Comparing Visual and Textual Languages

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/

Keywords

semantics of programming languages

Abstract

As visual languages become more popular, it becomes more important to have a formal framework for reasoning about their syntax, semantics, and the relationship between them. This paper is concerned with outlining a possible approach. It begins by looking at textual languages from a geometric point of view, and goes on to suggest how this may be generalized so as to be useful in understanding visual (especially dataflow-style) languages.

1 Introduction

The advent of visual programming and specification languages has posed a challenge to those interested in the relationship between syntax and semantics. Concrete syntax has had to be generalized, with textual grammars extended to graph grammars; abstract syntax cannot be viewed in terms of parse trees for those languages that explicitly depict graphs and hypergraphs; and perhaps most importantly, the question of selecting an appropriate underlying semantic model has to be reviewed when there is a major shift in syntax.

Such a review is necessary because there are generally a number of possible foundations of equivalent power. In pure mathematics, set theory (e.g. ZFC) can be used as a basis, or category theory can be used; and each can be defined in terms of the other. In programming, Turing machines can be chosen as an underlying, operational model, or the lambda calculus can be chosen as a functional model; and again, each can be defined in terms of the other. In both these cases, powerful concepts are imported from the meta-language used to describe the foundations; precision is added by relating the concepts to one another and restricting their use; and everything else is defined in terms of this small, precise collection. For example, rewriting rules import notions such as equivalence, associativity, and replacement from operations inherent in ordinary text manipulation; if we did not know that different occurrences of a word were supposed to indicate the same word, we would have great difficulty in formally expressing the idea of referential transparency.

One possibility when considering various alternatives as bases of visual language semantics is to look at those developed for text, and pick from among them; a dataflow language could have a functional basis, a Petri net language could have an operational basis, and so forth; and to a large extent, that is what has happened. It is however potentially limiting; as syntax becomes richer, it is possible that keeping restrictions suitable for text could cause us to forget or ignore new, additional kinds of expression [3]. This paper is concerned with exploring the kinds of things we might want to say, and how we might go about saying them.

It begins by looking at textual languages from a visual point of view. It seems likely that a useful visual language should have a textual subset, since text has been carefully optimized for certain kinds of statements over a considerable period of time. This examination of text involves the use of allegories, related to categories much as relations are to functions. A meaning-representation relation is introduced, and chains of these relations are constructed.

Then an informal notion of size is introduced; it is argued that a good representation is one in which small changes in style lead to small changes in size, and small changes in meaning require only small changes in representation.

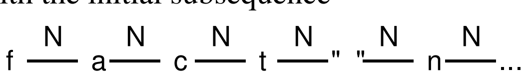
Two related kinds of composition structures are introduced that tend to be difficult to describe easily in text, these motivate the visual approach. The first is non-hierarchical composition, in which (e.g.) three objects A, B and C may be composed such that A is composed with B, B is composed with C, and A and C are also composed with one another. [The popularity of commutative diagrams in category theory suggests that text is inadequate for describing this kind of structure clearly and simply.] The second is multiple-result composition, in which mappings can be composed directly with individual results of multiple-result mappings; for example, a mapping f might return two values, one of which is to be an argument for g, and the other of which is to be an argument for h. [The popularity of pattern-matching tuple results in languages like Miranda suggests that the feature is useful, though Miranda requires the use of intermediate variable names or auxiliary selection functions when composing mappings in this way.]

2 Text: Syntax and Meaning

A piece of text is a sequence of characters, so we can start looking at its structure by making this sequence explicit, introducing an adjacency relation between neighbours. As an example, the text

fact n = if n = 0 then 1 else n * fact (n-1)

contains 44 characters (including spaces), so we can represent it as a chain of 44 elements with the initial subsequence

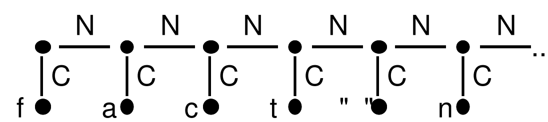


where " " indicates a space. [Note: diagrams like this are not self-contained, in the sense that additional information would need to be available for them to be unambiguous. Such information might be available via hypertext link in a real system.] N indicates a dyadic non-commutative neighbour relation; a textual representation of the above could consist of a conjunction of propositions involving it, of the form

N(f,a) & N(a,c) & N(c,t) & N(t," ") & ...

Looking at this geometrically, each letter can be viewed as representing a point, where the point has an attribute which is the letter. In this case, the letter can represent the point because it has no other important attributes; however, making this explicit, and tagging each point with its attribute, we can represent the structure as

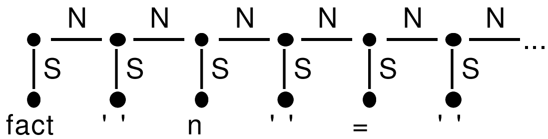
The notion of an attribute of a point is a kind of shorthand for the existence of a relation between that point and another point associated with the value of the attribute. Making this explicit in the representation, we have



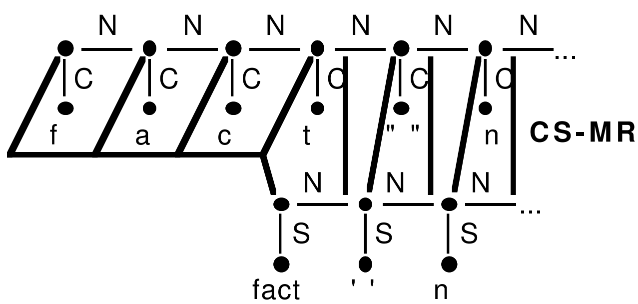
where C is a relation between a point in the text and the points associated with characters. When two points in the text have the same attribute, they are related by C to the same value point; thus in this example three text points are related to the "f" point: the first one, the point for the "f" in "if", and the point for the "f" in the recursive "fact" call.

It is possible to extend this further; a character point can be viewed as having both a value (its meaning) and a representation (perhaps a bitmap). Alternatively, a text point might be directly related to a representation, though this would mean that different representations of a character (e.g. bold or italic) would be considered different characters altogether. Such additional precision is not needed here, however, and will not be pursued.

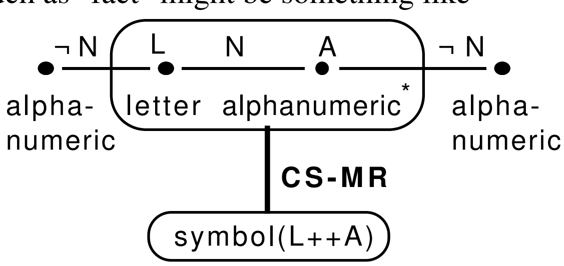
Given the structure of points and relations above, we can ask what it means. A linguistic utterance is (usually) a representation of a structure in some other domain which (often) cannot be directly represented; it refers to something. Letter-based languages use sequences of letters to refer to symbols; so we can start to determine the meaning of the example by relating it to a sequence of symbols. Using the same N relation as above (allowing it to be polymorphic), and introducing an S relation for symbols analogous to the C relation for characters, the initial subsequence in the symbol space can be represented by



Looking more closely at this, the first four characters are related to the single symbol 'fact', the next character (the space) is related to a space symbol, the next character is related to an 'n' symbol, and so forth; the relation has varying arity. Furthermore, the N relation between the "t" of fact and the following space is related to the N between 'fact' and ' '; that is, the meaning/representation relation relates both objects (points in the character and symbol spaces) and relations among those objects:



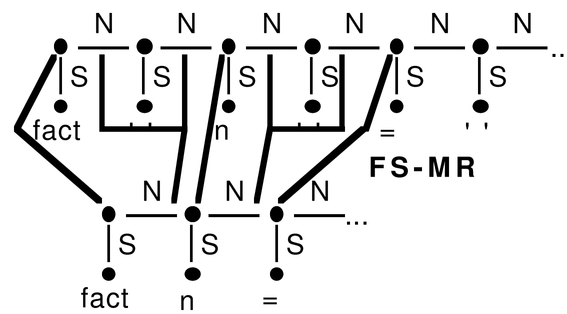
where the character-symbol meaning/representation relation **CS-MR** is in bold. This relation can be defined as a collection of relations among parameterized patterns; for instance, the rule for identifiers such as "fact" might be something like



where a negated relation (such as ¬ N) indicates that no relation of that form is present. [This is not intended to define a notation for defining pattern matching among arguments of relations; it is merely intended to indicate how such a pattern match can be localized and have negative constraints, in the way that extensions to Petri nets can have inhibitor arcs.]

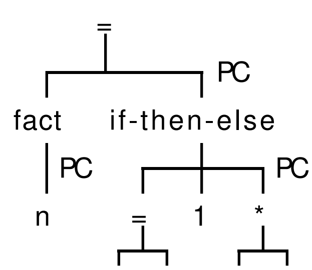
When dealing with mappings that take objects to objects and morphisms to morphisms, it is standard to introduce identities and associativity over morphism composition, and call the mappings functors, i.e. morphisms over categories. The analogy to a category for relations is an allegory [1]; thus, meaning as defined here is a relation over allegories which preserves (some of) the structure of the spaces being related. [There does not seem to be a special word for relations over allegories; "relator" is a plausible possibility and will be used here.]

The decision to introduce a space symbol was taken so that every character takes part in a meaning. Then, another relation can be introduced, that forgets about the spaces:

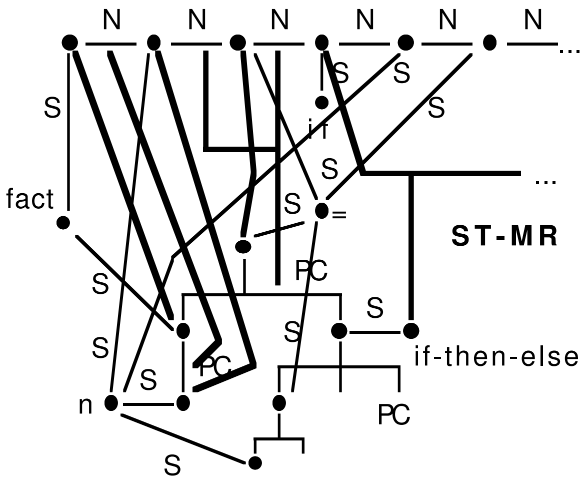


In this case, the relator **FS-MR** is dyadic among objects, but not among relations.

The meaning of a sequence of symbols depends on how the language is defined. When a set of BNF-style production rules are used to define a context-free grammar, it is common to view the meaning as being a parse tree with a structure reflecting the application of those rules. In this example, the tree begins



where PC is the relation between parents and children, and has varying arity. [Different relations can be introduced for the various children, of course; it is up to the designer of the relation domain as to whether uniform arity is worth the increase in the number of elements.] Making points explicit and adding the S relation to the structure, the relator **ST-MR** between the symbol sequence and the symbol tree starts as



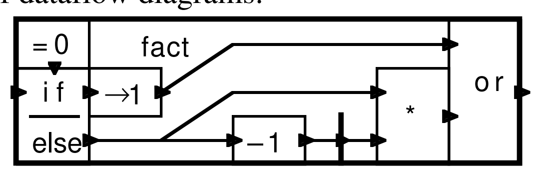
[As can be seen, explicit graphical representation of these relations can make the structure difficult to interpret. It was done here to show that while such things can be done, choosing a single visual representation for complex structures is not always best; a mixture of styles is often clearer.]

The next step is to go from (abstract) syntax to semantics, which can be done in the usual way and so will not be pursued here (though for consistency, relations rather than mappings would be used). [The process can be simpler if some semantic analysis is done when generating the parse tree (i.e. when defining **ST-MR**), because it can be easier to resolve ambiguities in linear rather than tree structures. An example is the way that '=' is used for both definitions and testing for equality.] The various relators can be composed with one another to obtain the overall relator between a concrete syntactic representation in terms of sequences of characters, and the semantic structure it represents.

One thing that the tree form of the structure tends to hide is that different branches of the tree are often intended to refer to the same underlying semantic object. In the example, there are three such "shared" objects: fact, which occurs twice; n, which occurs four times; and 1, which occurs twice. At some stage in the transformation from syntax to semantics, this should be made explicit. One of the attractions of visual languages is that sometimes this conversion from tree to directed graph can be avoided, because the graph structure can be directly represented in the syntax; this will be seen for the parameter n in the following section.

3 Diagrams: Syntax and Meaning

The process outlined above for textual programs can be generalized and applied to non-textual programs. The first point to note is that the concrete syntax is richer; objects can be structures of lines, surfaces and volumes, as well as merely points. As a result, the relations combining objects must be extended; linear neighbour (N) and parent-children (PC) relations are not enough. We can start by looking at a representation of fact in the 2D viz [2] style of dataflow diagrams:



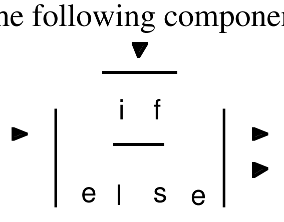
The meaning of this is as follows. Arrowheads (▶) indicate the direction in which data can pass through an interface. Objects are represented as boxes, which may be named or which may have contents explicitly represented. When both name and contents are present, the name is defined to be the box with that contents; thus, the overall box here is named fact; it has one input and one output.

The input goes into an if-else box. In dataflow, it is natural to think of boxes acting as barriers (like retracts) and data switches; instead of true-false relations, a box succeeds by letting data through, and fails by blocking it. If-else applies a condition to incoming data; if that condition holds, the data is passed to the if output, while if it fails to hold, the data is passed to the else output. Here, the condition is (=0);

i.e. any inputs which have the value 0 go to the if branch, and everything else goes to the else branch. The if branch leads to a (→1) box, which is another name for the K combinator applied to 1; i.e. whenever it receives an input, it outputs the value 1. It is strict with respect to failure; when there is no input, there is no output. The (→1) box leads along a "wire" to an or box, which again is interpreted in terms of success/failure rather than truth values: if either of its inputs succeed, that input value is passed to the output. Thus, if the input of fact is 0, it passes through the if, is transformed to 1, and flows through the or to be the overall result.

If the input of fact is not 0, it passes through the else branch to a "wire" which splits into two. The upper branch goes to a (*) box; the lower branch passes through a (–1) box, decrementing the value, and then through a bar linked with the wall of fact. By syntactic convention, this means that the bar represents the contents of the box; i.e. it is a recursive call. The result of the recursion is passed to the (*) box, and the result of that goes to the or and then out of fact. Thus, the result of the else branch is n*fact(n–1), as we would hope and expect.

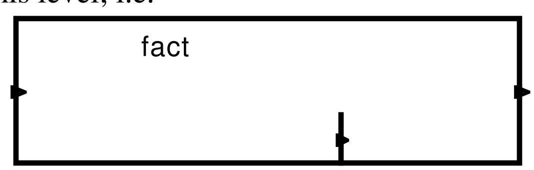
The basic syntactic units are characters, icons (e.g. arrowheads), and lines. These are combined into structures in a number of stages, just as sequences of characters are combined into graphs when dealing with pure text. For instance, the if-else box has the following components:



i.e. six characters, four arrowheads, and five line segments. The relations among these are relative position, adjacency, adjoining and overlay. The first step is to combine characters into symbols, as with normal text; this generates the symbols 'if' and 'else'. The next step is to combine the symbols (and the separating line) into a name, and also to combine the edges of the box into a rectangle. After this, there are six components: the if-else name, analogous to the if-then-else name of the parse tree above; the rectangle; and the four arrowheads. Finally, these are combined to form a box, which is a single object at the abstract syntax level.

Some boxes require the parsing of sequences of symbols, as with text; for instance, the (=0) box has a name that is really an expression. This illustrates how visual components can be understood as an extension of text, rather than as a pure replacement; thus, the benefits of text (conciseness, default structuring) are not lost.

The example has 13 objects at the abstract syntax level: 7 boxes (6 interior and the enclosing one), 3 arguments within boxes (the 0 and 1s) and 3 line structures. Some components of these objects are shared; for instance, the input interface of the enclosing box is shared with that of the if-else box. It is assumed that the recursive call and the box name are part of the enclosing box at this level; i.e.



is a single structure. By comparison, the textual fact has 14 objects at the abstract syntax level, assuming parentheses are analogous to box boundaries and so can be associated with the main operation they enclose. On the one hand, text requires parameters to be named explicitly at each point of use; on the other hand, a diagram can require explicit structuring information, e.g. the lines connecting separate boxes.

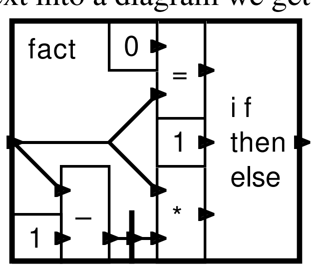
4 Comparing text and diagrams

There are a number of ways to compare different styles of expression; these include conciseness, understandability, expressive power, and flexibility. Work on program complexity (e.g. [6]) has not generally been extended to visual languages, simply because many of the traditional metrics are not applicable; it is hard to talk about lines of code when code is not written in lines. One approach is to count components at the levels of concrete syntax, abstract syntax, and semantics, though this can lead to misleading results; for instance, at the concrete character level a program with single letter variables is often harder to read than one with meaningful names. It can be argued that counting abstract syntactic components is a natural progression from Halstead's measure involving simple token counts, and that counting semantic components is similarly a progression from his measure involving numbers of distinct tokens. Looking at the fact example, then, we have:

fact	concrete syntax elements	abstract syntax elements	semantic elements
text	44	14	9
diagram	57	13	10

where the semantic elements of the text are {fact, n, =(def), if-then-else, =, 0, 1, *, –} and those of the diagram are {fact, =, 0, if-else, →, 1, –, *, or, the line running through the recursive call}. [The other lines merely express identities, and so are equivalent to multiple instances of n.] The diagram has considerably more concrete elements because of explicit rather than implicit argument-result links; otherwise, the two are quite similar (partly because this is such a small example).

A rather more interesting comparison involves the translation of the underlying semantic structures of each into the other style; for example, an if-then-else box is defined that takes three arguments (one a boolean) and returns a single result, while an if-else textual entity is defined that succeeds on one of its two outputs. Translating the text into a diagram we get



while translating the diagram into text we get

fact n = X → 1 or Y* fact (Y–1)
where (X,Y) = if-else n (=0)

The syntax-semantics comparisons for these yield

fact	concrete syntax elements	abstract syntax elements	semantic elements
text	59	24	14
diagram	57	10	8

The textual semantic elements are {fact, n, =def, X, →, 1, or, Y, *, –, where, if-else, =, 0} and the diagram semantic elements are {fact, 0, =, if-then-else, 1, –, *, the line through the recursive call}. The diagrammatic approach is stable under the translation; it can easily accommodate the change of style. However, the textual approach is less so because of multiple results; additional variables are needed. The problem becomes more noticeable as programs get larger and hierarchies of namespaces are introduced; text requires the introduction of more names simply because they are used to indicate links between different parts of a program.

It may be noticed that the diagrammatic version of the functional (textual) fact is actually smaller than that produced as dataflow. A question that naturally arises is whether the dataflow style actually has any benefits, if it leads to larger programs. There are several issues here. One is psychological; dataflow reasoning goes from the input to the output, and so allows program developers and readers to use operational intuitions. In contrast, functional reasoning starts at the end and works backwards. Furthermore, lazy and eager evaluation strategies yield different results; this ambiguity is less important in dataflow.

Reasoning about the temporal properties of programs [4,5,7,8] seems easier with dataflow; guards appear earlier, so less effort is needed in deciding which branches need to be pruned off where. It is less important to have to know about the semantics of components (e.g. strictness properties). This is especially important when there are connections between parallel branches.

References

[1] P J Freyd and A Scedrov, Categories, Allegories (North-Holland, Amsterdam, 1990).

[2] C M Holt, Visualizing an Algebra of Objects, in: Proc IEEE VL91, Kobe Japan (IEEE CS Press, Los Alamitos Cal, 1991) 225-230.

[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] J Tian and M V Zelkowitz, Complexity Measure Evaluation and Selection (IEEE Trans SoftEng 21 8, Aug 1995) 641-650.

[7] 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.

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