UNIVERSITY OF COLORADO, BOULDER

Department of Computer Science

The Enhancement of Understanding through Visual Representations

Heinz-Dieter Boecker, Gerhard Fischer and Helga Nieper

Department of Computer Science
and
Institute of Cognitive Science
University of Colorado, Boulder

In the Proceedings of the CHI'86 Conference
on Human Factors in Computing Systems
April 13-17, 1986, Boston

Abstract

It has been argued for a long time that the representation of a problem is of crucial importance to understanding and solving it. Equally accepted is the fact that the human visual system is a powerful system to be used in information processing tasks. However, there exist few systems which try to take advantage of these insights. We have constructed a variety of system components which automatically generate graphical representations of complex structures. We are pursuing the long-range goal of constructing a *software oscilloscope* which makes the invisible visible. Our tools are used in a variety of contexts: in programming environments, in intelligent tutoring systems, and in human-computer interaction in general by offering aesthetically pleasing interfaces.

ECOT 7-7 Engineering Center • Campus Box 430 • Boulder, Colorado 80309 • (303) 492-1502

1. Introduction

The way a problem is represented has a strong influence on whether we can understand and solve it. Simon [Simon 81] argues that solving a problem simply means representing it so as to make the solution transparent. He argues that this is especially true for mathematics which exhibits in its conclusions only what is already implicit in its premises; mathematical derivations can be viewed as changes in representation, making evident what was previously true but obscure.

Today the technological base (e.g. raster screens, mice) exists for using computer screens as a truly two-dimensional medium for the representation of arbitrary information structures which can be manipulated directly with a pointing device. Despite the existence of these technological possibilities most systems still use the screen as a "glass-teletype".

Believing in the general hypothesis that the limits of our thoughts are all too often identical with the limits of our imagination and visualization capabilities, we have developed over the last several years a rich variety of visualization tools to make using the computer a more rewarding and less error-prone experience. These tools have been used successfully to enhance learning, understanding and debugging of complex artifacts.

2. Insight and Understanding versus Verification

There exist two different views in computer science about its crucial issues which are summarized in Fig. 2-1 [Newell, Simon 76; Fischer, Boecker 83].

	View 1	**View 2**
Computer Science is:	a formal, mathematical discipline	an experimental discipline
Main tools for the development of systems are:	formal specification techniques	rapid prototyping / experimental programming
Basic Challenge:	think more clearly	better tools are needed (because humans have a bounded rationality)
Programming Methodology:	do not write programs which cannot be verified before they are written	design is an error correcting process

Figure 2-1: Two Opposing Views about the Crucial Issues in Computer Science

It obviously depends on the application area which view proves to be more appropriate. For all the areas which we have been interested in (e.g. Artificial Intelligence, Cognitive Science, Human-Computer Communication, Use of Computers for Learning and Instruction), there is no doubt that the second view in Fig. 2-1 gives the adequate characterization. Based on this perspective we have constructed the tools described in sections 3 and 4.

2.1 Examples

Over the last several years we have carried out informal experiments to understand the impact of graphical representations to generate insight and understanding in problem solving:

1. **The Rope Around the Earth** (see [Fischer 79]): *A rope is tied around the earth at the equator (assuming the surface of the earth is smooth). If we extend this rope by one yard and form a concentric circle around the earth, will the difference between the earth and the rope be big enough that a mouse can get through?* Almost everybody's intuitive answer to this problem is "no". It can be easily proven using simple mathematics that the resulting difference is *independent* of the size of the surrounded object and definitely big enough that a mouse can get through. Based on the counter-intuitive nature of the problem this proof *verifies* the result but it provides *no insight and understanding.* The problem remains: How do we make people believe the proof, i.e. understand the solution. We cannot give a detailed description here of what kinds of models may be appropriate to provide this understanding and insight. One possibility is to consider the following situation: A rope is lying on the ground between Boulder and Denver and we lift it up by ten inches; can we do this without increasing the length of the rope? This thought experiment indicates the relationship between radius and curvature.

2. **Number Scrabble** vs. **Tic-Tac-Toe** (see [Simon 81], page 151) are two isomorphic versions of the same game. It can be shown that subjects perform much better in Tic-Tac-Toe than in Number Scrabble. We assume that the rules of Tic-Tac-Toe are known. The rules of Number Scrabble are: it is played by two people with nine cards face up (e.g. the ace through the nine of a card game). The players draw cards alternately. The player who can first make up "15" with exactly three cards, will win. Representing the board for Tic-Tac-Toe as a magic square shows the isomorphism between the two games. One interpretation of the data would be that the more visually-oriented version of Tic-Tac-Toe is easier to play.

3. **The Design of a Roulette Table**: Teaching high school students problem solving with LOGO, we asked them to work on the following problem: *simulate a roulette table with slots 0 to 18.* Given was a random number generator which returned a number between 0 and 9. Most students came up with the solution "*sum* of *random* and *random*" which they felt quite happy with. Until they plotted the results in a graph they did not find out that their roulette table did not give a uniform distribution. Even though the visual representation of the results did not show to them how to produce a fair roulette table, it uncovered the incorrect solution in an obvious way.

Our experiments with these and similar problems suggest that the right kind of representation can provide insight and understanding for a problem. One of the advantages of visual representations over their formal, propositional counterparts relates to the difference between *observation* and *deduction*. In most situations, the former can be accomplished more cheaply in terms of the computations involved, and visual representations facilitate observation since important properties are *directly* observable.

2.2 The Role of Visualization in Software Design

Our goal is to build software components which allow us to take advantage of the power of the human visual system to provide insight and understanding instead of relying only on verification methods. Being interested especially in ill-structured problems [Fischer, Schneider 84], we have found (like all other researchers investigating design problems empirically) that the recommendation "think more clearly" is not good enough; overwhelming evidence shows that there is an urgent need for better tools because humans have a bounded rationality working on complex problems. A verification system [deMillo, Lipton, Perlis 79] which ends up with either the result "correct" or "incorrect" contributes little to an understanding of a problem. In addition, verification procedures require that we can start with an existing specification of a problem (which is then compared with the implementation) whereas the crucial activity in solving ill-structured problems is exactly to generate this specification.

The great potential of a computer system is that multiple representations can be generated *automatically* (i.e. without requiring the user to do much additional work) and *dynamically* (i.e. taking the actual work of someone into account). In the following sections we will present first steps towards a *Software Oscilloscope* which contributes to the goal of generating understanding and insight into the behavior of complex artifacts. The components of the proposed *Software Oscilloscope* all share the property of being of graphical nature, thus exploiting the powerful human visual system. One of the first systems that was designed with this principle in mind was the PYGMALION system [Smith 77]. A good survey of the current state of the art is contained in [Computer 85].

3. Visualization of Data Structures

3.1 Static Aspects: The KAESTLE Editor

The most important data structure of Lisp[1] is the *list*. With KAESTLE [Boecker, Nieper 85] the graphical representation of a list structure is generated automatically and can be edited directly with a pointing device. By editing we do not only mean changing the structure itself but rearranging the graphical layout as well. KAESTLE is integrated in a window system, the user interface is menu-based (see Fig. 3-1).[2]

KAESTLE is a valuable tool for the Lisp beginner to understand certain aspects of the programming language which are difficult to explain otherwise (e.g. the difference between copying and destructive functions: Fig. 3-2 illustrates the effects of **append** and **nconc**; the normal textual representation displayed in the "toplevel" window reveals no difference between the results of these two functions).

More experienced Lisp programmers use it heavily to display and explore data structures which are difficult to represent symbolically, namely circular and reentrant structures (see "kaestle-window-1" in Fig. 3-1). KAESTLE, as part of a programming environment, can be used for designing, debugging and documenting Lisp programs.

Planning the Spatial Layout of List Structures. To generate a graphic representation of a list structure it is necessary to find a place on the screen for each element belonging to the structure (for more details see [Boecker, Nieper 85]). A *fully* automated layout planning algorithm faces the following problems:

- *List structures may be complex networks.* A very good but time consuming planning algorithm is not helpful in an interactive system.
- *The available space is often insufficient for displaying the entire structure.* How can it be decided which parts of the structure to omit?
- *The semantics of the list structure (i.e. its logical structure) should be taken into account.*

Therefore our basic approach is building *cooperative, symbiotic* systems [Fischer 83] for the human and the computer. In KAESTLE the computer uses a simple planning algorithm: It doesn't pay attention to arrows crossing boxes or other arrows, and the display is truncated at the right and lower border of the display area. After this "first draft" of the graphical layout is generated by the computer, a sophisticated user interface allows the user to move or delete parts of the display or to display additional substructures. Thus the user can decide which parts of the structure he or she wants to see and eventually arrive at a "nice" output by moving around parts of the display.

---

[1] Lisp is the implementation language of all described systems.

[2] All the supporting software tools used in our systems (e.g. the window and menu system, the knowledge representation system ObjTalk [Rathke, Lemke 85]) were developed in our research group. The systems run on a BITGRAPH raster display terminal.

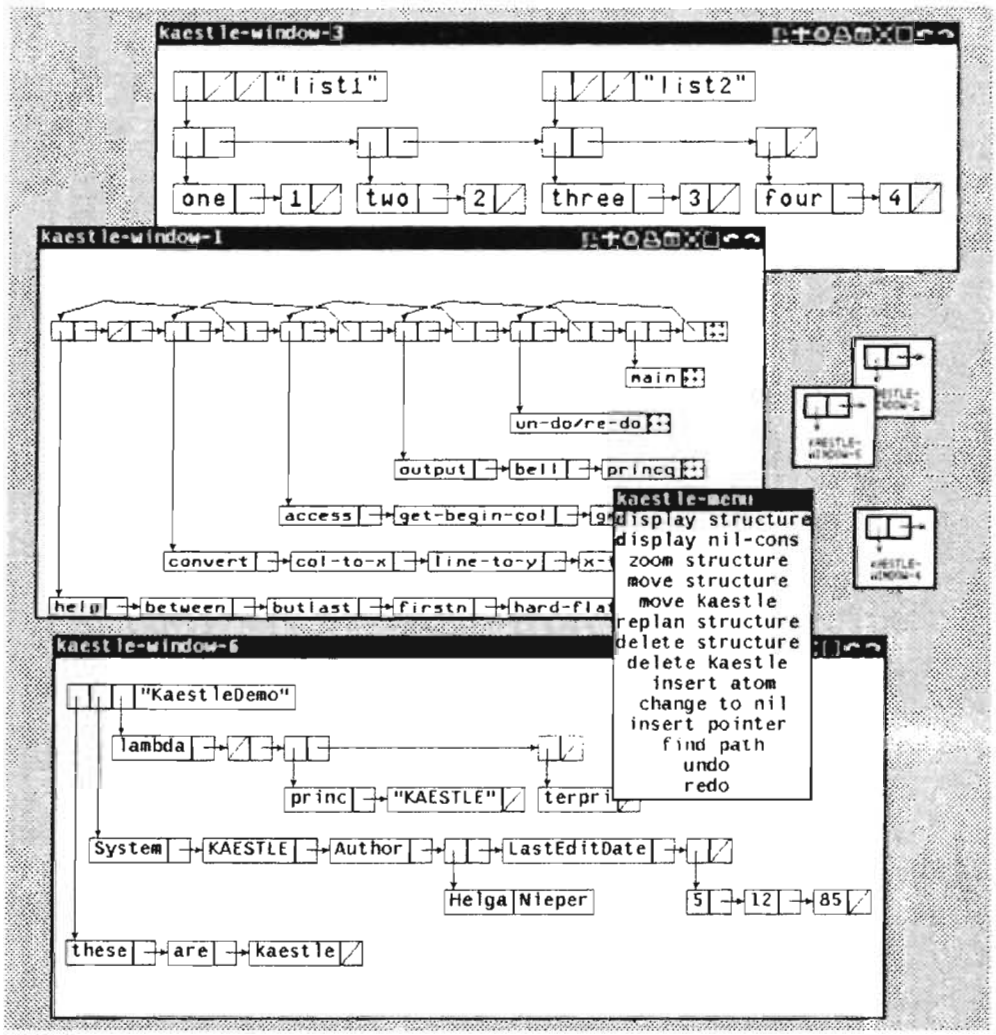

Figure 3-1: KAESTLE: Integrated in a Window System

Functionality of the System. The following operations are available in KAESTLE, among others:

1. *Generating a graphic display:* Multiple independent structures may be displayed at a desired position.
2. *Changing the graphic representation:* displaying substructures which are truncated in the current display, deleting substructures from the display, moving substructures on the screen.
3. *Changing the structure:* The manipulation of the graphic representation (insertion of atoms or pointers) immediately changes the underlying structure.
4. *General undo and redo mechanisms.*

The user can access this functionality by selecting actions from context-sensitive menus which become visible when they are needed.

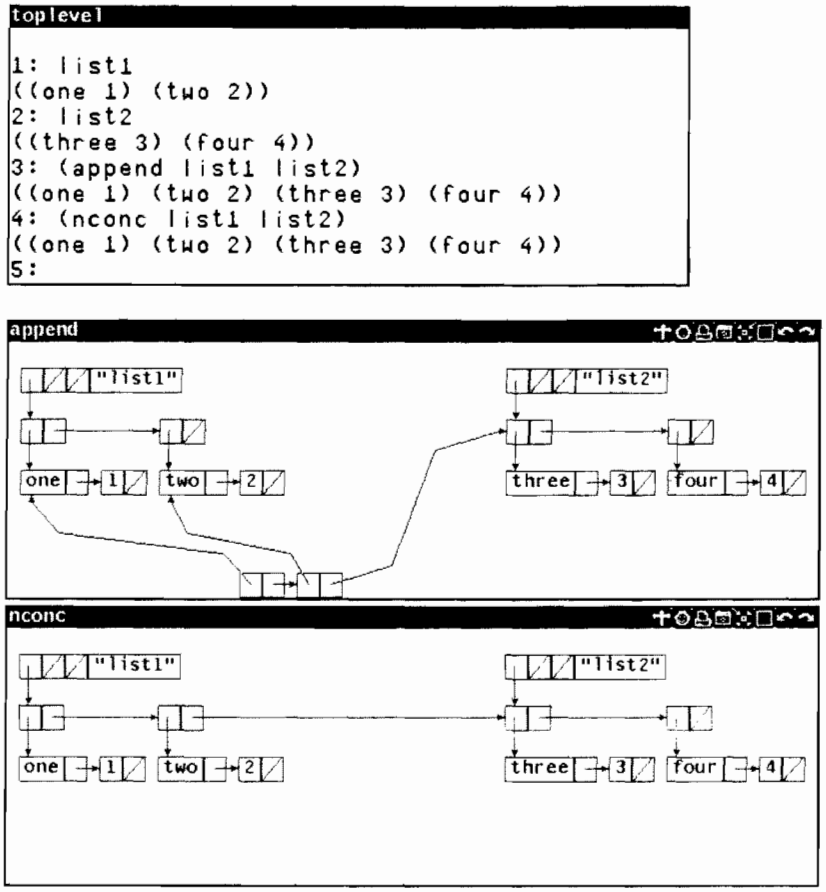

Figure 3-2: The Difference between Copying **append** and Destructive nconc

3.2 Dynamic Aspects: What Happens to the Structures?

KAESTLE may not only be used to display (and edit) static structures but as a *monitoring tool* for running programs. The standard FranzLisp trace package can be used for this purpose by updating the contents of KAESTLE-windows whenever an "interesting" function is entered (`traceenter`) or left (`traceexit`). The trace facility can also be used to generate a sequence of *snapshots* of a data structure while running the program (see Fig. 3-3 which illustrates a recursive, destructive algorithm which reverses a list).

4. Visualization of Control Structures

4.1 Static Aspects: What is the Structure of the Program?

A program composed of a large set of usually rather simple functions may be appropriately described as a *network of functions* which mutually call each other. A graphical representation of such a network is displayed in Fig. 4-1. FooScape displays functions as ellipses that are connected by arrows. The tool is primarily meant to get a first, rough overview over some piece of software. It is especially useful for languages that do not allow a lexical nesting of function definitions.

The planning of the layout (placement of the ellipses and arrows) is done automatically (see [Boecker, Nieper 85] for more details). However, because the "beauty" of the solution sometimes is not acceptable

```
(def my-nreverse
  (lambda (l)
    (cond ((dtpr (cdr l))
           (prog1 (my-nreverse (cdr l))
                  (rplacd (cdr l) l)
                  (rplacd l nil)))
          (t l))))
```

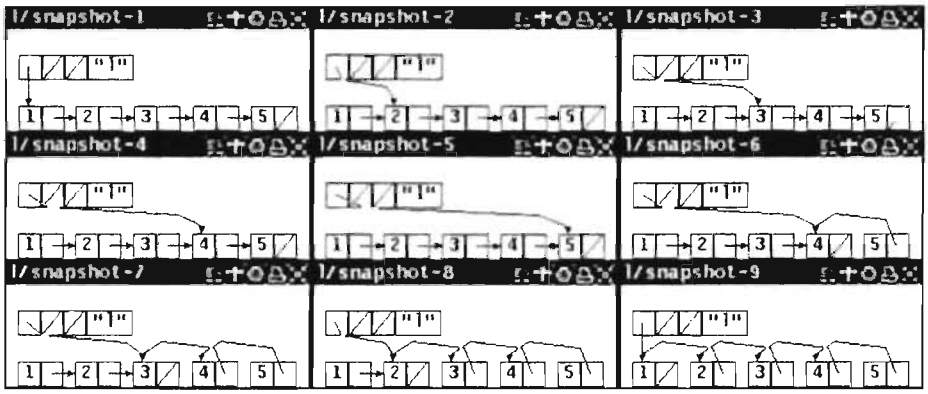

Figure 3-3: Generating a Sequence of Snapshots of a Data Structure

we allow the user to modify the layout interactively by moving "bubbles" around or by altering the set of functions included in the display. The interaction style is similar to the one described in connection with the KAESTLE system.

4.2 Dynamic Aspects: What is the Program Doing?

Traditional techniques for monitoring the dynamic behavior of programs (e.g. breakpoints, dumps) suffer from the fact that they capture just one state of the data and too often generate a huge amount of data. The inspection tools described here try to avoid these disadvantages while on the other hand trying to preserve the dynamics of the processes they look at. Being able to see a program run gives one a grasp of detail that is hard to obtain in any other way.

FooScapes can be readily extended to display the dynamic behavior of programs. The basic mechanism for accomplishing this is provided by the standard FranzLisp trace package. Fig. 4-2 shows two snapshots of an *animated* FooScape. A function name is highlighted - i.e. flips from white to black - whenever the function is active.

The impression given by a "running" FooScape bears some resemblance to the control panels of (outdated) computer systems: You can tell from the pattern of lights that are switched on what the system is doing. Recently *sound* has been added to the FooScape tool: Each of the functions is assigned two specific tones that are played when a function is entered and left, respectively. Preliminary experience with this experimental version seems to confirm the hypothesis that the human audio system is even more capable to monitor sequences over time than the human visual system: *As long as the program plays this Bach style music everything is ok.*

The usefulness of the tool depends on its appropriate use: The programmer has to exercise care in selecting the functions to be included in the FooScape. If the granularity is too fine (the functions included are too primitive) nothing but a flickering screen will be seen whereas hardly any dynamic behavior can be

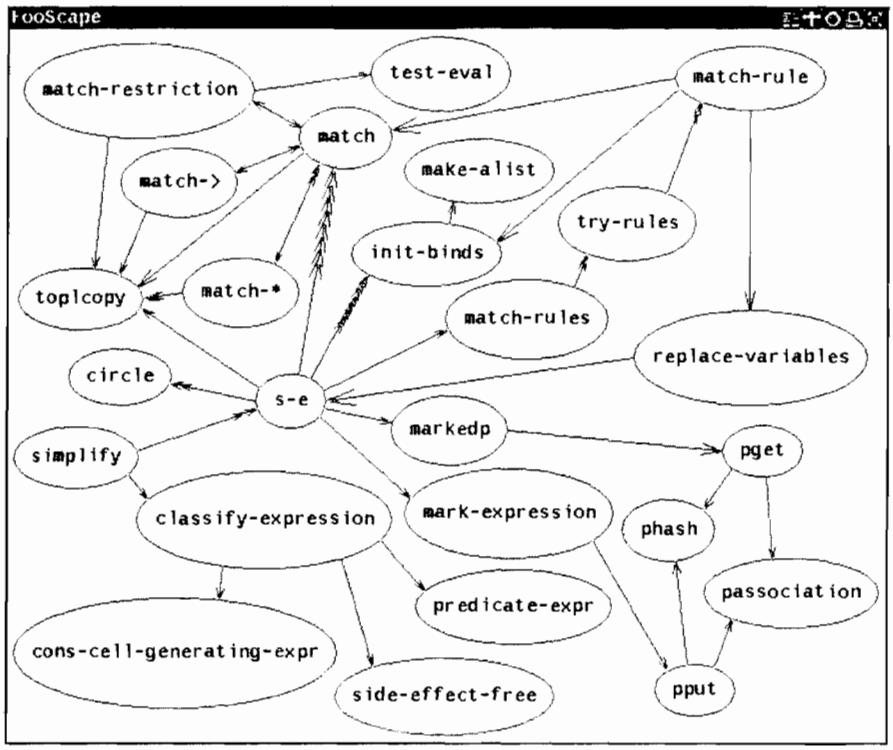

Figure 4-1: FooScape: A Landscape of Functions

observed if the granularity is too coarse. To give the user control over the granularity we have provided some means to exclude functions temporarily from being traced (see the functions shaded with a gray raster-pattern in Fig. 4-2). Experience shows that the selection of the right level of detail requires some familiarity with the tool.

5. Experiences with our Visualization Tools

The tools described have been used for almost two years by a large group of researchers and students on a regular basis. This use has triggered new ideas about additional tools of the same kind and applying them as building blocks in larger applications. For a widespread use it is of critical importance that these tools are tightly integrated and easily accessible within the general programming environment. Nothing is a better indication of the usefulness of a tool than that people start using it, without being forced to use it (e.g. in an employment situation) or without being asked to use it (e.g. in a psychological experiment). We will briefly describe two domains in which the tools were used successfully: programming environments and user support systems.

5.1 Programming Environments

Currently our visualization tools are primarily used as integrated components of a window-based programming environment. The case study below shows how the KAESTLE system is used to debug a Lisp function.

KAESTLE allows tinkering with data structures and supports primitive forms of "programming by example". The user may test algorithms on specific examples having KAESTLE keep a record of what

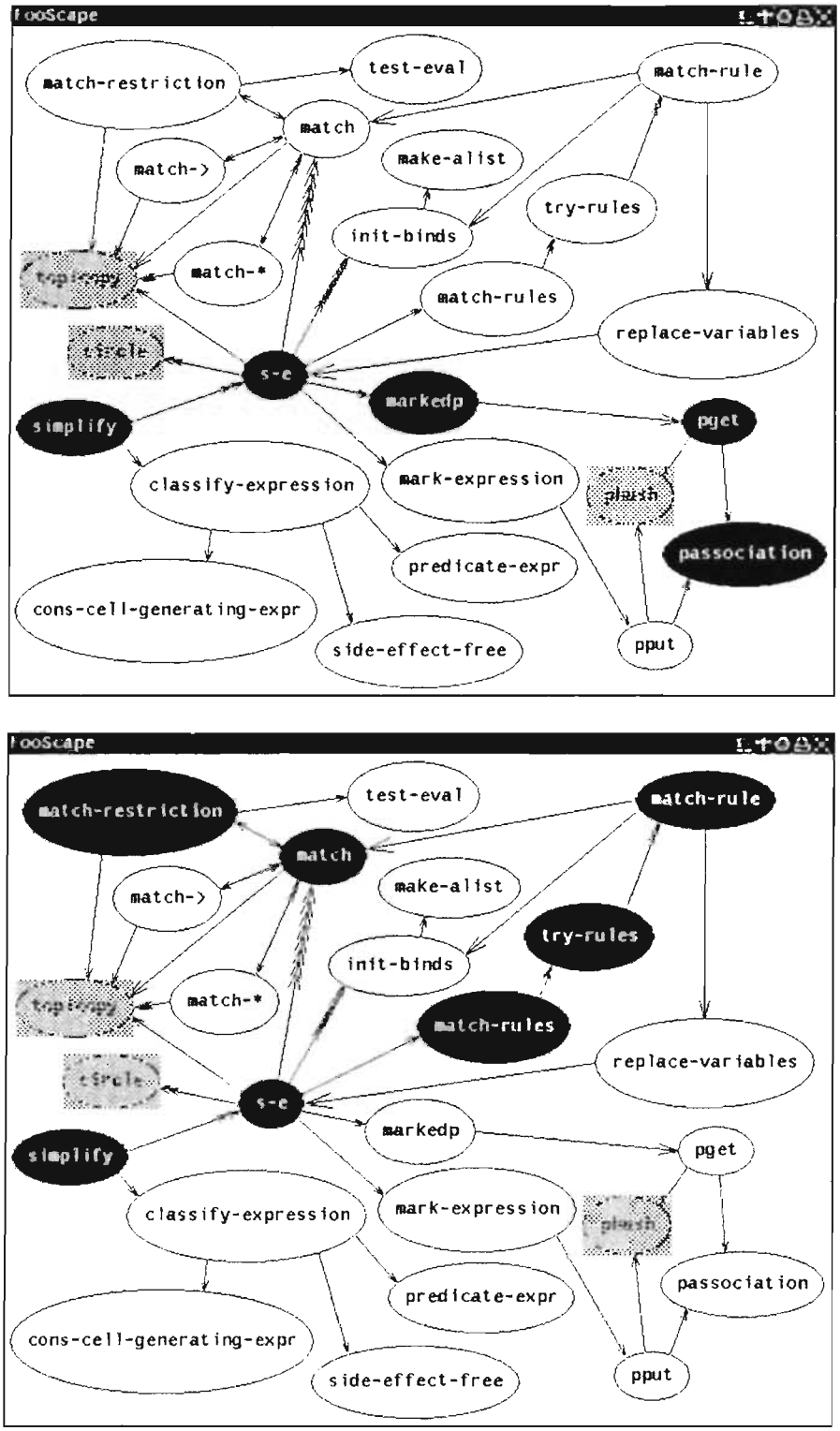

Figure 4-2: FooScape: Showing the Dynamic Behavior of a Program

happens to the data structures. Visualization tools like this reduce the conceptual transformation distance between the symbols and primitives of a Lisp function and the manipulated data structure. They essentially turn a cons-cell into an object which can be easily manipulated.

A Case Study: Self-Organizing Linear Lists. Several techniques are summarized under the rubric of *self-organizing linear lists*. For this case study we will focus on the actual implementation of a specific algorithm that reorganizes a linear list by pulling an element to the front of the list whenever it is accessed. The idea is to exploit statistical properties of the data in order to speed up later operations. We have implemented self-organizing data structures as property lists and association lists in Lisp. Fig. 5-1 shows how the association list `a-list` is reorganized through the `self-org-assq` access function.

```
1: (setq a-list '((Francois . Paris) (Maggie . London) (Helmut . Bonn) (Ronald . Washington)))
((Francois . Paris) (Maggie . London) (Helmut . Bonn) (Ronald . Washington))

2: (self-org-assq 'Helmut a-list)
(Helmut . Bonn)

3: a-list
((Helmut . Bonn) (Francois . Paris) (Maggie . London) (Ronald . Washington))

4: (self-org-assq 'Francois a-list)
(Francois . Paris)

5: a-list
((Francois . Paris) (Helmut . Bonn) (Maggie . London) (Ronald . Washington))
```

Figure 5-1: Self Organizing Linear Lists

When implementing the algorithm to accomplish this behavior one of the authors of the paper had defined a buggy `self-org-assq` access function that for some unknown reason chopped off the last element of the association list when reorganizing it; Fig. 5-2 shows what happened.

```
1: (setq a-list '((Francois . Paris) (Maggie . London) (Helmut . Bonn) (Ronald . Washington)))
((Francois . Paris) (Maggie . London) (Helmut . Bonn) (Ronald . Washington))

2: (self-org-assq 'Helmut a-list)
(Helmut . Bonn)

3: a-list
((Helmut . Bonn) (Francois . Paris) (Maggie . London))
```

Figure 5-2: A Buggy Implementation

The question was: What happened to the last element? Without any visualization tools it would have been necessary to undertake the tedious and error-prone task of debugging that function by essentially running a simulation of it, drawing cons-cells with pencil and paper and making heavy use of an eraser to redirect pointers. With the help of KAESTLE, the bug was easily discovered. Fig. 5-3 shows what happened internally: the cdr of the last cell erroneously was made to point to itself. Having this clue available the code was easy to fix. No other tool of the Lisp programming environment would have been able to provide this quality of support.

5.2 Intelligent User Support Systems

We are in the process of integrating our visualization tools as components of an intelligent user support system (we use the term "intelligent user support system" as a generic word for passive and active help systems [Fischer, Lemke, Schwab 85], documentation systems [Fischer, Schneider 84], explanation systems

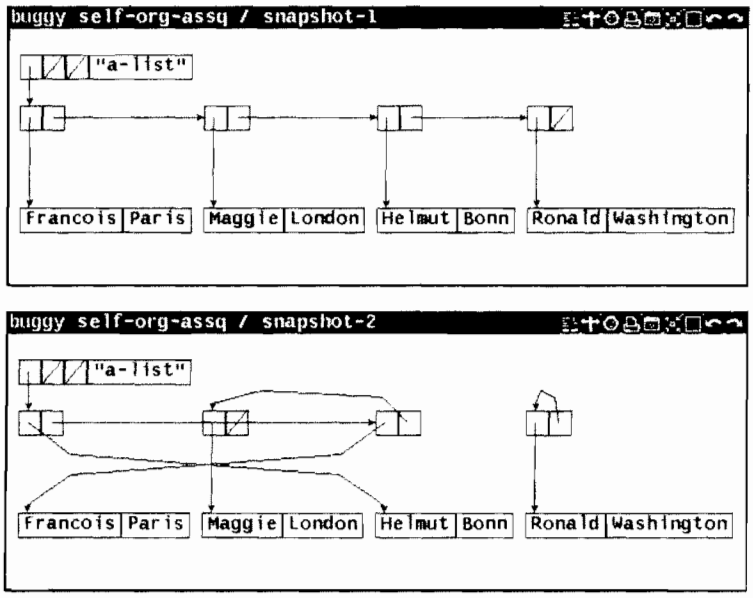

Figure 5-3: How (Ronald . Washington) was Isolated

and advisory systems). It is quite obvious that they may be used to provide *explanations and illustrations* of complex processes to the user or generate explanatory materials within tutorial environments. The important point is that they can do this *on the fly*, i.e. explanations do not have to be precompiled and stored for later use, but instead can be generated dynamically when they are needed, using *actual* data.

We believe that general visualization tools may be used to complement other tools, like video disks, in a natural way. The main advantages of our tools are: The designer of the explanation facilities is freed from foreseeing all conceivable future situations. By integrating these tools with models of the user [Fischer, Lemke, Schwab 85], advice and information will only be given when it is relevant for the actual situation.

5.3 Lessons Learned

One of the most striking lessons learned in implementing the various kinds of visualization tools relates to the automatic planning of graphical layouts of data and control structures. Although sometimes difficult it is usually possible to find algorithms that produce *some* solution in acceptable time. To automatically produce a *pleasing*, aesthetically *nice* layout, however, is a different story. Problems arise because nobody knows exactly what the properties of a *nice* layout are, and the semantics of the structures displayed sometimes require alternative representations that can not be deduced from the syntax of the structures.

Our experiments and implementations also show that programming environments require substantial computational resources when augmented by visualization and monitoring techniques. Personal work stations are necessary to take full advantage of these techniques on a broader scale.

There is a close connection between visualization techniques and "direct manipulation" [Shneiderman 83]. The success of these systems depends on the designer's skill and artistic capabilities in choosing layouts, icons and graphical representations that are natural (i.e. convey the meaning of what they represent) and therefore easily understood. Systems of this sort require expertise in the task domain, but less detailed

knowledge of computers. The world the user has to deal with is explicitly represented. This gives the user the feeling of a more immediate control. We have found that less error-prone interaction results from such an environment because it is not necessary to describe actions (which then get carried out by an interpreter) but we can do them directly.

6. Conclusions

The commercial success of systems taking advantage of rather simple visualization techniques (e.g. spreadsheet programs) indicates that there is a great potential for building more visually based software. Techniques of this sort make computer systems attractive to people which have so far been alienated and scared by their very formal nature and non-transparency. Our experience with the visualization tools described has shown that they are one of the most promising approaches making computers understandable and transparent for all kinds of users. Providing a *software oscilloscope* will be even more important when computers become more intelligent. For example, these tools are necessary for understanding the complex internal processing (e.g. inference processes, inheritance networks) of knowledge-based systems. Without them we are left with "black boxes".

Many interesting problems remain to be solved to further enhance our understanding of the usefulness of visualization techniques as tools for understanding and insight. Not the least of these problems is to build them for a large variety of applications and eventually come up with a toolkit so that they can be easily constructed. In many situations, however, it is not good enough to make the invisible visible [Boecker, Nieper 85]. What is required are techniques that assist the user in making the *relevant* facts and relations visible, e.g. intelligent summarizers and filtering techniques. The techniques which we have developed so far go mostly just in one direction: They generate visual representations of programs. It is an interesting and challenging goal to develop systems which would go in the other direction: Programs would be constructed through "programming by example" [Gould, Finzer 84]. The user would "play" with graphical representations, doing experiments to gain insight into special cases and eventually define (being supported by the system) the general algorithm. This approach may contribute that programming as an intellectual activity will be attractive and achievable by a large group of people.

References

[Boecker, Nieper 85]
H.-D. Boecker, H. Nieper, *Making the Invisible Visible: Tools for Exploratory Programming*, Proceedings of the First Pan Pacific Computer Conference, The Australian Computer Society, Melbourne, Australia, September 1985.

[Computer 85]
IEEE Computer, Vol. 18, No. 8 (ISSN 0018-9162), *Special Issue on Visual Programming*, IEEE Computer Society, August 1985.

[deMillo, Lipton, Perlis 79]
R.A. De Millo, R.J. Lipton, A.J. Perlis, *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, Vol. 22, No. 5, May 1979, pp. 271-280.

[Fischer 79]
G. Fischer, *Multiple Representations*, 1979, MMK-Memo, Institut fuer Informatik, Universitaet Stuttgart.

[Fischer 83]
G. Fischer, *Symbiotic, Knowledge-Based Computer Support Systems*, Automatica, Vol. 19, No. 6, November 1983, pp. 627-637.

[Fischer, Boecker 83]
G. Fischer, H.-D. Boecker, *The Nature of Design Processes and how Computer Systems can Support them*, Integrated Interactive Computing Systems, North Holland, European Conference on Integrated Interactive Computer Systems (ECICS 82), 1983, pp. 73-88.

[Fischer, Lemke, Schwab 85]
G. Fischer, A. Lemke, T. Schwab, *Knowledge-Based Help Systems*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco), ACM, New York, April 1985, pp. 161-167.

[Fischer, Schneider 84]
G. Fischer, M. Schneider, *Computer-Supported Program Documentation Systems*, Proceedings of INTERACT '84, IFIP Conference on Human-Computer Interaction, IFIP, London, September 1984.

[Gould, Finzer 84]
L. Gould, W. Finzer, *Programming by Rehearsal*, Technical Report SCL-84-1, Xerox Palo Alto Research Center, May 1984.

[Newell, Simon 76]
A. Newell, H.A. Simon, *Computer Science as an Empirical Inquiry: Symbols and Search*, CACM, Vol. 19, No. 3, 1976, pp. 113-136.

[Rathke, Lemke 85]
C. Rathke, A.C. Lemke, *ObjTalk Primer*, Technical Report CU-CS-290-85, University of Colorado, Boulder, February 1985.

[Shneiderman 83]
B. Shneiderman, *Direct Manipulation: A Step Beyond Programming Languages*, IEEE Computer, Vol. 16, No. 8, August 1983, pp. 57-69.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1981.

[Smith 77]
D.C. Smith, *Pygmalion, A Computer Program to Model and Stimulate Creative Thought*, Birkhaeuser Verlag, Basel, 1977.