Visualizing Solutions with Viewers

Ulrich Neumerkel, Christoph Rettig, Christian Schallhart

Institut für Computersprachen
Technische Universität Wien
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract

Visualization can be a powerful aid for learning a programming language. It may be used to reinforce central language concepts. In the context of Prolog and CLP-languages, however, most approaches to visualization aim at procedural aspects. Instead of explaining what a relation describes, visualization is used to animate procedural machinery. In this paper we present approaches to visualizing aspects of Prolog programs that try to avoid unnecessary and irritating procedural details. Answer substitutions are visualized with the help of so called viewers. Some procedural aspects are explained with animations. The viewers have been integrated into a side-effect free programming environment and are used in introductory Prolog and CLP courses. The didactical impact of our approaches is discussed.

1 Introduction

Common approaches to program visualization focus on visualizing a program's state and animating its execution mechanism. While program animation is very compelling, this approach may cover the essential declarative and procedural aspects of a program because of the rather complex representations of Prolog's state. In an introductory Prolog programming course at TU-Wien a different path has been chosen for teaching and visualizing Prolog focusing on the declarative side of the language. Two new elements facilitate teaching declarative programming with Prolog. First, a new programming methodology has been developed whose main part consists of several reading techniques that explain specific properties of a Prolog program. In this manner both declarative and procedural aspects are covered without using the commonly used notions of proof trees and execution traces. Second, a side-effect free programming environment helps to avoid many frequently encountered errors, facilitates a specification oriented style of programming and eases communication with the lecturer. In this paper we discuss the integration of visualization devices into the programming environment. Our approach tries to separate the declarative and procedural concerns during visualization. We therefore focus on visualizing particular answer substitutions in a side-effect free manner.

Contents. After presenting the programming environment GUPU in Section 2, the notion of viewers is discussed in Sect. 3. Some problem specific viewers are shown in Sect. 4. The

problem of visualizing Prolog's chronological backtracking is addressed in Section 5. Finally animations of labeling strategies for CLP(FD) programs are discussed.

2 The programming environment GUPU

The programming environment GUPU (Gesprächsunterstützende Programmierübungsumgebung, conversation supporting programming course environment) has been developed to realize Prolog programming courses. GUPU has been used since Spring 1992 for Prolog programming courses at TU-Wien. It has been continually developed since then. The major objective of GUPU was to provide a side-effect free programming environment.

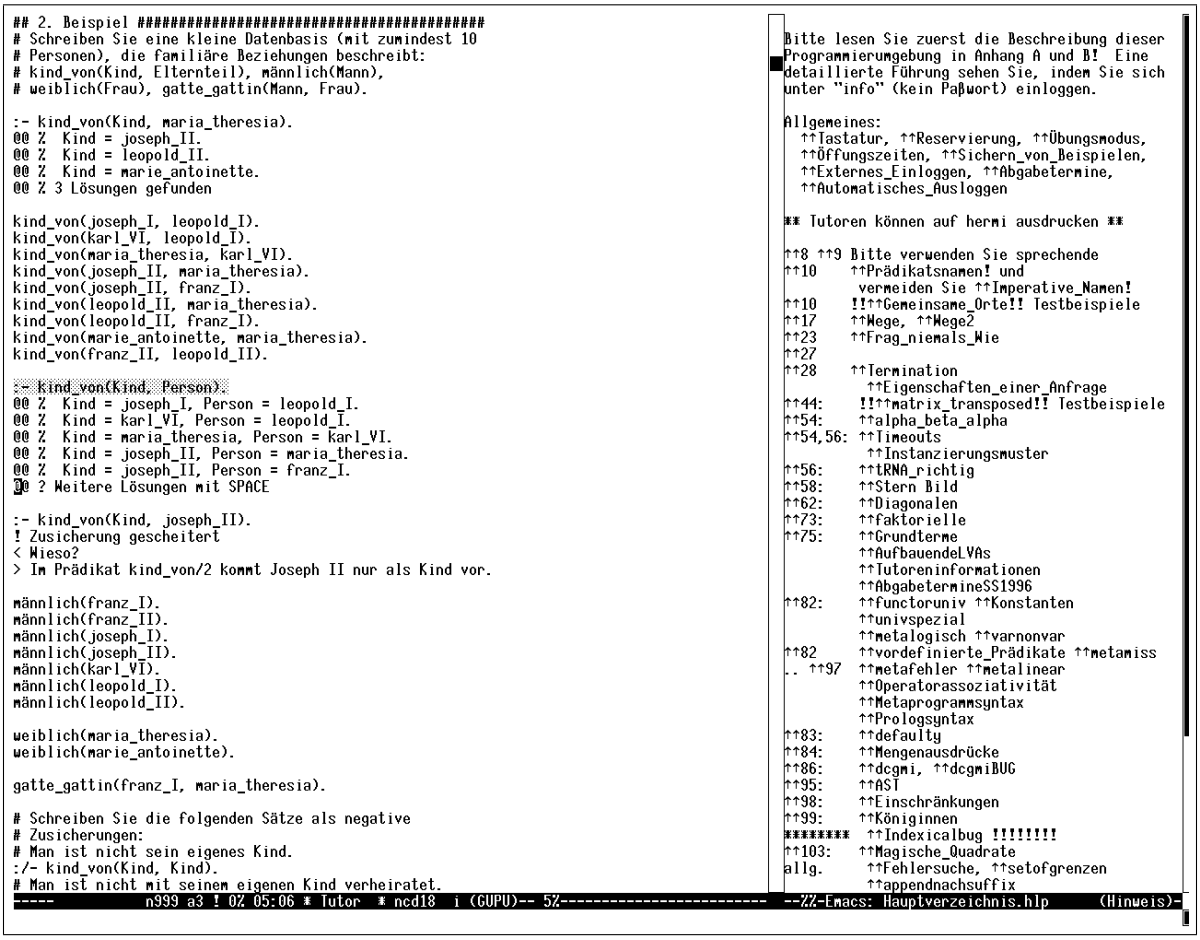


Figure 1: GUPU's screen

GUPU's screen consists of two EMACS-windows (Fig. 1). The left contains examples to be solved. The right contains help (read-only) texts containing links to further texts. All interaction is performed through these two windows. State oriented notions like files or top-level shells are absent. The whole state (i.e. the text) is visible in the left window.

Simple interaction. GUPU's metaphor is very simple: The left window is like a paper sheet containing at first only example statements. The student adds more text and saves it

from time to time. Comments from the system or lecturer are written back *into* the text.

When saving an example the following actions are performed. The example is saved, checked w.r.t. syntactic restrictions, compiled, loaded, and assertions are tested. It is not possible to save an example without seeing the comments from the system on it. In particular all assertions (test goals) are tested. "Last minute changes" that are no more tested cannot happen. Inconsistency between the visible text and the Prolog system is kept to the minimum. At any time only a single example might be modified.

Error messages are inserted into the program text as lines beginning with an exclamation mark. These lines are regular text. As long as they starts with an exclamation mark, they are deleted upon the next saving and are generated anew.

```
← child_of(joseph_II Parent).
! child_of(joseph_II<<*>>Parent).
! Argument list incomplete: A , is missing.
```

Assertions. Lines beginning with an arrow are called assertions. They serve as test cases for a program and allow to write down test cases with ease prior to implementing the actual program. It is therefore possible to specify a relation before its actual implementation. Simple assertions (like ← true.) should succeed within a limited amount of time. Goals that take longer, but still succeed, are annotated with $\overset{\$}{\leftarrow}$. Those that do not succeed because of an infinite loop are marked with $\overset{\infty}{\leftarrow}$. Goals that should fail are annotated with $\not\leftarrow$, $\overset{\$}{\not\leftarrow}$, and $\overset{\infty}{\not\leftarrow}$ respectively. Every time an example is saved assertions are tested. In case an assertion does not behave as specified, an error message is issued.

Interactive queries. The only means to interact with the system beyond the automatic testing upon saving is to ask queries. All assertions can also be used as queries. By clicking on the arrow of the assertion, answer substitutions are displayed. Redundant solutions are labeled separately. The comment ↑↑Redundant below is a link to a help text about redundant solutions.

```
← child_of(joseph_II, Parent).
@@ % Parent = maria_theresia.
@@ % Parent = maria_theresia. % ↑↑Redundant
@@ % Parent = franz_I.
@@ %% 3 solutions found, 1 thereof redundant.
```

Solutions are displayed in chunks of at most five at a time. Also very large sequences —in particular infinite sequences— can be displayed with ease. The traditional Prolog top-level shell requires one to type the two keys [;] [RETURN] for each solution which is somewhat tedious and causes most students to stick with the first solution. But many errors in predicates show up only "on backtracking". These errors are much more difficult to find. Seeing several solutions helps to detect these errors. Similar to error messages the inserted answer substitutions are regular text lines beginning with the @-sign. Upon the next saving they are deleted.

3 Elementary viewers

Answer substitutions are the only way to get some response from pure Prolog. Often this is taken as a pretext to introduce impure features and side-effects. There are however approaches similar to pure functional languages to provide side-effects in a pure logic language. Current state of the art in providing I/O in a pure logic language (like Mercury [1]) restricts nondeterminism in I/O-related predicates. Modes and other procedural notions have to be taken into account. This approach is not desirable within the focus of our course. Some less universal primitives have been implemented. There is no direct output, but (the terms of) answer substitutions can be viewed with some viewers. There is no input except for the program text. While this is a serious restriction, our limited interaction allows to integrate complex programs like the partial evaluator Mixtus[4].

```
← text(Cs) ⋘ Query.

← postscript(Cs) ⋘ Query.

← html(Cs) ⋘ Query.
```

To use a viewer for answer substitutions a Goal is annotated with ⋘ as follows: ← Viewer ⋘ Goal. The most elementary viewers display ASCII-text, Postscript, and html. All of them expect a string (a ground list of characters codes) as argument. An additional text window, Postscript viewer, or Web-Browser displays the string accordingly.

With the help of viewers the error prone printing procedures of Prolog are replaced by grammars, reinforcing the notion that grammars can also be used to describe the actual output as indented text or simple "ASCII-graphics".

```
← hpuzzle(Xs).
$⊬ hpuzzle(Xs), false.
hpuzzle(Zs) ←
   hpuzzlerel(Zs),
   labeling_zs([], Zs).

← hpuzzlerel(Xs).
← hpuzzlerel([2,7,3,4,5,1,6]).
⊬ hpuzzlerel([1,3,7,6,2,4,5]).
⊬ hpuzzlerel(Xs), false.
hpuzzlerel(Zs) ←
   Zs = [A,B,C,D,E,F,G],
   domain_zs(1..7,Zs),
   all_different(Zs),
   A+B+C #= H,
   B+D+F #= H,
   E+F+G #= H.
```

Predicate hpuzzle/1 describes the solutions to a simple puzzle problem. In front of each predicate several assertions are given which are tested each time the program is saved. With assertions of the form ⊬ Query, false. it is stated that the particular Query terminates universally.

In most cases the actual solution of a predicate must be converted with an additional predicate into the string representation. Seeing the string as an answer substitution (and therefore a list of integers) is not very useful. If the predicate responsible for text representation is determinate (describing a single string for a particular solution) it can be wrapped into the viewer as follows.

← text(Cs↑solution_text(S, Cs)) ⋘ pred(S).

In this manner only the answer substitutions for S are displayed.

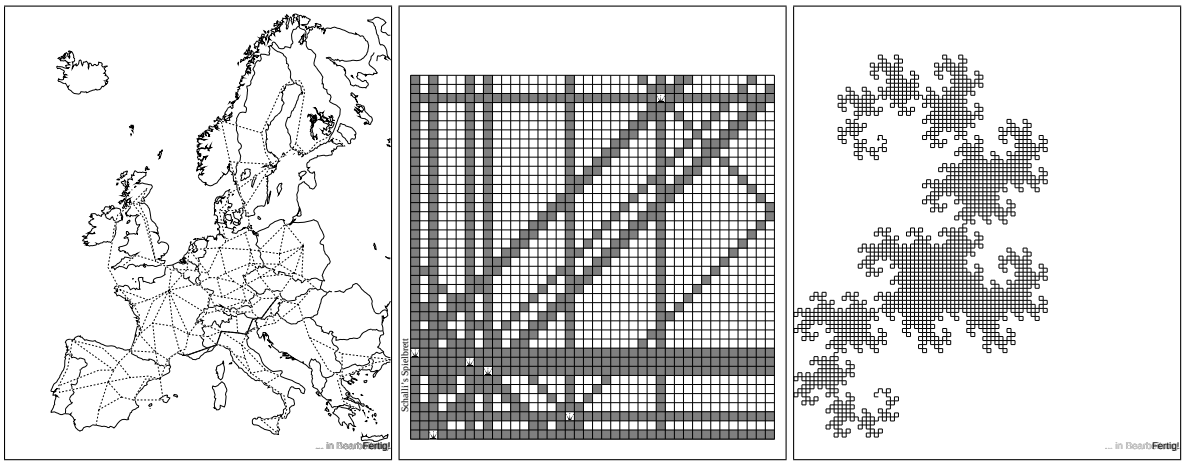
```
← text(Cs) ⋘ hpuzzle(Zs), phrase(hpuzzletext(Zs),Cs).
@@ % Cs = [49,32, ... ,10], Zs = [1,3,7,6,4,2,5].
← text(Cs↑phrase(hpuzzletext(Zs),Cs)) ⋘ hpuzzle(Zs).
@@ % Zs = [1,3,7,6,4,2,5]. % No answer substitution for Cs
hpuzzletext([A,B,C,D,E,F,G]) ⟶
   ( digit(A),          "␣␣␣␣␣",          digit(E), nl ),
   (      "|",          "␣␣␣␣␣",               "|", nl ),
   ( digit(B),     "− −", digit(D), "− −",   digit(F), nl ),
   (      "|",          "␣␣␣␣␣",               "|", nl ),
   ( digit(C),          "␣␣␣␣␣",          digit(G), nl ).
```

```
┌─────┐
│1   4│
│|   |│
│3--6--2
│|   |│
│7   5│
└─────┘
```

4 Problem specific viewers

Based on the elementary viewers more complex viewers are predefined for specific problems. The major advantage of problem specific viewers is the immediate visual feedback they provide about the accurateness of a solution. Each solution displays a corresponding picture.

The viewer ← europa(Path) ⋘ Query. shows the list of cities in a path through the European railway network (left picture). By inspecting the actual railway map it is easy to see whether a path leads directly from one city to another. Queens on a checkerboard are displayed by marking all threatened fields (middle picture). It is therefore obvious whether or not the queens threaten each other. As long as the field of a queen remains white, the queen is not threatened. A sequence of left and right turns of a line is described by a list of ls and rs. E.g., a simple square is thus ← lrs(LRs) ⋘ LRs = "lll". The folding of a paper strip can be described with a grammar, resulting in the well known dragon curve.



5 Visualizing backtracking

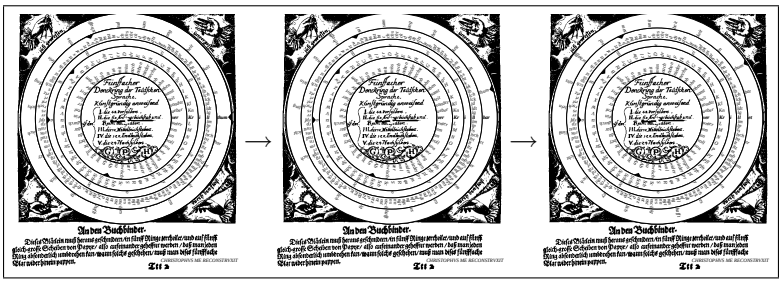
Backtracking is one of the more sensible areas of Prolog. Prolog's actual control mechanism is divided into two orthogonal controls, AND- and OR- control, which are tightly interlaced during execution. Visualizing the actual control mechanism as done by procedural debuggers seems to lead to more confusion than insight. In particular, we have the impression from earlier courses that debuggers increase the unnecessary and incorrect usage of cuts. Instead of avoiding the alternate clause form the beginning (either by shallow cuts or better means) cuts are placed at the point where the failure occurs that leads to the undesirable alternative. It seems that procedural tracers suggest doing this error because right before displaying the undesirable alternative the place where failure occurs is displayed. Such deep cuts often lead to programs lacking steadfastness.

On the other hand understanding the rôle of goal ordering is essential to understanding efficiency considerations. An appropriate metaphor was therefore required which is able to express the idea of chronological backtracking without going into the details of Prolog's execution. In a first attempt clocks and odometers were considered, because they seem to be an obvious metaphor for "chrono-logical" backtracking. However, their visualization did not lead to satisfactory results. On the one hand the examples were difficult to motivate, on the other hand the uniformity of the data (digits) makes places of digits difficult to distinguish.

Georg Phillip Harsdörffer realized a Lullian machine to describe German words [2]. His device consists of five concentric rings with word constituents like prefixes, central letters, and suffixes. By turning the rings all word combinations can be seen; many of which being only remotely related to German words. The constellation on the right shows the inexistent word "verkrillbar". Harsdörffer's device shows in an obvious manner the totality of all possible combinations of word constituents. In contrast to the usual visualizations of alternate solutions via proof trees the complete mechanism is visible. This representation can now be used to explain the systematic enumeration of solutions via Prolog's chronological backtracking.

The possible words of the five rings can be described with the predicate wordconstituents/1. Prolog enumerates solutions to this predicate in chronological order. For every solution the corresponding constellation of the rings is displayed in the viewer. The viewer itself is completely side-effect free. It simply displays a particular solution. However, by inspecting the sequence of solutions, we get the intended coincidental impression that the rings are rotating.

```
wordconstituents([C1,C2,C3,C4,C5]) ←
    ring1(C1),
    ring2(C2),
    ring3(C3),
    ring4(C4),
    ring5(C5).
```



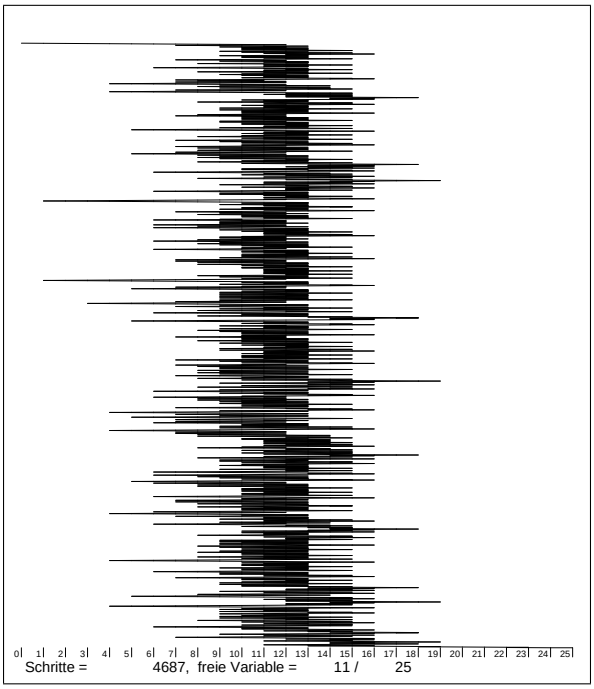
```
← harsdörffer(Xs) ⋘ Xs = [ver,'Kr',i,ll|_], wordconstituents(Xs).
@@ % Xs = [ver,'Kr',i,ll,thum].
@@ % Xs = [ver,'Kr',i,ll,bar].
@@ % Xs = [ver,'Kr',i,ll,et].
@@ ? Further solutions with SPACE
```

The order of the goals in wordconstituents/1 directly affects the way how the rings are "rotating". The ring of the last goal rotates fastest. Whereas the first ring is the slowest one. In this manner it is possible to explain the essential properties of chronological backtracking (and its weaknesses) without going into the details of Prolog's machinery.

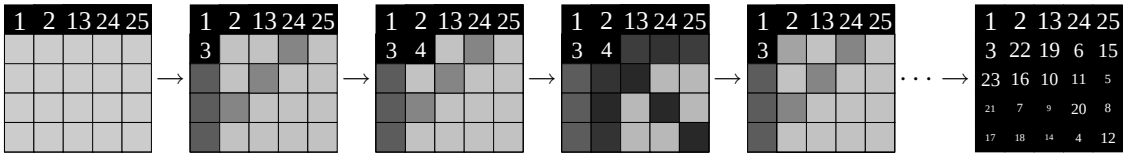
6 Visualizing labeling strategies

The viewers presented so far have visualized a particular answer substitution. For understanding some procedural aspects of constraint programs viewers for animating labeling strategies have been realized.

Most CLP(FD)-programs are divided mainly into two separate parts. In the first part variables, their ranges, and their specific relations are stated with the help of constraints. The second part is used to obtain specific solutions in a nondeterminate manner by using labeling predicates. This structure separated the declarative and procedural aspects. While the first part is responsible for the correctness of solutions, the second part is concerned with effectively finding them by using a particular labeling strategy. The major concern for labeling is therefore to understand the effectiveness of a particular strategy. Details of the actual labeling process are mainly independent from correctness concerns. The simple generic viewer on the right displays the number of variables bound during labeling.



Generic viewers do not represent the actual labeling process very well. Therefore problem specific viewers are provided. For magic squares the process of labeling is illustrated as follows. The darker a field the fewer values are possible for a variable. A black field contains an actual number. The font size of the number represents the time when that variable was bound. Newer (more volatile) bindings are thus represented by smaller numbers. A naïve left-to-right labeling produces the following steps[1]:



Implementation. Labeling animations are realized similar to the approach proposed in [5]. Blocking implication is realized with SICStus Prolog's[3] blocked goals. The number of possible values for a variable is obtained with reïfied constraints, that are connected to a side-effect producing blocked predicate. For each possible value of a variable a separate reïfied constraint is used.

```
... ←
    ...,
    E #\= N0 #⇔ B,
    show_element_i_j_n(B, E, I,J, N),
    ... .
```

[1] http://www.complang.tuwien.ac.at/ulrich/gupu/io-magic.html contains runnable Postscript animations

Our implementation of animated constraint solving is independent both of the internal representations of the constraint system as well as the actual labeling predicates. However, the disadvantages of our approach are very high memory requirements and imprecise visualization. The number of auxiliary constraints required is proportional to the sum of all variable ranges. In the case of magic squares each variable on the $n \times n$ square has n^2 possible values. Therefore at least n^4 reïfied constraints and blocked goals are required. Further, our animations are only an approximation of the actual constraint solving process, because actual constraints and our auxiliary constraints for animation are scheduled in an implementation dependent manner. Some unsuccessful propagations may therefore not be displayed at all.

7 Conclusion

In this paper we have presented approaches to visualize aspects of Prolog and CLP(FD) programs avoiding unnecessary detail of the actual procedural machinery. With the help of a new methodology and programming environment GUPU many of the distractions and perplexions beginners face when exposed to Prolog are avoided but the deeper task — understanding declarative programming — still remains a challenge. In fact, most students suffer from expectation failure. The usual approach to programming simply does not work with Prolog. Understanding all the details of Prolog's execution mechanism is not helpful at all. Unlearing these common habits, appreciating a more specification driven approach to programming is still a difficult task that can be supported with the help of appropriate visual metaphors.

Acknowledgements. The contribution of the tutors Markus Schordan, Wolfgang Faber, Niko Neufeld, Walter Binder, and others has been very valuable. The picture of Harsdörffer's machine is from text machines http://berlin.icf.de/~inscape/.

References

[1] Z. Somogyi, F. Henderson, Th. Conway, The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. JLP 29(1-3): 17-64 (1996).

[2] G. Ph. Harsdörffer, Philosophische und Mathematische Erquickungsstunden, Nürnberg (1651), reprinted Frankfurt/Main (1990). See http://berlin.icf.de/~inscape/.

[3] Intelligent Systems Laboratory, SICStus Prolog User's Manual, Swedish Institute of Computer Science (1997).

[4] D. Sahlin, The Mixtus Approach to Automatic Partial Evaluation of Full Prolog, NACLP (1990).

[5] P. Van Hentenryck, V. Saraswat, Y. Deville, Design, Implementation, and Evaluations of the Constraint Language cc(FD), in A. Podelski, Constraint Programming: Basics and Trends, LNCS 910, (1994).