meaning operation; the former is characterized more fully in Section 3. Then, an application (the depiction of temporal logics) illustrates how different kinds of spaces can be combined together to form an overall structure.

2 Language Space Examples

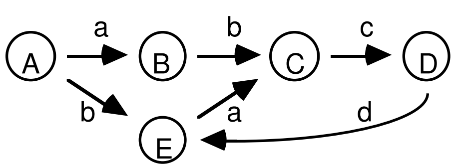
It is standard practice in denotational semantics to define '"semantic valuation functions", which map syntactic constructs in the program to the abstract values (numbers, truth values, functions etc.) which they denote.' [9] It is possible to take a rather more general view, in which a given space S is viewed as syntactic if it has a mapping M of a particular kind to another space T. Linguistically, elements of the source space S refer (via the meaning mapping M) to elements of the target space T. Constraints on the possible structure of S are grammatical rules; the set of possible spaces satisfying those rules, which can be mapped via M to T, comprise a language. S is a language space to the extent that its structure is preserved by M.

An example of a language in this sense is that used to describe the group of invariant operations on an equilateral triangle. This group is generated by r (rotate by 120°) and f (flip about the vertical axis); r has period 3, f has period 2, and the group has six elements {id, r, rr, f, fr, frr}.



It is possible to abstract away from the meaning of the group (triangle operations) and consider the group G in its own right, understood in terms of its inherent structure. When this is done, we need a mapping M from G to the triangle operations if we are to "understand its meaning"; M is a reification operation, an adjoint to the abstraction operation used to obtain G in the first place. Of course, G can have many other mappings M', M" etc. which yield different meanings; a language can have more than one model.

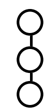
Another example of a space is a directed graph with labelled nodes and arcs.



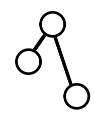
The graph itself has a particular structure which can be analyzed; for instance, the maximum degree of any node can be calculated. It becomes a language space if we introduce a meaning M which (for example) associates nodes with states, and arcs with state transitions. The state transition system can have other descriptions; for instance, one might use right-linear BNF rules, where terminal symbols indicate transitions and non-terminals indicate states. Thus, two descriptions of the same system can have very different structures. Note that a graph is itself a structure to be denoted; when this is done, the space is itself part of a semantic domain. Thus, language chains can be constructed, composing the various meanings.

Programming languages are usually textual; since the development of Algol 60 and BNF, this has generally meant that programs have an abstract linear structure. Since program evaluation/execution is seldom purely linear, there is often a mismatch between the structure of the syntax and that which it describes. Unlike the group and graph examples above, this involves a meaning M which does not entirely preserve structure. Sequences of primitive statements like assignments, inputs and outputs can be understood to have a syntactic structure reflecting what is meant; but the linearization of branching control or data flow (as in conditionals and concurrency) creates a structural gap, where M has to transform shapes as well as components.

```
cin >> x;
y = x*3 + z;
cout << y;
```



```
if  (x==3)
{ y = 5;}
else { y = 6;}
```



To the extent that this occurs, effort is needed to understand what is meant by a given representation. On a small scale, this is not a problem; on a large scale, it can prove insurmountable when trying to understand a program.

At the syntactic level, the mapping from textual characters to symbols (lexical analysis) can also be viewed as defining meaning. An alphanumeric sequence starting with a letter is mapped to an identifier symbol, a sequence of digits is mapped to a number symbol, and so forth. The character space is a language space because the ordering between sequences of characters representing symbols is preserved. The symbol space can then be mapped to another space, and the meaning mappings composed. The character space represents the third space to the extent that its structure is preserved by the meaning composition. For instance, a sequence of assignments, inputs and outputs has the same linear order as the characters which represent them, so part of the character relation is preserved.

3 Representation Spaces

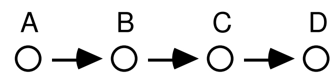
Little has been said about the nature of semantic domains, though reference has been made to imperative features in the examples. It is not the intent here to construct a programming semantics; rather, the concern is with represention. In fact, it could well be that different parts of a program are best constructed with different underlying computational models or paradigms; opinion is divided as to whether a Grand Unified Theory of Programming is possible (or even desirable). However, it would be very helpful if program components could be composed, irrespective of the nature of their contents; so it is worth looking at what would be necessary to make this feasible.

3.1 Points, Attributes and Relations

The first notion that has to be addressed is what a primitive can consist of. Maintaining a geometric perspective, a primitive is called a point. Mathematics tends to be somewhat vague about what points are exactly, just as it is about what values are; the concern is more about what kinds of questions can be asked concerning them. Different metamathematical viewpoints suggest different styles of enquiry; it is not the intent here to choose among the alternatives. However, it is reasonable to say that a point has attributes, that hold information. One possible kind of question is "Does this point have this attribute?" while another is "What is the value of this attribute for this point?" Both of these entail some notion of mapping in the metalanguage. A somewhat weaker question might be "Does this kind of relation hold among the attributes of this point?" which of course entails the notion of relation. These concepts are of equivalent power, since any mapping induces a relation among its arguments and result, while a relation induces a mapping from its contents to the boolean domain. The choice of which should be taken as more fundamental depends on which is clearer for a given application, and may depend on the point in question. [Both can be grounded in syntactic properties for manipulating text; however, this is not particularly helpful in increasing explanatory power when we are concerned with generalizing representations beyond the textual.]

Progress can be made by looking at what is to be done with a point. In particular, much has been made above of spaces, in which (presumably) points can be embedded. This leads to questions of the form "Does this space have any points with this attribute?" or "How many points have this attribute in this space?" Rather than making a once-for-all decision about which is better, this can be associated with the space; thus a space can have attributes independent of the points within it. However, point-independent spatial attributes and the points within a space are not enough to characterize it; there is an additional notion of structure required. This can be defined as introducing relations among the points, where a relation indexes attributes against collections of points. Possible questions include "Are there any points with this attribute in this relation?", "How many points have this attribute in this relation?", and "Given this relation and this point associated with this index, what points are associated with this (other) index?" [It is possible to embed structural information of this kind in the attributes of the points of a space; but then, such spaces have no inherent structure, which makes talking about classes of spaces somewhat obscure. Conversely, all the attributes of a point can be removed by inducing relations in the space that carry the same information. This leads to such complex spatial structures that once again, talking about classes of space is made more difficult.]

So far, then, a point has attributes, and a space containing points has attributes and relations among its points. This can be simplified by observing that an attribute can be understood as a relation with a single parameter (when dealing with mappings, it would be an anadic mapping); and furthermore, there could be relations among the attributes of a point. Thus, points and spaces are both simply associated with collections of relations. As an example, a space of points might have a DiGraph relation, with Source and Target indices; then, we might ask "Does DiGraph allow us to index point A with Source and point B with Target?" [I.e. is there an arc from A to B?] Another question might be "What points are indexed by Target in DiGraph when Source indexes the point A?" [I.e. where can we get to from A in a single step?]



DiGraph(Source=A, Target=B)
DiGraph(Source=B, Target=C)
DiGraph(Source=C, Target=D)

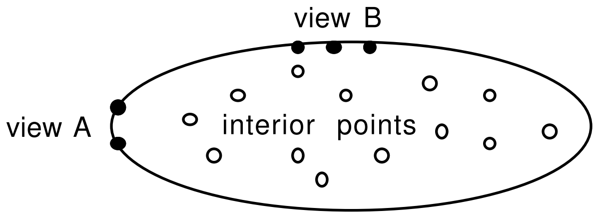
It is assumed here that a relation may have any number of indices, and each index may be associated with any number of points in a given instance. For instance, a relation R may associate the index 1 with points A, B, and C; the index 2 with points D and E; and the index 3 with points A, D, and F. Then, one might ask "What is associated with index 2 in R, when A is associated with 1?" If this question is allowed in the given space, the response would be that points D and E are associated with 2 in that instance.

R(1={A,B,C}, 2={D,E}, 3={A,D,F})

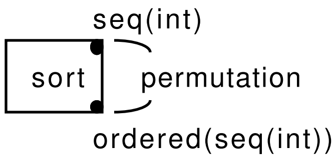
This level of flexibility is enough to provide hypergraphs, n-adic mappings, and multiple-result mappings.

3.2 Spaces as Objects

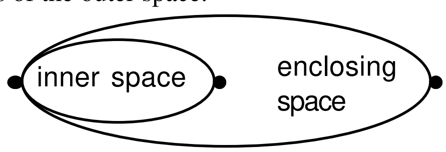
The discussion so far has been concerned with what is in a space; the next concern is with what is visible. That is, a space may be viewed from within, with particular kinds of questions available; and it may also be viewed from the outside, such that different and restricted kinds of questions may be asked. There may be more than one outside view, in which different points and relations are visible (accessible to questioning). A view of a space may be understood as a relation between the space and an observer, and so fitted into the same overall structure of points and relations, if relations are allowed to be higher-order and polymorphic. A space viewed through such a relation is here called an object: points directly accessible to an observer comprise the interface (for that view), while other points are hidden, belonging to the interior of the object.



Accessible attributes of interface points introduce typing information, and accessible relations among interface points introduce specification information.



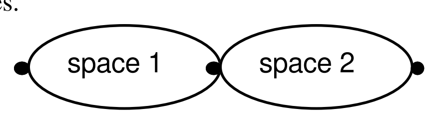
One space may contain another; in some sense, they are related by a "within" relation. The enclosing space will have a view of the enclosed object; then, the interface points of the object are also points of the outer space (and may be interior or interface points of the outer space), and interface relations of the object are relations of the outer space.



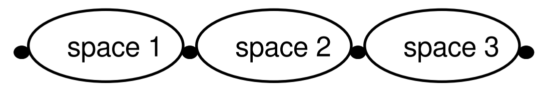
A representation of an object satisfies a separation property if distinct points of the object's interface are distinct in the enclosing space (i.e. the representation does not conflate points together).

When a space contains an object, relations can be defined between elements of its interface and other components of the enclosing space. One of the simplest of these relations is that of identity, which has the effect of unifying its parameters. The meaning of this may depend on the enclosing space, since there are various ways to define the unification of collections of attributes. An attribute might be present in the result if it is present in either of its "ancestors", or it might be present only if it is present in both of its ancestors. If ancestors have an inconsistent attribute, all other attributes might fail, only that attribute might fail, or one ancestor might be given preferential status, such that its attributes "win".

Two objects enclosed in the same space can be composed by unifying one or more points of their interfaces.



This naturally induces a composition of relations involving those points, and allows the composed objects to be viewed as a single, more structured object. [The point of composition could be hidden by embedding the objects in a space with an interface that did not contain that point.] More than one component of an object can be on an interface of its enclosing space; then, the outer interface can have a visible structure. Two objects with structures as interface components can be composed by unifying those structures together; that is, unification may be structural, as well as applying to (shapeless) attributes. Composition is associative, in the sense that a composed structure does not depend on the "order" in which unifications of this kind occur.



Relations may apply among points, among relations among points, among interfaces between objects, among relations between interface relations, and so forth; they are higher order [7], and might be looked at as a relational analogue to morphisms in category theory. A reason for preferring relations is that given the goal of allowing multiple views of a given object, a single relation can associate all of the views with the object being represented; when direction is required, each view would require a separate morphism (with its adjoint, if it is desired to go in both directions). At a trivial level, a single relation suffices to link the number 25 with its Arabic and Roman representations, whereas four functions are needed between semantic and syntactic domains (and two more are needed between the two syntactic domains).

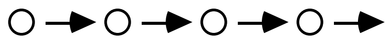
```
R( value=25, arabic="25", roman="XXV")
   vs.
f1(25) = "25";   f2("25") = 25
f3(25) = "XXV"; f4("XXV") = 25
```

This simplification can be important when dealing with program structures of considerable complexity.

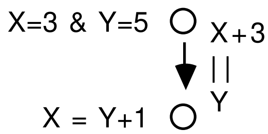
4 Temporal Spaces

The value of the relational approach outlined above can be illustrated by applying it to the domain of temporal logic programming. There are a wide variety of different temporal logics (e.g. [1,2,8]) and ways of describing them (e.g. [3,4,5,10,11]); despite work in combining such logics (e.g. [6]), it does not seem easy to compose program components in the traditional way. The following offers a naive view of how heterogeneous temporal structures can be constructed; it is meant primarily for illustrative purposes.

The simplest model of time is linear, discrete and finite; the temporal domain is isomorphic to an initial subset of the whole numbers. A temporal logic associates predicates with points of time, and it also relates values at different times to one another. This can fit into the framework described above in a number of ways. One approach is to represent the time space as a linear graph, with arcs representing the temporal ordering relation.

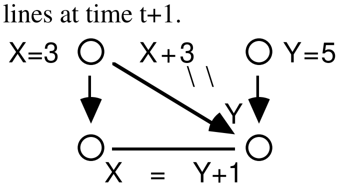


Predicates are attributes of the points; e.g. time t might be associated with "X = 3 & Y = 5", and time t+1 might be associated with "X = Y+1". Then, the arc from t to t+1 might be associated with "Y@+ = (X+3)@–", where "@–" indicates the time directly preceding the relation, and "@+" indicates the time directly following the relation. [This can be written more pictorially without the textual operations.]



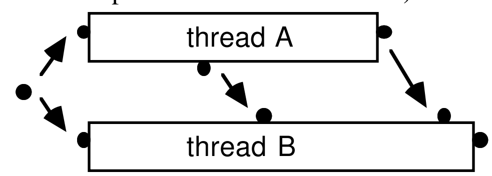
We can then calculate that at time t, X+3 has the value 6, so that is the value of Y at time t+1; and this means that X has the value 7 at time t+1. It is possible to relate non-adjacent times by introducing additional arcs, without modifying the point structure. A relation among more than two temporal locations requires the introduction of a notation for associating expressions with the times they are to be evaluated at (e.g. by using coloured links and coloured "@"s in predicates).

Another possible time space for this time model might arise if there was concern about the values of a given variable, e.g. if a time sequence required debugging. That variable might be separated out from the main time line, so that from the point of first use to the point of last use, one line would be for the variable, while another line would be for everything else. Relations between the given variable and any others could be indicated by arcs (or coloured links) as before; for example, if X and Y were to be separated in the above example, there would be an arc from the X line at time t to the Y line at time t+1, and an undirected arc between the X and Y lines at time t+1.

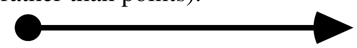


The question of stability (whether a variable has a well-defined value at a given time if it is not explicit, i.e. whether old values are preserved by default) is not addressed here; it would be a property of the time space.

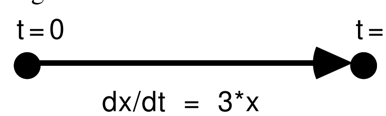
The separation of a variable's timeline from that of the temporal program as a whole suggests that time structures can be non-linear. This can be generalized, allowing arbitrary directed acyclic graphs to represent time structures, points and arcs still being associated with predicates and temporal relations respectively. The spatial temporal relation then defines a partial rather than a total order; points that are unordered with respect to each other can be understood as representing concurrent states (this need not imply that they occur simultaneously). Linear time sequences can be represented as named thread spaces, with communications channels being interfaces on those threads; operations can be introduced for partitioning state spaces (needed when a thread is spawned) and combining state spaces (needed when a thread dies, if its state is to be passed on to another thread).



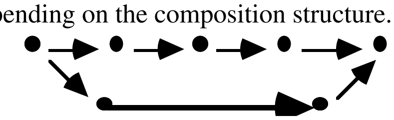
Another temporal variant is to allow time to be continous in a given space (this might be represented using lines, rather than points).



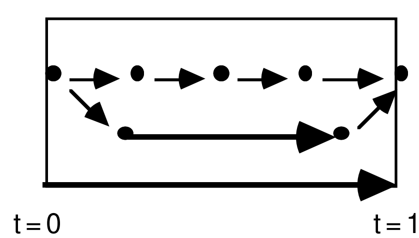
Predicates can still be associated with points in time, and relations can be defined among such points; but there is also the possibility of introducing continuous relations between time and other attributes. Such relations might take the form of differential equations.



Continuous and discrete temporal spaces may be composed by unifying interface points, yielding either total or partial orders; thus, a continuous process might be parallel to a discrete one, or one might turn into the other, depending on the composition structure.



A language space can have a metric built into it; for instance, a 2D space might use one dimension to carry semantic information. Points in the space could be bound to values for attributes, depending on where they are with respect to that axis. For instance, if the x axis is bound to time, then it acts as a global clock for events in that space; their time depends on where they are.



Timed spaces of this nature can be discrete or continous, and composed either sequentially or concurrently; such decisions are left up to the designer of the space in which it is used.

5 Conclusions

The temporal examples of the previous section illustrate how different semantic structures can be composed into a single program, by unifying their interface points. This can be extended to a variety of programming styles. For instance, a Prolog procedure can be combined with a Haskell function by unifying a relational parameter of the former with either an input or an output of the latter. In the former case, the Prolog variable should be instantiated for the composition to make sense; in the latter, the Prolog should be unconstrained. [Questions of type inferencing and scoping of module contents are beyond the scope of this discussion, though it is observed that the values exported by a module comprise its interface under a particular view.] An imperative procedure is no more difficult to compose, when its parameters are either input or output; a reference parameter must be able to change over time, and so requires a time line to describe its interface behaviour. Concurrent threads and processes are no different in nature, though they have additional interface elements (links) which again may require time structures in their specification. An attempt to unify such a time structure with a single point simply fails, of course.

The semantic diversity outlined above can be matched by syntactic variability. A space may be represented in any number of dimensions, though its concrete syntax is restricted to at most 3D; but objects of different dimensionality can be composed just by unifying interface points. Thus, a program may contain linear, planar, and volume spaces, all linked together. Since text is a linear space whose points have character attributes, this means that a visual language as envisaged can easily include text as a sub-language; there is no need to make an either/or choice. However, it might be helpful if a textual piece of program had a visually-oriented view of its interfaces, to simplify the task of composing it with other program components.

References

[1] J Allen, Maintaining Knowledge about Temporal Intervals (CACM 26 11, 1983) 832-843.

[2] D Gabby, Modal and Temporal Logic Programming, in: A Galton, ed., Temporal Logics and their Applications (Academic Press, London, 1987) 197-237.

[3] C M Holt, An Algebra of Lines and Boxes, in: Proc IEEE VL94, St Louis (IEEE CS Press, Los Alamitos Cal, 1994) 55-62.

[4] F Jahanian and A Mok, ModeChart: A Specification Language for Real-Time Systems (IEEE Trans SoftEng 20 12, Dec 1994) 933-947.

[5] G Kutty et al, Visual Tools for Temporal Reasoning, in: Proc IEEE VL93, Bergen, Norway (IEEE CS Press, Los Alamitos Cal, 1993) 152-159.

[6] Z Manna and A Pnueli, The Temporal Logic of Reactive and Concurrent Systems (Springer-Verlag, Berlin, 1992).

[7] Karl Meinke, Universal algebra in higher types (Theoretical CS 100, 1992) 385-417.

[8] B Moszkowski, Executing Temporal Logic Programs (CU Press, Cambridge, 1986).

[9] J E Stoy, Denotational Semantics (MIT Press, Cambridge MA, 1977).

[10] J Tsai, S Yang and Y-H Chang, Timing Constraint Petri Nets and Their Application to Schedulability Analysis of Real-Time System Specifications (IEEE Trans SoftEng 21 1, Jan 1995) 32-49.

[11] H Vestli, Visual Specification of Actor Configurations, in: Proc IEEE VL94, St Louis (IEEE CS Press, Los Alamitos Cal, 1994) 110-117.