Heterogeneous Visual Languages
– Integrating Visual and Textual Programming –

Martin Erwig & Bernd Meyer
FernUniversität Hagen
58084 Hagen, Germany
[martin.erwig | bernd.meyer]@fernuni-hagen.de

Abstract

After more than a decade of research, visual languages have still not become everyday programming tools. On a short term, an integration of visual languages with well-established (textual) programming languages may be more likely to meet the actual requirements of pratical software development than the highly ambitious goal of creating purely visual languages. In such an integration each paradigm can support the other where it is superior. Particularly attractive is the use of visual expressions for the description of domain-specific data structures in combination with textual notations for abstract control structures. In addition to a basic framework for heterogeneous languages, we outline the design of a development system that allows rapid prototyping of implementations of heterogeneous languages. Examples will be presented from the domains of logical, functional, and procedural languages.

1. Obstacles to a VL breakthrough

In the last decade a lot of visual programming languages (VPLs) have been designed and implemented, but in practice, visual programming still is the exception to the rule. Why is there so little acceptance for VPLs in general? There is at least one reason independent from the advantages and disadvantages of visual programming: Software design has a strong tendency to keep to well-established programming paradigms, even if improved languages are available. Partly this is due to investments made in the older paradigm, and partly the reason is to be found in personal reservations of software developers who are afraid of the efforts required to master a new paradigm. This is why languages like Fortran and Cobol are still used in so many places, and it could well be a reason for visual programming not to become a standard in the near future.

Some reasons resulting from design problems of visual programming languages are evident, as well. Experience with graphical user interfaces and VPLs has shown that visual notations are most efficient when taken directly from an application domain, especially if the user is already familiar with them instead of having to learn an entirely new visual notation devised from scratch by the VPL designer. Yet, in contrast to *special purpose* VLs and graphical user interfaces, only very few visual *programming* languages are employing domain specific visual notations.[1] This is not so surprising, for most VPLs are designed to be general purpose languages and thus cannot commit to any particular application domain. Nevertheless, with a design philosophy like this the integration of visual expression is neglected exactly where it might be most useful. On the other hand, VPL designers are trying to find visualizations for all kinds of abstract programming concepts like data structures, control structures, abstraction, functions, variables, etc. While in some cases there are easily comprehensible and usable visual counterparts (e.g., data-flow graphs [3]), it is very hard, if not impossible, to find adequate visual equivalents for several other concepts. Interestingly enough, exactly those concepts for which convincing visualizations are difficult to find may conveniently be described by texts. This is, e.g., the situation with complex control structures and recursion. There seems to be evidence that more concrete concepts (e.g., data structures or simple control structures like iteration over a number of elements) can in many cases best be described visually, while highly complex and abstract concepts (like recursion and functional abstraction) can often better be described and explained textually [4]. In general, no programming language can do entirely without using texts.[2]

Observing this, it seems a reasonable goal to integrate visual and textual languages such that both modes can be used in parallel, each where it is superior. We are proposing a schema for heterogeneous visual programming languages (HVPLs) that combines the convenience of visual notations with the abstraction power of textual languages. In our framework domain-specific *visual* notations can easily be integrated with conventional *textual* programming languages, and thus standard programming languages can readily be specialized as application-specific HVPLs. The design of a HVPL does neither enforce nor restrict the usage of textual or visual expressions in an actual program. Both types can be arbitrarily mixed and

1. Among these exceptions are, e.g., LabView [1] and MPL [2], a language that could be characterized as being heterogeneous in our sense.

2. In this context it is interesting to observe that almost every visual language is using textual labels to indicate non-local connections between different parts of visual programs.

1049-2615/95 $4.00 © 1995 IEEE

can even operate on the same structures. The programmer is entirely free to choose whatever fits best and to find his personal balance between both modes. The textual basis for a HVPL is usually some standard programming language (C, Lisp, Prolog, ML, etc.). In combination with the programmer's freedom to control the amount of visual expression actually employed, this guarantees a smooth migration from textual to heterogeneous languages with an extremely low learning threshold.

The remainder of the paper is structured as follows: In Section 2 the general notion of an HVPL is introduced, and Section 3 outlines the structure of a prototyping system for HVPLs. In Section 4 we will give a semi-formal framework for the integration of visual sub-languages in textual languages. In particular, we will demonstrate the compilation of picture expressions and their integration with textual source code. Some reasonably sized examples will be given in Section 5, and finally, Section 6 presents conclusions and shows directions for future work.

2. Heterogeneous programs

How does an HVPL integrate visual and textual expression? The basic idea is that a programmer often wants to use his favorite standard programming language, while at the same time he would like to be able to use graphical notations from the problem domain in his code. These graphical notations should not only serve as illustrations, but should rather be real program code, in the sense of an exact description of some part of an algorithm or a data structure. Therefore, the concept of HVPLs allows to integrate a visual sub-language *V* into a textual language *L* such that *V*-expressions can be used as substitutes for certain *L*-constructs. The HVPL environment translates the *V*-expressions used in a program into their textual equivalents and reintegrates these translations with the textual parts of the original source program according to an extended grammar for *L*. As the target language for picture translation is as well *L*, the compilation output for a heterogeneous program written in *L+V* is an *L*-program. This, of course, can be processed by any standard interpreter or compiler for *L*. A side advantage of this scheme is that we can even modify and revise existing programs written in *L* using visual expressions from *V*. The resulting heterogeneous programs can then be reprocessed and can again be translated by the original compiler for *L*. Thus, we achieve a way of maintaining and extending programs in a heterogeneous visual language which have originally been written in a textual language. Hence, there is no need to discontinue the maintenance of existing software when migrating to a visual environment.

To give an impression of programming in HVPLs, let us give a first (toy) example. Assume we need to work with finite state diagrams. As finite automata can be deterministic or non-deterministic, a suitable choice for such a task is a logic language like Prolog. To obtain an HVPL for this domain we extend Prolog by a visual sub-language of state diagrams to be used as replacements of complex Prolog structures, see Figure 1. In such a language a program

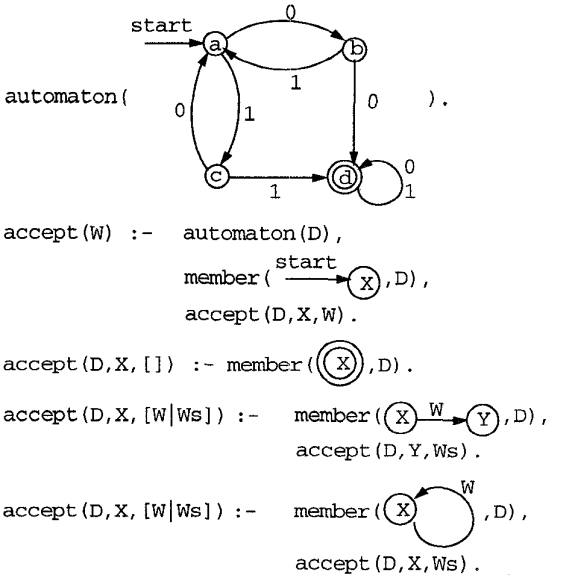


Fig. 1: A Heterogeneous prolog program handling state diagrams

to test whether a string *W* is accepted by a given automaton can be written without having to textually code the automaton or any access to its structure. The HVPL environment compiles each picture into a corresponding Prolog structure, which then replaces the picture in the resulting code. The third rule of `accept`, e.g., is translated to

```
accept(D, X, [W|Ws]) :-
 member(trans(X, Y, W), D), accept(D, Y, Ws).
```

In this case we are only extracting data structures from visual expressions, and each picture is locally replaced by a textual data structure. More complex examples are possible in which picture translations are embedded non-locally or where control information is extracted in addition to data structures. This will be discussed in Section 4.

The only customizations needed to implement this HVPL on the basis of Prolog are a grammar-like rule file that specifies the translation of state diagrams plus a trivial extension of Prolog's attribute grammar to describe the embedding of picture translations into textual Prolog code. Full examples together with these specifications will be given in Section 5.

3. Design of the programming environment

The prototyping environment provides a framework for editing and compiling heterogeneous programs and can

be customized for different heterogeneous languages with minimal effort. In our approach processing an HVPL proceeds in three phases. First, the *V+L* program is split into a purely textual part replacing all the pictures by unique identifiers. Additionally, a database of pictures indexed by these identifiers is created. The second phase is the translation of each picture into its textual equivalent. In the last phase the converted textual source code is parsed according to an attribute grammar for *L*+pictures and is then recombined with the pictures' textual equivalents by means of syntax directed translation. The resulting *L* program is processed by a compiler for *L* (Figure 2).

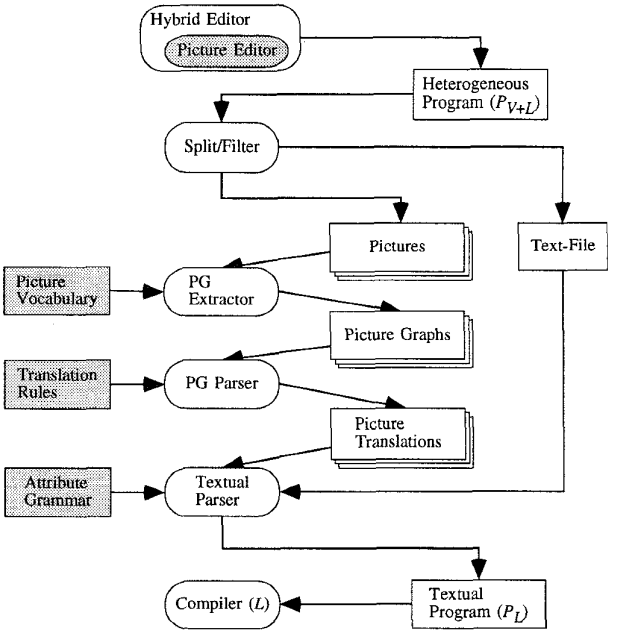


Fig. 2: Design and customizability of the prototype environment

The translation of pictures consists of two different steps. First, a picture graph (PG) representing its structural and geometrical properties is extracted from each picture. In a second step, a production system works on these PGs to generate the textual translations. All these steps are treated in depth in the next section. There are four tailorable modules in the system (shown shaded in Figure 2). The first part to be customized is the PG extractor that generates the PG database. As we understand pictures as sets of spatially embedded objects (e.g., *circles*, *rectangles*, *points*, *lines*) with certain spatial relations (like *inside*, *intersects*), the implementor must specify the object set and the relation set and must provide methods to test the spatial relations. These methods are defined as Lisp/CLOS functions. The second part to be customized is the PG parser which generates textual translations from PGs. The implementor must define a production rule set as the specification of picture translation.

Both, the PG extractor and the PG parser, must individually be configured for a new visual sub-language. If the host language or the mode of embedding changes, the textual parser must be adapted by specifying an attribute grammar for the new host language. For local embeddings a standard grammar can be used almost without modifications. (An embedding is local if for every picture only a single block of text is generated which is replaced into the very position that the picture occupied before).

Most of the work of tailoring the system for a particular HVPL is spent on the HVPL editor. The base system provides a MacDraw-like graphics editor and a heterogeneous editor that can handle pictures and texts in the same flow. For most visual sub-languages and diagram types it will be reasonable to extend or to replace the graphics editor, so that the system can support special interaction modes required for convenient editing of *V*-expressions. If, e.g., we are working with circuit diagrams, we will need only little functionality of a general purpose graphics editor, but at least some functionality of a circuit CAD tool will be required instead. The graphics editor is the only part of the system that has to be customized by real programming. Garnet [5] is used as a powerful graphics toolkit so that extensions of the editor are easily implemented. The graphics editor, of course, is totally independent from changes in the textual host language.

4. A framework for HVPLs

As we have pointed out, pictures are separated from the textual part of the program in a first step, then translated into a textual equivalent, and finally reassembled with the textual program. While the separation of textual and visual structure is a task of the standard environment and does not need to be modified for different HVPLs, the parsers used in the second and the third phase must be customized by a visual and a textual grammar, respectively.

4.1 Translation of pictures into picture graphs

As a first step, each picture is translated into its corresponding picture graph (PG). The PG that belongs to a visual expression depends on the picture vocabulary defined for this particular visual sub-language. A picture vocabulary consists of a set of atomic object types (e.g., OT = {*circle*, *rectangle*, *point*, *label*}) and a set of spatial relation types (e.g., RT = {*touches*, *intersects*, *inside*}). A PG is a bipartite graph containing two types of nodes: Object nodes and relation nodes. Let P be a picture consisting of a set of objects O. The corresponding picture graph PG(P) contains exactly one object node for each element in O. The set of relation nodes R and the set of edges E are built in the following way: For each n-tuple (o_1, ..., o_n) of

objects in O and each n-ary relation r in RT test $r(o_1, ..., o_n)$. If it holds, insert a new relation node v into R and insert (undirected) edges between each o_i and v into E labeling each edge by the corresponding argument position i. Then PG(P) = ($O \cup R$, E). Edges that are incident to nodes representing symmetric relations are not labeled. This guarantees that semantically equivalent pictures have isomorphic PGs. All the nodes in a PG are typed by either a relation type or an object type. Furthermore, each node has a list of named attributes which is used during picture translation and is initially set to contain the geometry (*geo*) for graphical objects and, in addition, the displayed text (*val*) for objects of type *label*. Figure 3 shows a picture and its corresponding PG for the picture vocabulary OT = {*circle, arrow, label*} and RT = {*inside, attached, connects*}.[1]

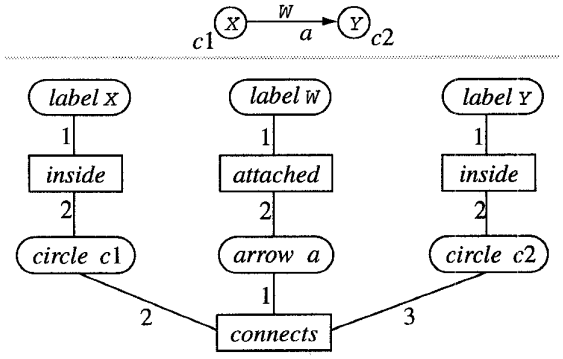


Fig. 3: A picture and its PG

4.2 Translation of PGs into textual structures

The translation of PGs into text is performed by a production system that computes the attributes of PG objects. A translation may consist of several distinct text fragments, each stored in a separate attribute. These attributes finally serve as an interface to the textual parser, which recombines the original textual structure with the translated picture code. Each rule has the form

$$G \Rightarrow g_1, ..., g_m \mid [x_1.attr_1 := f_1, ..., x_n.attr_n := f_n]$$

where G is a PG, $attr_i$ is an attribute name and f_i is a formula that calculates the value for this attribute. The symbol "|" separates the attribute structure from the optional guard conditions g_i that are evaluated before a production is applied. A production is applicable if its left-hand side G matches (i.e., is isomorphic to) a subgraph of the PG being transformed and if all the guard conditions are true in the context of the bindings induced by the current match. G may either be a PG or (as a syntactic shorthand) its corresponding picture. G can be extended by labeling individual picture objects (nodes) with variable names to be used in the formulas on the right-hand side.

1. We use Courier font to distinguish labels from object identifiers.

The term *var.attr*, which can be used in guards and assignments, references the attribute named *attr* of the object labeled by *var*. An example of a rule to translate inner nodes of binary tree pictures is:[2]

$\Rightarrow$ *left-of*(*y*, *z*) | [*x.code* := `branch(`*y.code*`,` *z.code*`)`]

and one of the rules used for translating the PG of the introductory example is

$\Rightarrow$ [`W.code := trans(X.val, Y.val, W.val)`]

Note that the translated code is actually collected in the arc labels and not in the arcs themselves to allow for multiply labeled arcs in state diagrams.

On every evaluation cycle the applicable rule with the highest priority is executed. The priority is implicitly given by ordering productions in the rule file. If a pattern graph matches more than one subgraph, the actual match is selected arbitrarily. Of course, the attribute list and the guard conditions on the right-hand side can only be evaluated if all the attributes used therein have already been bound, either by an earlier rule application or, for *geo* and *val*, during PG extraction. If some guard formula cannot be evaluated because of unbound values, it yields false, and another match of the same rule is tried. If no match satisfies the guard, a rule with lower priority is selected instead. If an attribute formula f_i cannot be evaluated for the same reason, it is delayed, i.e., the rule fires, but it cannot set the attribute $attr_i$ immediately. Thus, the assignment $x_i.attr_i := f_i$ is frozen and woken up again as soon as some other rule application has set the missing value(s). This roughly corresponds to establishing a uni-directional constraint.

Every attribute will be assigned only once,[3] i.e., a rule that would set some attribute which is already bound will no more be applicable. Evaluation stops when no more rules can be applied or when a rule containing the optional *STOP* command is executed. After the evaluation has finished, the attributes of all nodes in the PG are collected and grouped by their names according to the order in which they were set. The resulting attribute collections become the attributes of the picture and serve as the interface structure to the textual parser.

4.3 Merging picture translations into program text

The first step in the compilation of a heterogeneous program is to replace the pictures in the original program by unique textual tokens. The resulting text is analyzed by

2. We are using the Times font for expressions of the rule language and Courier for expressions that are handled as strings, i.e., `branch(...,...)` will actually appear in the attribute code, while *y.code* and *z.code* will be replaced by the corresponding attribute values.

3. An attribute with a frozen assignment is regarded as set.

an ordinary attribute grammar parser. Each time the non-terminal ‹*picture*› is encountered, the parser reads a picture token from the input and looks up the picture translation for the appropriate PG. The attributes computed during its translation become the attributes of this occurrence of ‹*picture*›. Thus, by using standard syntax-oriented translation techniques it is simple to assemble the final translation. Let us have a short look at two examples. The attribute grammar for our introductory example is an extension of Prolog's standard grammar that allows to replace term structures by pictures. The following rules are used, given that the attribute *code* contains the texts derived from a picture:[1]

‹*term*› → ‹*atom*›
  ‹*term*›.*code* := ‹*atom*›.*code*
‹*term*› → ‹*picture*›
  ‹*term*›.*code* := *concat*(‹*picture*›.*code*)
‹*term*› → ‹*atom*› (‹*termlist*›)
  ‹*term*›.*code* := ‹*atom*›.*code* ⊕ (⊕ ‹*termlist*›.*code* ⊕)

The translated code obtained from a picture can as well be distributed among several places. If, e.g., some additional code has to be added at the end of the rule in which the picture occurs, the picture translation may store this code in an additional attribute *trailer*, say. This attribute would be used in

‹*term*› → ‹*picture*›
  ‹*term*›.*code* := *concat*(‹*picture*›.*code*)
  ‹*term*›.*trailer* := *concat*(‹*picture*›.*trailer*)

and would be passed from production to production until finally used in:

‹*body*› → ‹*goallist*› .
  ‹*body*›.*code* := ‹*goallist*›.*code* ⊕ ‹*goallist*›.*trailer* ⊕ .

In future, different types of non-terminals for integrating multiple picture sub-languages will be usable. Each of these non-terminals will be specified by an individual picture vocabulary and a transformation rule set applicable only to this particular picture type.

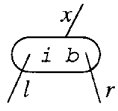
An important detail is that there is no restriction imposed on pictures to only represent data structures. Any code, including control information, can be derived from a picture. Examples and applications of this will be shown in Section 5.3.

5. Examples

We will now present some examples of reasonable size. For the sake of brevity we will not use visual notations adapted from real application domains here because such structures are usually very rich and thus are complex to translate and describe. Instead we focus on well-known examples of data structure manipulation.

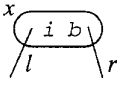
Our first example is the maintenance of AVL trees. Below, the code for insertion into an AVL tree is given in two different language paradigms: Logic programming (Prolog) and functional programming (ML). Both, logic languages and functional languages, make intense use of matching (unification) to analyze and to construct data structures using terms of the respective languages as patterns to be matched. This principle is exploited in embedding the visual sub-language. Every visualization of a tree structure is translated into a textual term, and the picture is locally replaced by this term. Thus, a picture can be put anywhere a term is used.

The visual sub-language of trees defined here can be used in both host languages without any change. It has an amazingly simple definition. Its basic object types are *roundtangles*, *triangles*, *lines*, and *labels*. The only rules required to define and to translate it are:



⇒ *left-of*(*l*, *r*), *left-of*(`i`, `b`) | [*x.out* := `branch(i.val, b.val, l.out, r.out)`]

⇒ [*x.out* := *v.val*]



⇒ *left-of*(*l*, *r*), *left-of*(`i`, `b`) | [*x.code* := `branch(i.val, b.val, l.out, r.out)`] *STOP*

The ordering of the rules insures that the root of a tree picture is matched last. Not only can we use the same visual language for ML and Prolog, but also the extension of both languages' grammars is almost identical. In Prolog it is:

‹*compound-term*› → ‹*picture*›
  ‹*compound-term*›.*code* := *the*(‹*picture*›.*code*)

For ML, we need two rules (for patterns and expressions) obtained by simply exchanging ‹*compound-term*› by ‹*pat*› and ‹*exp*›, respectively. To keep the examples short, we present only the code for insertion into the left sub-tree here. The full program, of course, contains analogous code for the insertion into the right sub-tree. The logic of the program is simple. If a new element is to be inserted into a tree *T*, either a new tree is created (if *T* is empty) or it is recursively inserted into the left (right) subtree of *T* if it is smaller (greater) than the key in *T*'s root. Once it has been inserted, *T* must possibly be rebalanced by rotation or by double-rotation if the depth of the modified subtree has increased. The latter information is passed around as a boolean parameter. The most difficult parts to understand in an AVL program are the rotation and the

1. Note that every picture attribute like ‹*picture*›.*code* is a collection of values. Therefore a function is applied to transform the collection into a linear text structure. Here *concat* is used to transform a collection of strings into a single string. Another function is *the* to extract the single element from a collection.

double-rotation of a tree out of balance. In textbooks these parts of the program are usually explained by graphical examples, see, e.g., [6]. Using an HVPL we can now directly program with these illustrations, and thus the function of a rotation becomes evident at once.

5.1 Logic programming

The heterogeneous Prolog code for the AVL insertion is given in Figure 4. According to the above translation rules, the first rotation, e.g., is mapped into the following standard Prolog code:

```
rebalanceL(true,
  branch(K, -1, branch(A, -1, X, Y), R),
  branch(A, 0, X, branch(K, 0, Y, R)), false).
```

As can be observed in the visual program, the programmer is entirely free to choose visual or textual expressions to handle tree structures. Thus, simple code fragments can be written very dense using textual terms while difficult passages can be programmed with the aid of visual structures.

5.2 Functional languages

Since ML is a strongly-typed language we first need the following type definition:

```
datatype T = branch of real * int * T * T
           | empty
```

This defines a term constructor `empty` of type `T` and a constructor `branch` taking four arguments of the types as defined above and returning a tree object. (Actually, it is not difficult to imagine type definitions also given by pictures.) Now, ML functions for inserting and rebalancing look almost the same as the Prolog program from above. We give a slightly different implementation which combines rebalancing and insertion in a single function. Pictures are used only in displaying complex patterns so that the two rotations can easily be identified, see Figure 5. (Note that expressions like `T as` ⟨*pat*⟩ define `T` as a shorthand for the complex pattern ⟨*pat*⟩.)

The textual ML program corresponding to the heterogeneous one is simply obtained by substituting the tree pictures by terms as it was already shown in the Prolog example.

5.3 Imperative programming

In imperative languages programs containing a lot of pointer manipulations are certainly the most difficult to read. We show how pictures can be used instead of textual pointer redirections to enhance understanding such programs. We consider a C-program for inserting elements

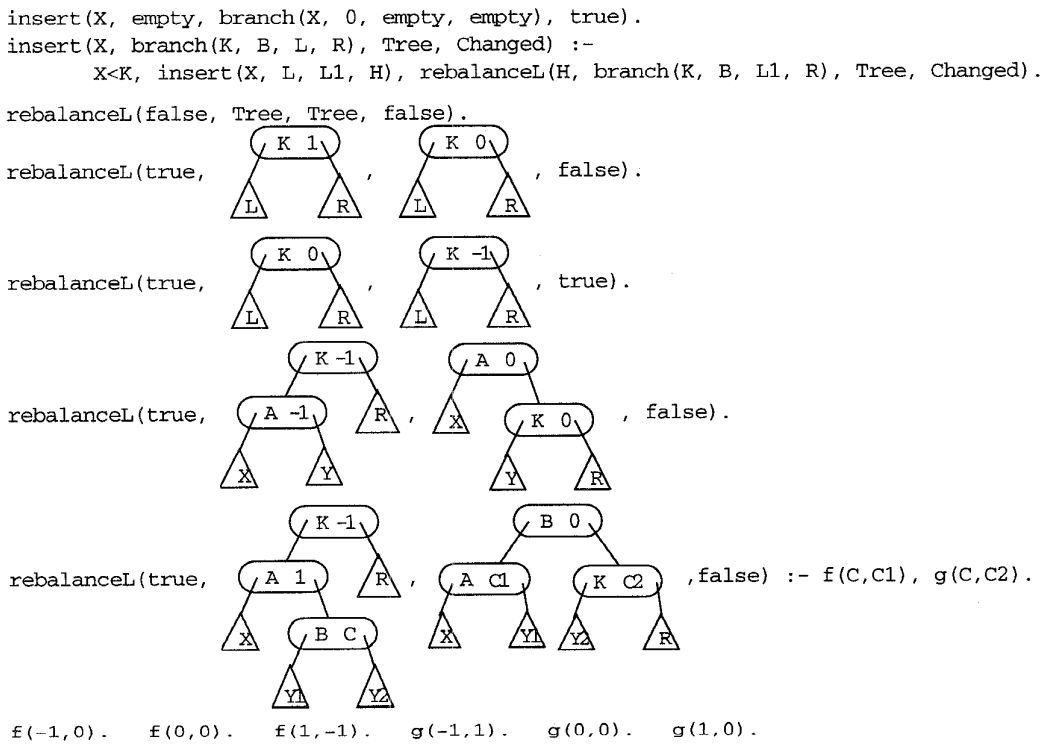
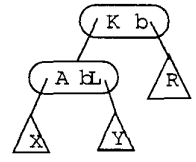


Fig. 4: Heterogeneous Prolog code for AVL tree insertion

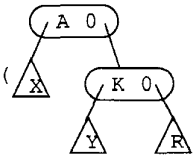
```
fun f(x) = if x=1 then ~1 else 0
fun g(x) = if x=~1 then 1 else 0

fun insert (x, empty) = branch (x, 0, empty, empty)
 | insert (x, branch (K, b, L, R)) =
   if x < K then let
        val (Li, h) = insert (x, L)
```

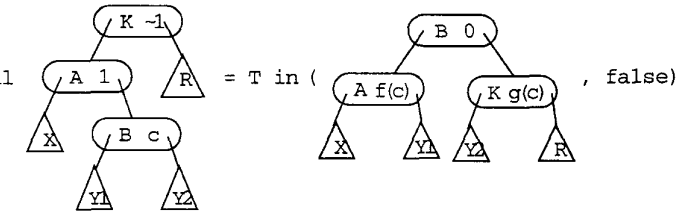



```
        val T as                    = branch (K, b, Li, R) in
```

```
     if h then
        case b of
            1 => (branch (K, 0, Li, R), false)
          | 0 => (branch (K,~1, Li, R), true)
```




```
          | ~1 => if bL = ~1 then (          , false)
                else let
```




```
                end
     else … (* analogous case for x > K *)
```

Fig. 5: Heterogeneous ML program for inserting into an AVL tree

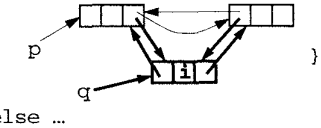
into a doubly-linked list. Given the following list data structure:

```
typedef struct dlist {
    int key;
    struct dlist *prev, *next;
} ELEM, *DLIST;
```

the heterogeneous C-program shown in Figure 6 inserts a new element right after the element pointed to by p.

```
insert(p,i)
DLIST p;
int i;
{
    DLIST q;

    if (p->next != NULL) {
```




```
    }
    else …
}
```

Fig. 6: Heterogeneous C program for list insertion

Thin arrows are interpreted as displaying the current situation in the pointer structure at the time `insert` is called while bold arrows denote the intended pointer redirections.

The production system of Figure 7 defines the translation of pointer pictures into sequences of C assignments. There are actually three rules for setting the *redirect* attribute: This is to insure that those pointer assignments are generated first that use cell addresses which are changed by other redirections. Note that only "local" pointer redirections can be properly handled by this rule system. The difficulty is to generate consistent sequences of assignments in general.

Finally, the integration into C is obtained by simply adding the following rules to the C-grammar:

‹compound-statement› → ‹picture›
‹compound-statement›.code := concat(‹picture›.new) ⊕ concat(‹picture›.upd) ⊕ concat(‹picture›.redirect)

6. Conclusions and further work

We have shown how a smooth integration of visual languages and textual languages can be achieved, and we

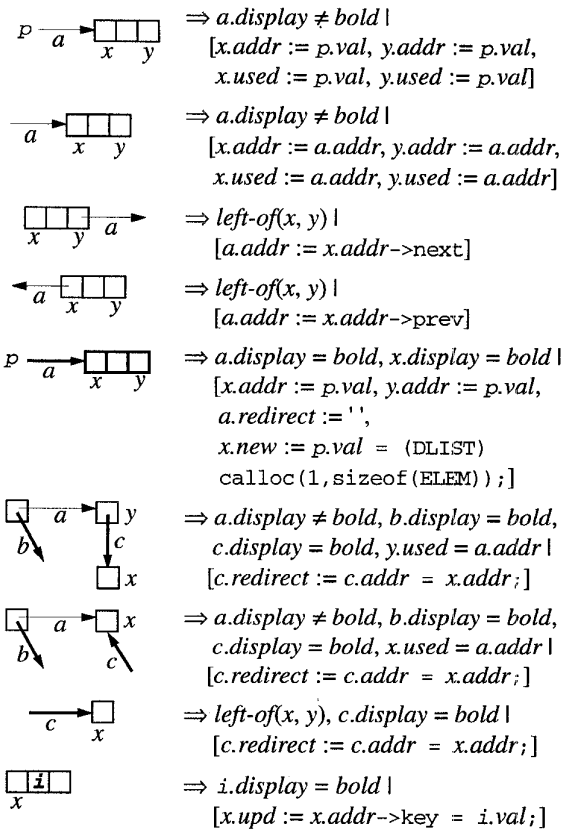


Fig. 7: Rule system for pointer manipulations

have outlined the design of a prototyping environment for working with heterogeneous languages. Looking back at the experiences gathered with purely textual and purely visual languages, making heterogeneous languages available seems to be a reasonable path to improved usability of visual notations. The main benefit of HVPLs is the freedom to choose and work with domain-specific visual notations without having to design and implement entirely new languages each time the visual sub-language is changed. Both, the visual part and the non-visual part of an HVPL, can be reused when the respective other part is changed. Therefore, adaptions to new domains can be accomplished with minimal effort. Another important advantage is that the overall programming paradigm may be chosen independently from the visual sub-language. Thus only minimal learning is required to employ visual programming in practice. The balance between textual and visual expression is not fixed by the language, but is rather left to the programmer. As heterogeneous programs directly translate into code of the corresponding textual host language, there is no runtime performance penalty imposed.

A related approach is pursued in the Andrew project [7]. Andrew is mainly a toolkit for programming multiparadigm environments, whereas we focus on developing a framework and a platform for heterogeneous programming languages along the lines of grammar theory and compiler technology.

Some problems not yet solved remain. The foremost difficulty concerns debugging: When programming with visual structures we would certainly like to use them during debugging, too. This is currently not possible since heterogeneous programs are compiled without any information about the visual structures remaining. Secondly, we cannot yet handle multi-level embeddings of pictures and text (i.e., texts that embed pictures which in turn embed texts which embed pictures that ...). In future we will be working on such extensions of our basic framework.

In some cases the approach of Section 4.2 leads to very complicated rule systems. It is planned to replace the picture translation modules by a more powerful parsing mechanism in future. We regard a combination of picture logic [8] with constraint handling methods like those presented in [9] as a promising candidate. With such a parser the picture translation rules would themselves be a heterogeneous language. Thus, one reason to use a simpler and easily implementable translation mechanism first is to bootstrap the parser implementation.

The prototype system presented here is currently being implemented with Lisp/CLOS and the Garnet graphics toolkit for X11 workstations and Macintosh platforms. Next we will look into diverse application domains to gain a deeper understanding of their requirements. We believe HVPLs to be a great improvement over purely textual languages while at the same time avoiding the problems that come along with a migration to purely visual languages.

7. References

[1] J. Kodosky, J. MacCrisken, and G. Rymar. Visual Programming Using Structured Data Flow. *IEEE Workshop on Visual Languages*, Kobe, Japan, pp. 34–39, 1991.

[2] R. Yeung. MPL–A Graphical Programming Environment for Matrix Processing Based on Logic and Constraints. *IEEE Workshop on Visual Languages*, Pittsburgh/PA, pp. 137–143, 1988.

[3] T.B. Brown and T.D. Kimura. Completeness of a Visual Computation Model. In *Software Concepts and Tools* 15 (1994): 34–48.

[4] J. Barwise. Heterogeneous Reasoning. *Working Papers on Diagrams and Logic*, G. Allwein and J. Barwise (Eds.), Indiana University, Bloomington, pp. 1–13, 1993.

[5] B.A. Myers, D.A. Giuse, R.B. Dannenberg et al. Garnet; Comprehensive Support for Graphical, Highly-Interactive User Interfaces. In *IEEE Computer* 23, No. 11 (1990):71–85.

[6] J. Nievergelt and K.H. Hinrichs. *Algorithms & Data Structures*. Prentice-Hall, Englewood Cliffs/NJ, 1993.

[7] W.J. Hansen. Andrew as a Multiparadigm Environment for Visual Languages. *IEEE Symp. on Visual Languages*, Bergen/ Norway, pp. 256–260, 1993.

[8] B. Meyer. *Visual Logic Languages for Spatial Information Handling*. Doctoral Thesis (in German), FernUniversität Hagen, 1994.

[9] R. Helm and K. Marriott. A Declaratice Specification and Semantics for Visual Languages. *Journal of Visual Languages and Computing* 2 (1991):311-331.