An Approach to Effortless Construction of Program Animations

J. Ángel Velázquez-Iturbide[*a], Cristóbal Pareja-Flores[b], Jaime Urquiza-Fuentes[a]
[a] Departamento de Lenguajes y Sistemas Informáticos, Universidad Rey Juan Carlos, C/Tulipán s/n, 28933 Móstoles, Madrid, Spain
[b] Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Avda. Puerta de Hierro s/n, 28040 Madrid, Spain

ABSTRACT

Program animation systems have not been as widely adopted by computer science educators as we might expect from the firm belief that they can help in enhancing computer science education. One of the most notable obstacles to their adoption is the considerable effort that the production of program animations represents for the instructor. We present here an approach to reduce such a workload based on the automatic generation of visualizations and animations. The user may customize them in a user-friendly way to construct more expressive program animations. These operations are carried out by means of a user-friendly manipulation based on the metaphor of office documents. We have applied this approach to the functional paradigm by extending the WinHIPE programming environment. Finally, we report on the successful results of an evaluation performed to measure its ease of use.

Keywords

Authoring tools and methods, evaluation of CAL systems, improving classroom teaching, interactive learning environments, programming and programming languages.

1. INTRODUCTION

Learning environments for computer programming often use visualizations and animations as alternative representations to written programs (Stasko, Domingue, Brown & Price, 1998). In general, multiple external representations (MERs) are a common feature in learning environments. They increase student motivation and can be designed to serve several learning purposes (Ainsworth, 1999).

There is a firm belief among computer science educators that visualization can make a difference in helping students learn programming concepts more effectively. For instance, in a survey made among participants at the ITiCSE 2002 conference of the ACM, 93% of the 66 respondents agreed or strongly agreed that this was the case, 7% were neutral or had no opinion, and none disagreed (Naps, Roessling, Almstrum, Dann, Fleischer, Hundhausen *et al*., 2003). However, research on algorithm animation systems has mainly focused on their technical implementation. As a consequence, two notable obstacles (Naps *et al*., 2003a) to their widespread adoption in education have not been sufficiently addressed and they now are the foci of attention. Firstly, there is no evidence of their educational benefits. Empirical evaluations of the educational effectiveness of animations (Hundhausen, Douglas & Stasko,

[*] Corresponding author. Fax: +34 91 488 70 49. E-mail address: angel.velazquez@urjc.es (J.Á. Velázquez-Iturbide).

2002) have led to consensus about the importance of student engagement in obtaining educational success (Naps *et al*., 2003a). Secondly, the construction of animations requires considerable effort on the part of instructors, to the point of being a bottleneck for the widespread adoption of animation systems. This article focuses on this second factor.

Several formal or informal studies support the importance of instructors' workload. For instance, a well documented experience is reported by Pollack and Ben-Ari (2004), who describe several animation systems and the criteria used to select one of them. Some factors relate to the adequacy of the system to the course or to the control of animations, whereas others relate to the effort required of the teacher: ease of installation, support to easily create demonstrations, support to create data sets to answer questions, and a save/load facility.

In a more general setting, several surveys have been conducted on the educational use of visualizations. The report of a working group organized at the ITiCSE 2002 conference contains information about three surveys concerning the educational use of visualizations (Naps *et al*., 2003a). The "preconference survey" contains more elaborate information about the factors that make the respondent or respondent's colleagues reluctant or unable to use animations. The options can be grouped into factors related to the time it takes to prepare the infrastructure (e.g. to install the software), the time it takes to develop animations, and the quality and adequacy of the tools (e.g. to adapt animations to the teaching approach of a course). The option most cited as a major impediment was the time it takes to develop animations (by two thirds of the respondents). Karavirta, Korhonen and Tenhunen (2004) have also recently conducted a more specific survey on development effort, involving 22 respondents. According to their results, the two most common advantages of animation systems were the features allowing users to create animations easily, and sufficient navigation possibilities in the final animation.

A key dimension in the effort problem is the level of abstraction of animations. Program visualization is typically performed automatically (e.g. by pretty-printing the code or by displaying the state of data structures), but the level of abstraction obtained is low. The use of algorithm animations provides a more abstract view that relates to the underlying algorithmic ideas, but their construction is typically very demanding. As effortlessness can only be achieved automatically at the level of program animation, we wondered whether this could be a satisfactory starting point to more expressive animations. This hypothesis is supported by the existence of program animation systems which provide complex animations. For instance, the Jeliot system (Haajanen, Pesonius, Sutinen, Tarhio, Teräsvirta & Vanninen,1997; Sutinen, Tarhio & Tërasvirta, 2003) allows program animations to be generated semi-automatically. The system automatically identifies the "interesting events" of the animation of a Java program and generates an applet to animate it. The user defines and customizes the views by means of simple dialogs. However, Jeliot exhibits some important drawbacks for educational use. In particular, the lack of a load/store facility directly affects the instructor's workload.

In this article we describe a new, user-friendly approach to constructing animations requiring minimal effort. Our approach simultaneously allows users to generate visualizations and animations automatically, while also giving them powerful customization facilities. Visualizations and animations are obtained as a side-effect of program execution, so the animation system is embedded in a general purpose programming environment. The user can make program animations more attractive in two ways. Firstly, he/she may customize visualizations to meet his/her visual and typographical preferences. Secondly, and more importantly, the user may choose the relevant parts of a program execution (according to his/her own criteria) to form an animation which is more meaningful than one that would be obtained by simply employing a complete step-by-step or one-step execution. In order to keep user workload low, these facilities are carried out via a simple, user-friendly interactive manipulation, typically by means of menus or dialogs, thus avoiding the need to learn a customization or script language.

We have applied our approach to the functional paradigm. Compared to other paradigms, visualization of functional programs has seldom been addressed; the interested reader may consult a comprehensive survey elsewhere (Urquiza-Fuentes & Velázquez-Iturbide, 2004). From the visualization point of view and given its simplicity in comparison to other paradigms, functional programming offers a unique opportunity to experiment. In particular, our work is based on the programming environment WinHIPE (Velázquez-Iturbide, 1994; WinHIPE, 2006), which offers a view of the evaluation of expressions as term rewriting.

The structure of the article is as follows. In section 2, we explain the user view of our approach for functional programming. Section 3 describes our approach to produce low workload animations. In section 4 we describe the results of assessing our system. Finally, we include a discussion and we summarize our conclusions.

ANIMATION OF THE EXECUTION OF FUNCTIONAL PROGRAMS

In this section we briefly explain the user's perception of the capabilities of our programming environment WinHIPE (Velázquez-Iturbide, 1994). It is an integrated programming environment for the functional language Hope running under Windows. Firstly, we show the model of execution of functional programs provided by the environment, which is based on the evaluation of expressions. Then we summarize the visualization formats and the animation interface of functional expressions automatically generated during an evaluation.

Expression Evaluation

A programming environment for functional programming allows the evaluation of any expression composed of values and operators predefined in the programming language. The programmer can also declare data types and functions in a program. Once the program is compiled, the programmer can evaluate any expression composed of either predefined elements or elements declared in his/her program. The resulting functionality of the programming environment can be compared to a programmable calculator.

For example, consider the typical factorial function:

```
dec fact: num -> num;
--- fact n <= if n=0 then 1 else n*fact(n-1);
```

This definition can be used to evaluate functional expressions involving the factorial function. Thus, `fact 4` will yield the value `24`.

WinHIPE allows the inspection of the evaluation process in a controlled manner. It models the evaluation of an expression at a high level of abstraction, as a rewriting process whose initial expression is rewritten into intermediate expressions until a final expression (i.e. the value of the original expression) is obtained. The interpreter provides five actions to control the pace of the evaluation:

- Progressing one rewriting step.
- Progressing *n* rewriting steps.
- Progressing until a breakpoint is reached. Currently, a breakpoint is the application of any function selected by the programmer.

- Evaluating the redex. (The redex or "*red*ucible *ex*pression" is the next subexpression to be rewritten or reduced. Examples included in the article highlight redexes in bold type or with a different color.)
- Completing the evaluation of the expression.

After performing an action, the resulting intermediate expression is displayed and he/she can invoke the next action. For example, we can trace the evaluation of `fact 4` in different ways. In the following evaluation, a simple arrow denotes a rewriting step, an arrow labeled with 'r' represents progress to a breakpoint, one labeled with 'x' represents evaluation of the redex, and one labeled with '*' represents complete evaluation. We have highlighted in bold the redex of each intermediate expression.

```
fact 4
↓
if 4=0 then 1 else 4*fact(4-1)
↓
if false then 1 else 4*fact(4-1)
↓
4*fact(4-1)
↓
4*fact 3
↓r
4*(3*fact 2)
↓r
4*(3*(2*fact 1))
↓r
4*(3*(2*(1*fact 0)))
↓x
4*(3*(2*(1*1)))
↓*
24 : num
```

The user also has the possibility to backtrack the evaluation to a given intermediate expression and resume it. For instance, in the evaluation above, the user could wish to backtrack to the expression `4*(3*(2*(1*1)))` and advancing from there step-by-step in order to consult the results of recursive calls. In the event of backtracking, the expressions following the selected expression are deleted.

Expression Visualization

A novel decision in the design of the environment was that it should always show the *whole* of each intermediate expression, so that the programmer knows the current global state of the evaluation. Each expression is automatically visualized according to the formatting and typographic choices made by the programmer.

Typically, intermediate expressions are displayed in textual format, according to the definition of the language, as in the previous example. In addition, WinHIPE provides graphical representations for lists and binary trees (Velázquez-Iturbide & Vázquez-Presa, 1999), thereby avoiding the disadvantage of textual representations of data structures, which are frequently unreadable. The result is a mixed display, where lists and binary trees are displayed graphically and the other kinds of expressions are displayed textually. Lists are visualized as sequences of elements arranged horizontally left to right.

We will briefly explain the graphical representation of binary trees in order that the figures below can be understood. Binary tree representations are based on common aesthetic criteria (Walker II, 1990); for instance, all the nodes positioned at the same depth are displayed at the same level. Our representations may contain three kinds of rectangles:

- A rectangle representing a node of a tree.
- A rectangle representing a subtree when its shape is still unknown.
- A rectangle enclosing a tree. It does not have any special meaning, other than to highlight the transition from a textual representation to a graphical one (the tree).

Figure 1 shows two visualizations, corresponding to consecutive expressions produced during the computation of the inverse of a binary tree. The three kinds of rectangles are present.

Expression Animation

Animations are displayed in WinHIPE as a discrete sequence of visualizations. An animation can be watched by means of either one or two (consecutive) visualizations at a time. In the case of two consecutive visualizations, an arrow is displayed in between the visualizations (as in the previous textual examples) and it is labeled so that the evaluation actions that led from the first to the second are briefly explained. Figure 1 shows the “animation window” containing two consecutive visualizations at the same time. Notice that the arrow separating the two visualizations represents one rewriting step.

The animation window contains an VCR-like interface at the bottom of the window so that animations can be played in the following ways: advance or backtrack one visualization, play at the specified speed, pause, and go to the first or the last visualization. The number of the current visualization relative to the total number of visualizations is also displayed, as is the animation speed.

HANDLING OF ANIMATIONS

Our aim has been to allow the construction of animations with low workload for the user, while being able to make them educationally valuable. In this section we describe how to construct, modify, store and retrieve animations in WinHIPE; a user-friendly interaction is provided for these operations, based on the metaphor of an animation as a document.

Animations as Documents

We use a clear and simple framework for constructing animations based on the metaphor of office documents. Office applications (also called business applications or personal productivity tools) became very popular with Macintosh and Microsoft Office because they provided user-friendly graphical user interfaces. An important feature of office applications is their facility to produce documents without the necessity to learn any language. There are many kinds of office documents: word documents, spreadsheets, slide presentations, web pages, etc. However, they share some common properties:

- Documents are usually produced by typing with the keyboard and by selecting with the mouse.
- Documents are easily modified by customizing their typography or layout.
- Documents are not only constructed, but they are also reused. Documents are produced and stored; afterwards, they can be retrieved, modified and stored again.

- Depending on the kind of document, they are composed of different, standard parts: text, graphics, scripting code, etc.

We consider animations as “documents” delivered by our programming environment. Following such a metaphor, animations must be easy to construct and modify with a graphical user interface, and the user must also be able to store and retrieve them.

Construction of Animations

The process to create an animation consists of two consecutive steps (Naharro-Berrocal, Pareja-Flores & Velázquez-Iturbide, 2000), where the user only has to click menu options and checkboxes. Figure 2 gives an overview of this process, which we examine in detail.

In a first step, the expression e_1 is typed and evaluated. The user controls the evaluation using the actions he/she considers most appropriate ($a_1, \ldots, a_{n-1}$ in Figure 2), thereby automatically generating visualizations as a side-effect ($v_1, \ldots, v_n$ in Figure 2). This sequence of visualizations, separated by labeled arrows that denote the actions performed (as shown in Subsection 2.1), is displayed in the “visualization’s window”.

In a second step, the user must select the subset of those generated visualizations that will form the animation ($v'_1, \ldots, v'_m$ in Figure 2, where $m \leq n$). Each original visualization v_i has an adjacent check box (in the visualization’s window) to indicate whether it is selected to form the animation. By default, all of the visualizations are selected, but the user may exclude irrelevant ones (according to his/her subjective choice).

Once these steps have been followed, the user can play the animation. Figure 2 illustrates that this can be achieved in two ways: either playing it within the programming environment (using the animation window shown in Figure 1) or after generating an analogous web page (as explained below in subsection 3.4). Both kinds of animations have a use in their own right. Most typically, the first format is used for animations of programs in progress, and the second one is used for animations of finished programs, thereby allowing the user to complement them with code and explanations. The latter format will be used by teachers to prepare lessons and by students either to document assignments or to reinforce self-study.

One factor that must be taken into account to keep consistency during rewriting is that the arrow summarizing the rewriting relationship between each pair of visualizations must always describe the actions necessary to achieve such a transition. This description must be retained even if some visualizations are omitted. For instance, if three expressions e_1, e_2 and e_3 are related by one-step actions, this is represented as $e_1 \rightarrow e_2 \rightarrow e_3$. If e_1 and e_3 are selected but e_2 is omitted, the animation must show the transition from e_1 to e_3 as $e_1 \rightarrow^2 e_3$, thus making explicit the two steps necessary for such a rewriting. A simple algebra is used to maintain consistency in the rewriting information provided by arrows.

Web animations do not only contain the animation itself, they also have a more elaborate structure integrating text (Naharro-Berrocal, Pareja-Flores, Velázquez-Iturbide & Martínez-Santamarta, 2001). Their structure mirrors lessons on algorithmic problem solving since it consist of four parts: problem statement, algorithm description, program, and animation. The first two parts can be completed by the user with carefully written explanations. The animation also uses a video player interface. Figure 3 contains a sample of a web animation, with default dummy explanations.

Selection of visualizations

One of the steps in the construction of an animation is the selection of the visualizations that will form it. When long sequences of visualizations are produced, they do not fit into a single window; visualizations are then displayed in a window with a scrolling bar, the visualization's window. However, the programmer does not have a global view of all the visualizations necessary to make an appropriate selection.

One solution could be to maximize the number of visualizations that are displayed simultaneously to facilitate the selection of relevant visualizations, as in the PowerPoint slide classifier. Every visualization is reduced in size so that a grid composed of smaller images is displayed in a new window, the "miniature's window". (We use the term "miniature" to refer to the reduction of a visualization.)

However, the visualizations generated within WinHIPE have more complex "reading properties" than slides:

- They represent visualizations of varying size and shape (they can be squares, horizontal rectangles or vertical rectangles).
- They have different internal structures.
- Typically, they have a varying number of elements, therefore small fonts are preferred.
- In general, the user does not need to be able to read everything in a visualization, but simply the most relevant parts. However, all the parts in a visualization have fonts with equal size, so in practice this reading requirement implies being able to read any part of the visualization.

Therefore, the miniature's window must reduce visualization size while simultaneously preserving comprehensibility. Notice that the comprehensibility concept is fuzzy, as it relates to readability. It also depends on the particular algorithm: in some cases he/she only needs certain details about the current visualization, such as whether a condition is being tested, or which part of a data structure is being visited. However, in other cases, full details are required, e.g. values and operators in an arithmetic expression.

We have enhanced the miniature's window with some features that provide improved comprehensibility (Naharro-Berrocal, Pareja-Flores, Urquiza-Fuentes & Velázquez-Iturbide, 2002). The user is hardly aware of some of them, such as the algorithm used to reduce visualizations. However, other features affect user interaction:

- Size, proportions and layout. The miniatures have a size of 244×184 pixels by default, but the user may adjust their size manually, from 48×36 to 244×184 pixels in 10 per cent steps. Proportions of visualizations are maintained and they never become enlarged. Their layout is determined dynamically, fitting images within the window in a left to right and top-down process.
- Highlighting the current visualization. Although we ideally want to be able to examine all of the miniatures, our interest switches and is focused on one visualization at a time. One way of improving the comprehensibility of the focused visualization consists of highlighting it by showing it at "natural" size. We have modified (Gortázar-Bellas, Urquiza-Fuentes & Velázquez-Iturbide, 2003) the flip zoom interaction technique (Holmquist, 1997) so that the selected visualization (i.e. the focus) is displayed at natural size. The row to which the focus belongs is split into two rows, as shown in Figure 4: the first one aligned to the left and the second one, to the right. The focus is placed to the right of the first row if it fits; otherwise, it is placed to the left of the second row.

- User navigation. Initially, no miniature is selected, but the user may select and deselect any miniature with the mouse. He/she can also change the focus by using the keyboard to move the cursor both horizontally and vertically. When there is a change of focus, screen changes are limited to reduce user distraction (Gortázar-Bellas *et al*., 2003).

Figure 4 shows the miniature's window with all the visualizations generated while evaluating an expression that mirrors a binary tree. It contains 20 images, but only 10 have been selected to form an animation and one of them is zoomed.

Customization

At any moment and by means of simple dialogs, WinHIPE allows the programmer to customize the visualization of expressions in several ways (Velázquez-Iturbide & Vázquez-Presa, 1999):

- The programmer can select either pure textual or mixed text-graphics visualization of expressions.
- The redex can be highlighted in a different color with respect to the rest of the expression. An horizontal bar is also drawn above it.
- The typography of visualizations can be customized either by demonstration (for pretty-printing formats) or by using Word-like dialogs (for graphical elements). In its present version, WinHIPE allows the user to customize from a choice of 9 pretty-printing formats, 12 distances, 9 colors, 9 lines, and 8 additional graphic features. This kind of customization is accomplished by means of dialogs, where a visualization sample is displayed showing the effect of the new selection on the last generated visualization.
- Long expressions can be automatically abbreviated using a "context+focus" or "fish-eye" technique that only shows the parts most relevant to an understanding of the current evaluation state. This produces simplified visualizations that provide a balance between displaying the whole expression and only showing the subexpression containing the redex. The amount of information to be shown is controlled by the user via a natural number.

Figure 5 shows three different displays of an expression obtained while reversing a given binary tree. The second image is similar to the first one, but it has different typographic properties and the redex is not highlighted. The third image is an abbreviated fish-eye view of the first one; notice that it is shown more information of the left subtree because it includes the redex, and therefore it is more interesting at the current state of evaluation.

Storage and Retrieval

Given that an animation is considered a document, it can be stored and retrieved by means of an atomic action. From an implementation point of view, WinHIPE animations only require a directory to store several files containing its constituent parts. An animation is formed by the following elements:

- A program.
- An expression.
- A sequence of evaluation actions that provide a particular view of expression evaluation.
- Configuration options.

In effect, given a program and an expression to evaluate, the evaluation actions define the visualizations that will form our target animation. Configuration options specify both the appearance of individual visualizations and of the animation.

Web animations contain two more elements: user explanations of the problem and of the algorithm. A preliminary collection of web animations generated with WinHIPE can be found at the web site (WinHIPE, 2006, “links & downloads” section). These animations have not been modified after being generated, so they contain dummy explanations. (We are currently redesigning the functionalities and implementation of web animation support, so a new collection will be delivered in brief.)

Configuration and Iteration

In order to reduce interaction to a minimum, we have been concerned with some practical issues. In the first place, a configuration file saves customization preferences, saving the user from having to define them in every session. Configuration also includes the binding of the graphical format of binary trees to a user-defined data type.

It is unrealistic to think that animations are planed and built in a single step. As most products requiring human assistance, it is more common to iterate until the user is completely satisfied. This iteration is mainly found in three kinds of interaction. Firstly, the user may wish to change some customization options. In order to guarantee coherence, such a change does not affect a single visualization, but all the generated visualizations and the animation itself.

Secondly, the user may wish to generate some expressions which were omitted before in the evaluation. Ideally, the user schedules and performs the rewriting actions that will illustrate the evaluation. The expression `fact 4` in Subsection 2.1 is a good example of schedule, where the first steps are shown in detail, but after the first recursive call only some expressions are shown. However, it is more realistic to consider that the user may need to backtrack in the middle of the evaluation to modify the set of generated expressions. The environment provides a backtracking facility to rewind the evaluation to any intermediate expression, saving him/her from restarting at the beginning. As a consequence, the visualizations following the new active one are deleted.

Thirdly, the user may wish to change slightly the input data of an algorithm (i.e. the expression to evaluate), such as the value of an element in a data structure. Instead of having the user to recreate the animation from scratch, WinHIPE supports the rebuilding of such an animation in an easy and consistent way. A “rebuilding animation” facility is automatic for modifications in input data that do not alter execution behavior (i.e. the nature and number of rewriting steps remains constant). For situations where there are changes in the rewriting process, it also allows rebuilding the part they have in common.

EVALUATION OF THE ANIMATION SYSTEM

We have been using HIPE for seven years on a programming languages course with a strong functional programming component. From fall of 1997 until spring of 2002, we used a textual MS-DOS version of the environment (TurboHIPE). Since then we have used the graphic Windows version (WinHIPE).

In the spring of 2002 we held an evaluation session (Naharro-Berrocal, Velázquez-Iturbide, Pareja-Flores & Medina-Sánchez, 2001) with students to measure the educational effectiveness of the two versions of HIPE. We used two student groups, the experimental group (using WinHIPE) and the control group (using TurboHIPE). The results showed that students who used WinHIPE were enthusiastic about animations but they did not learn more than those who used TurboHIPE. We also become aware of the fact that students needed longer exposure to the Windows version in order to become familiar with the new user interface and to understand the graphical notation used for visualizations.

In the spring of 2004 we held a second evaluation session to test several aspects of the usability of WinHIPE; full details can be found elsewhere (Medina-Sánchez, Lázaro-Carrascosa, Pareja-Flores, Urquiza-Fuentes & Velázquez-Iturbide, 2004). Fifty-two subjects participated in the evaluation session. During the term, students had six laboratory sessions plus one session for the laboratory exam. We held the evaluation during the fifth laboratory session. During the first four sessions, the environment was used to compile and execute functional programs, so students were familiar with the Windows version. During these four sessions, the graphic capabilities of WinHIPE were available to students, but they were not addressed explicitly.

The evaluation tried to obtain empirical evidence on several aspects of WinHIPE**.** What is of particular relevance for the purposes of this article is evidence about whether animations can be built without effort. Of the three main hypotheses that guided the design of the evaluation, the two that are relevant to effort are that "animations are easy to use" and that "the animation construction process is easy to learn".

The lab session started with an explanation from the instructor about the goals of the evaluation. Then, students had to perform several tasks that cover the most important use cases of program animation:

- Program comprehension. Firstly, the animation construction process was explained. Afterwards, a problem was outlined and a program solving the problem was given out without any further explanation. Finally, subjects were asked to build an illustrative animation of the problem and they were given twenty minutes to perform this task.
- Program development. Students were given a problem analogous to the previous one and were given twenty-five minutes to solve it.
- Program debugging. Firstly, the teacher debugged a program using WinHIPE without animations; this took five minutes. Then, a buggy program with five errors was given to the students and they were asked to fix the errors within a time limit of twenty minutes.

After the evaluation, the subjects were asked to fill in a questionnaire about their experience with WinHIPE. The questionnaire contained nine questions to be answered by selecting numbers on a scale of 0-4. Two further questions to be answered in a free format were also included. Below we summarize the results of those questions concerning effort.

The first question was whether they had experience of using the visualization and animation capabilities of WinHIPE prior to the session. The results showed that 46.2% of the subjects had no previous experience, while 30.8% had used them sporadically, 15.4% frequently and 7.6% had used them extensively.

A second question was whether, after the session, students considered animations to be easy to use. The results showed that 7.69% of the students completely agreed with this statement, 53.85% roughly agreed, 30.77% had no opinion and 7.69% roughly disagreed. No student disagreed completely. Therefore, the percentage of students who agreed was eight times higher than the percentage of those who disagreed (61.54% vs. 7.69%).

These results were compared to previous experience of using animations with WinHIPE (question 1). We made multi-sample comparison, using ANOVA and Kruskal-Wallis test. The statistical analysis leads to the conclusion that animations are easy to use independently of students' previous experience of them.

To obtain an indication of the easiness of construction and use of animations, we also asked (question 7) whether the students had used WinHIPE animations to debug the program containing errors, even though they had not been asked to do so. About half of the students had done so (27 out of 52). When we related these results to the students' previous experience of using animations, we found that the use of animations did not depend on their having previous experience. As a consequence, we concluded that students who used animations to debug must have had a different reason for doing so, such as making use of the graphic support to debug effortlessly or even because they have a visual learning style.

DISCUSSION

We believe that there are several key issues that contribute to the simplicity of our approach to building animations. One of them is the use of the document metaphor for animations. Metaphors are a well-known way of explaining a concept by means of different concepts and we have found our document metaphor a useful way to achieve a sound specification of both user interaction and the facilities provided. It has also proven useful for the users by helping them understand how to handle animations.

Metaphors are often used to model the different parts of algorithm animation systems. Thus, the video player metaphor is typically used in most systems (Gloor, 1998) to control the direction and pace of animations. It consists of a set of standard buttons (play, pause, rewind, etc.) and a speed control, often complemented with algorithm-specific controls. Related stimulating metaphors for playing animations are comic strips (Biermann & Cole, 1999) and overlapping slides (Terada, 2000). The electronic book is another metaphor that helps to structure animation elements to form a hypermedia lesson or set of lessons on algorithms (Brown & Raisamo, 1997). Low-fidelity animations are inspired by story boards made of simple art supplies (Hundhausen & Douglas, 2002). Finally, a theater metaphor has been used to explain Jeliot (Haajanen *et al*., 1997; Sutinen *et al*., 2003).

As far as we know, the term "effortless" was used for the first time to describe the anonymous system by LaFollette, Korsh and Sangwan (2000). Their system was an integrated programming environment with a graphic user interface divided into several panes to show different parts of the program execution: code, globals, locals, parameters, call stack, heap, and operations. Transitions among different states of the program were controlled by the debugger and were smoothly animated. All of this functionality was provided with very little effort from the user: the type declarations of variables to be visualized ("self-animating" types) simply had to be written in capitals.

Effortless creation of animations (or simply "effortlessness") is a fuzzy term and therefore, it is too subjective to be measured accurately. It can be compared to software metrics, which are not formulas whose results definitively establish the quality of a piece of software, but they do facilitate its estimation. Karavirta, Korhonen, Nikander and Tenhunen (2002) tried to provide a deeper understanding of effortlessness by estimating it for several animation systems and relating their results to several categories. The systems were ranked with respect to effortlessness based on availability of a WYSIWIG (i.e. What You See Is What You Get) editor and, more importantly, the ability to provide of on-the-fly use. They identified generality and presentation style as the categories (Price, Baecker & Small, 1993) most relevant to effortlessness. Generality measures the area of use of the system and two subcategories were considered here, the first and most important one being applications and the second, language. Presentation style describes the appearance of visualizations and two further categories were considered: graphic vocabulary and animation. The authors conclude that generality and graphic vocabulary are relevant as follows: the more general a system is, the more effort is needed to create a visualization with it; the more primitive graphic vocabulary a system has, the more effort is needed to create a visualization with it.

Based on Karavirta *et al*.'s taxonomy, WinHIPE should require more effort than it actually does since its generality is medium/high. However, this is compensated by its complex graphic vocabulary (it deals with textual expressions, lists, and binary trees). We feel that they have not considered an important category that is at the heart of WinHIPE's effortlessness. The category we refer to is method (Price *et al*., 1993), which describes how the visualization is specified. In particular, the subcategory of visualization specification style describes the style in which the visualization is specified (including customization, if applicable). WinHIPE produces automatic visualizations based on a set of predefined formats. Customization of visualizations is carried out by dialogs and customization of animations, by selection.

More recently, Karavirta and colleagues have conducted a survey among teachers using animation in order to gain further insight into the factors that most influence effortlessness (Karavirta *et al*., 2004). In this second study, they considered effortlessness to be influenced by three categories: scope, integrability, and interaction techniques. However, we do not agree with this second proposal. From our point of view, effortlessness is simply a measure of the ease with which a system allows animations to be generated. It is indeed true that for animations to have a wider acceptance in mainstream computer science education there are additional factors, such as integrability into courses, which are important. However, these factors do not relate to a definition of effortlessness and need to be assessed independently, at least to avoid confusion. For instance, we can imagine an effortless system that is difficult to integrate into a course or, conversely, an adaptable system that requires considerable effort from the instructor. Evidently, none of these systems is ideal for a course, but if they are clearly assessed, the instructor can make a reasoned choice.

Further research is required if "effortlessness" is to be clearly understood. One promising area of research consists of identifying the approaches most widely used in effortless systems and studying how they manage to reduce workload for the user; several approaches can be found in Hundhausen and Douglas (2002) and in Karavirta *et al*. (2004). For instance, program visualization systems such as WinHIPE or LaFollette *et al*.'s reduce workload by being tied to a language processor that automates the generation of predefined visualizations.

CONCLUSIONS

Recently, some working groups have reported that one of the main obstacles to widespread adoption of algorithm animation systems is the excessive effort they place on instructors (Naps *et al*., 2003a; Naps, Roessling, Anderson, Cooper, Dann, Fleischer *et al*., 2003). We have presented an approach based on automation and customization that reduces the workload involved in building animations to a minimum. Visualizations and animations are generated automatically. The user may customize them in a user-friendly way to construct more expressive program animations. These operations are easily carried out via user interaction based on the office document metaphor. We have applied this approach to the functional paradigm, thereby extending the WinHIPE programming environment. An important factor that helps to reduce the effort required form the educator is that not only do we consider the construction of animations, but also their modification and maintenance. The empirical evidence obtained from a carefully planned evaluation supports our claim that our approach requires little effort.

Our proposal has focused on the effort required to develop and maintain animations. In addition, we are now improving WinHIPE with respect to the other major factors that may make an instructor reluctant to use animations (Naps *et al*., 2003b). For instance, better support could be given to the establishment of infrastructure (e.g. providing a simple installation package) or certain educational features could be incorporated to increase engagement (e.g. stop-and-think questions).

A key issue in our research was our desire to keep animations simple in order to make it easier to explore the feasibility of our approach. Consequently, we focused on the functional programming paradigm using operational semantics based on term rewriting and we restricted ourselves to the use of discrete animations. However, we think that our approach is general enough to be applied to any (sequential) programming paradigm. In particular, we are examining how to apply it to the object-oriented imperative paradigm in order to assess its generality.

ACKNOWLEDGMENTS

This work was supported by project TIN2004-07568 of the Spanish Ministry of Education and Science. We also want to thank Carlos Lázaro-Carrascosa and Mª Ángeles Medina-Sánchez for their aid in evaluating our system.

REFERENCES

Ainsworth, A. (1999). The functions of multiple representations. *Computers & Education, 33*(2-3), 131-152.

Biermann, H. & Cole, R. (1999). Comic strips for algorithm visualization. *New York University. Tech. Rep. 1999-778.*

Brown, M. H. & Raisamo, R. (1997). JCAT: Collaborative active textbooks using Java. *Computer Networks and ISDN Systems, 29*, 1577-1586.

Gloor, P.A. (1998). User interface issues for algorithm animation. In J.T. Stasko, J. Domingue, M.H. Brown & B.A. Price, *Software Visualization*. Cambridge, MA: MIT Press.

Gortázar-Bellas, F., Urquiza-Fuentes, J., & Velázquez-Iturbide, J.Á. (2003). Reduction and flip-zooming in comprehension-preserving miniatures of program visualizations. *Proceedings of the 3rd IASTED International Conference on Visualization, Imaging, and Image Processing, II* (pp. 855-861). Anaheim, CA: ACTA Press.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., & Vanninen, P. (1997). Animation of user algorithms on the Web. *Proceedings of the 1997 IEEE Symposium on Visual Languages* (pp. 360-367). IEEE Computer Society Press.

Holmquist, L.E. (1997). Focus+context visualization with flip zooming and the Zoom Browser. *Proceedings of the ACM SIGCHI'97 Conference on Human Factors and Computing Systems* (pp. 263-264). New York, NY: ACM Press.

Hundhausen, C.D., & Douglas, S.A. (2002). Low-fidelity algorithm visualization. *Journal of Computer Languages and Systems, 13,* 449-470.

Hundhausen, C.D., & Douglas, S.A., & Stasko, J.T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing 13*(3), 259-290.

LaFollette, P., Korsh, J., & Sangwan, R. (2000). A visual interface for effortless animation of C/C++ programs. *Journal of Visual Languages and Systems, 11*, 27-48.

Karavirta, V., Korhonen, A., Nikander, J., & Tenhunen, P. (2002). Effortless creation of algorithm visualization. *Proceedings of the Second Annual Finish/Baltic Conference on Computer Science Education* (pp. 52-56). Turku, Finland: Turku Centre for Computer Science.

Karavirta, V., Korhonen, A., & Tenhunen, P. (2004). Survey of effortlessness in algorithm visualization systems. *Proceedings of the Third Program Visualization Workshop* (pp. 141-148). Coventry, UK: University of Warwick.

Medina-Sánchez, M.Á., Lázaro-Carrascosa, C.A., Pareja-Flores, C., Urquiza-Fuentes, J., & Velázquez-Iturbide, J.Á. (2004). Empirical evaluation of usability of animations in a functional programming environment. *Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid Tech Rep 141/04*, http://vido.escet.urjc.es/winhipe/dwld/TR-141-04.pdf.

Naharro-Berrocal, F., Pareja-Flores, C., Urquiza-Fuentes, J., & Velázquez-Iturbide, J.Á. (2002). Approaches to comprehension-preserving graphical reduction of program visualizations. *17th ACM Symposium on Applied Computing* (pp. 771-777). New York, NY: ACM Press.

Naharro-Berrocal, F., Pareja-Flores, C., & Velázquez-Iturbide, J.Á. (2000). Automatic generation of algorithm animations in a programming environment. *Proceedings of the 30th ASEE/IEEE Frontiers in Education Conference* (pp. 6-12), Champaign, IL: Stiples Publishing.

Naharro-Berrocal, F., Pareja-Flores, C., Velázquez-Iturbide, J.Á. & Martínez-Santamarta, M. (2001). Automatic Web publishing of algorithm animations. *Informatik/Informatique, 2*, 41-45.

Naharro-Berrocal, F., Velázquez-Iturbide, J.Á., Pareja-Flores, C., & Medina-Sánchez, M.Á. (2001). Valoración de la eficacia de las animaciones de algoritmos en el aprendizaje de la programación funcional. *Working paper.* http://vido.escet.urjc.es/winhipe/docs/209-erpmeuiona.pdf.

Naps, T., Roessling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., & Velázquez-Iturbide, J.Á. (2003). Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin, 35*(2), 131-152.

Naps, T., Roessling, G., Anderson, J., Cooper, S., Dann, W., Fleischer, R., Koldeofe, B., Korhonen, A., Kuittinen, M., Leska, C., Malmi, L, McNally, M., Rantakokko, J. & Ross, R.J. (2003). Evaluating the educational impact of visualization. *SIGCSE Bulletin, 35*(4), 124-136.

Pollack, S. & Ben-Ari, M. (2004). Selecting a visualization system. *Proceedings of the Third Program Visualization Workshop* (pp. 134-140). Coventry, UK: University of Warwick.

Price, B., Baecker, R., & Small, I. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages and Systems, 4*(3), 211-271.

Stasko, J.T., Domingue, J., Brown, M.H., & Price, B.A. (1998). *Software Visualization*. Cambridge, MA: MIT Press.

Sutinen, E., Tarhio, J., & Tërasvirta, T. (2003). Easy algorithm animation on the Web. *Multimedia Tools and Applications, 19*, 179-194.

Terada, M. (2000). Animating C programs in paper-slide-show. *Proceedings of the First Program Visualization Workshop* (pp. 79-88). Joensuu, Finland: University of Joensuu.

Urquiza-Fuentes, J. & Velázquez-Iturbide J.Á. (2004). A survey of visualizations for the functional paradigm. *Proceedings of the Third Program Visualization Workshop* (pp. 2-9). Coventry, UK: University of Warwick.

Velázquez-Iturbide, J.Á. (1994). Improving functional programming environments for education. In M.D. Brouwer-Janse, & T.L. Harrington, *Man-Machine Communication for Educational Systems Design*. Berlin Heidelberg, Germany: Springer-Verlag.

Velázquez-Iturbide, J.Á., & Vázquez-Presa, A. (1999). Customization of visualizations in a functional programming environment. *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference* (pp. 22-28). Champaign, IL: Stipes Publishing.

Walker II, J.Q. (1990). A node-positioning algorithm for general trees. *Software – Practice and Experience*, *20*(7), 685-705.

WinHIPE web site (2006). http://vido.escet.urjc.es/winhipe (accessed 29th March 2006).

FIGURE CAPTIONS

Fig. 1. An animation with two consecutive visualizations

Fig. 2. An overview of the construction process of an animation

Fig. 3. A web animation generated automatically with dummy explanations

Fig. 4. The miniature’s window

Fig. 5. Three visualizations of the same expression

FIGURES

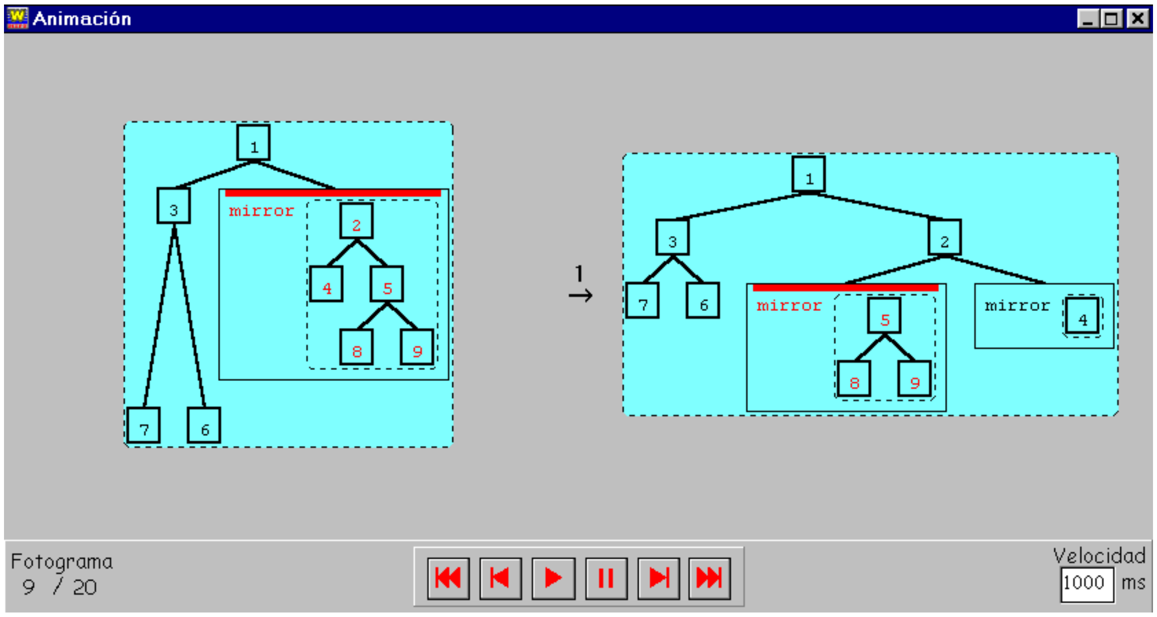


Fig. 1 An animation with two consecutive visualizations

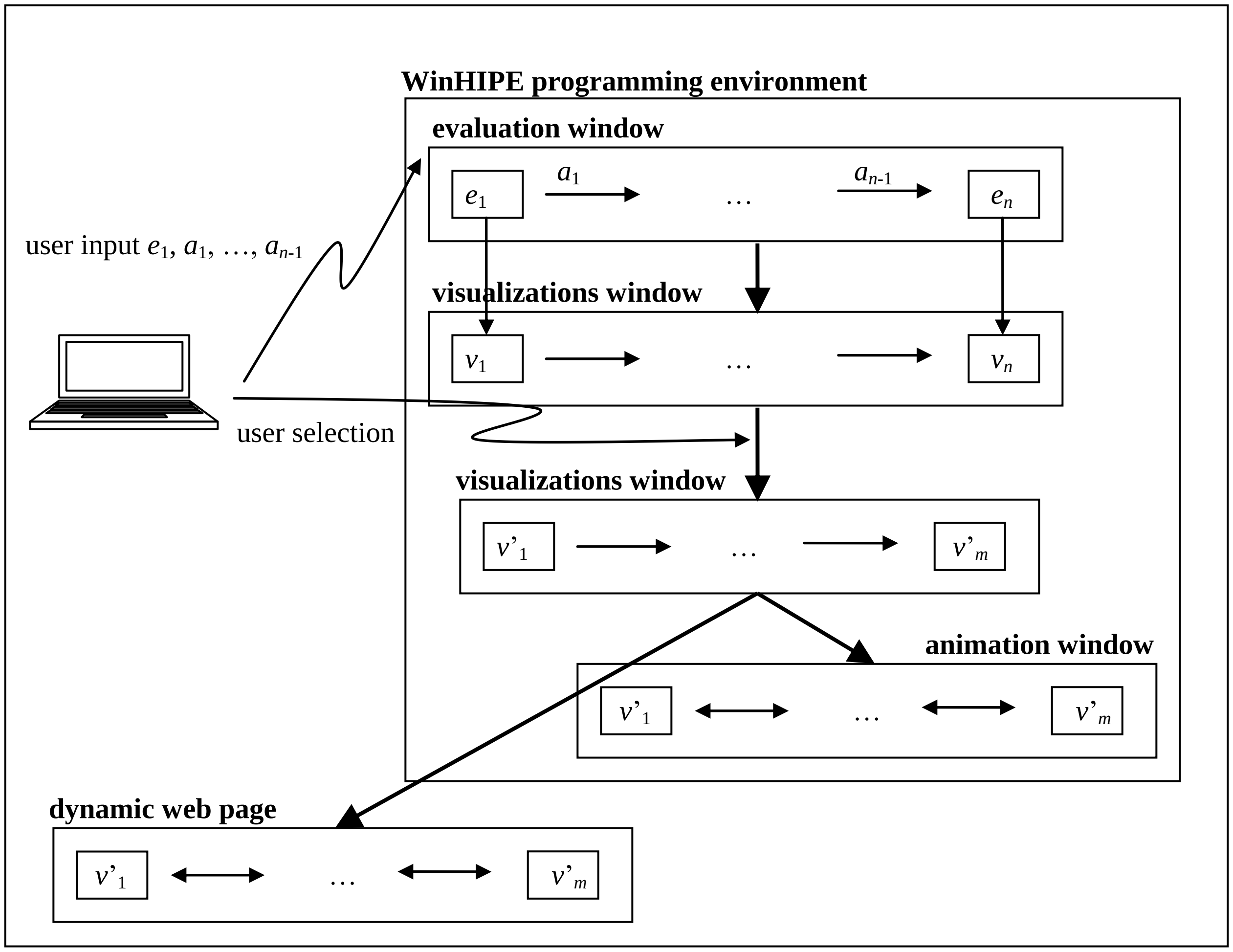
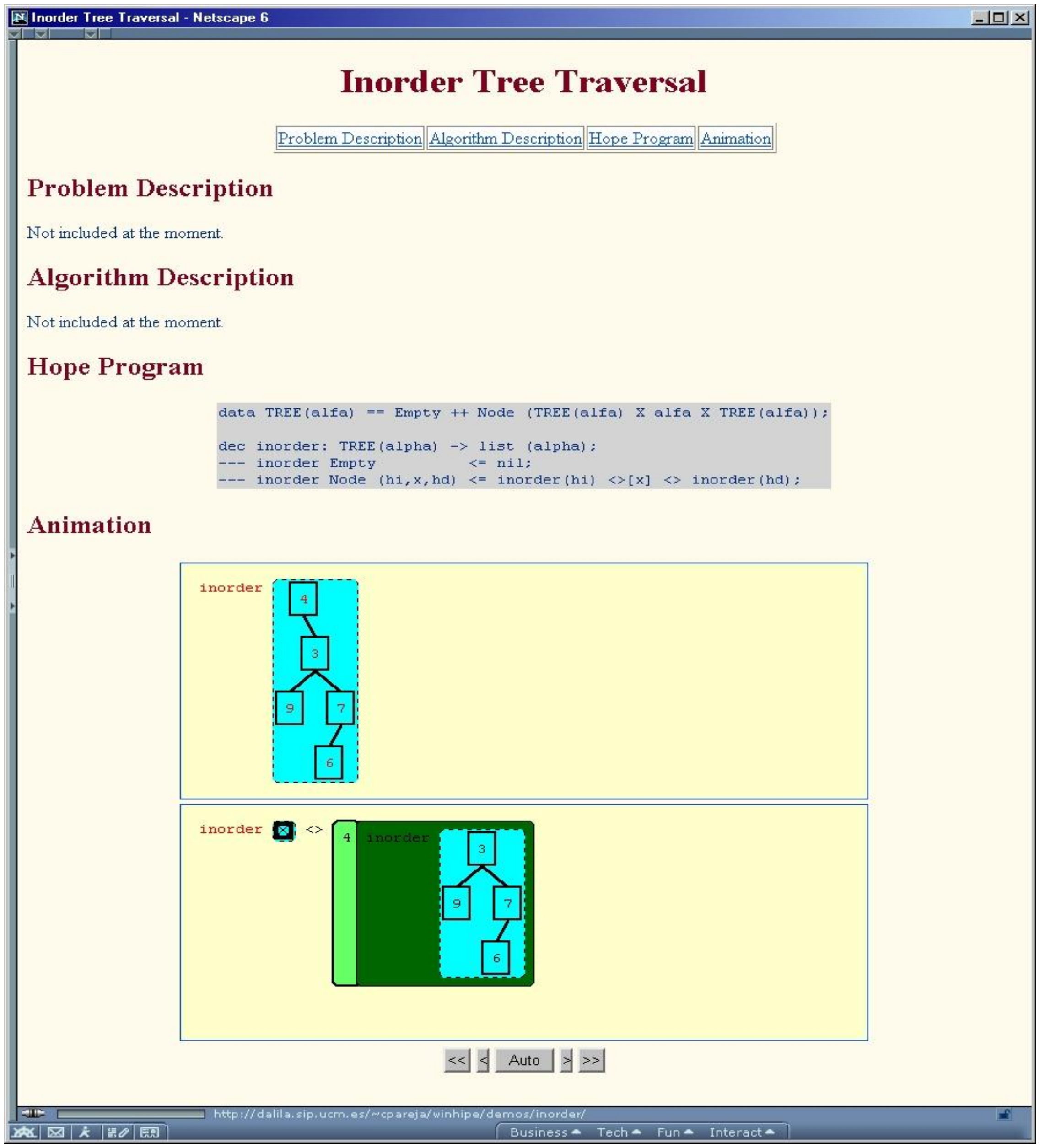


Fig. 2 An overview of the construction process of an animation



Inorder Tree Traversal - Netscape 6

Inorder Tree Traversal

Problem Description | Algorithm Description | Hope Program | Animation

Problem Description

Not included at the moment.

Algorithm Description

Not included at the moment.

Hope Program

```
data TREE(alfa) == Empty ++ Node (TREE(alfa) X alfa X TREE(alfa));

dec inorder: TREE(alpha) -> list (alpha);
--- inorder Empty          <= nil;
--- inorder Node (hi,x,hd) <= inorder(hi) <>[x] <> inorder(hd);
```

Animation

Fig. 3 A web animation generated automatically with dummy explanations

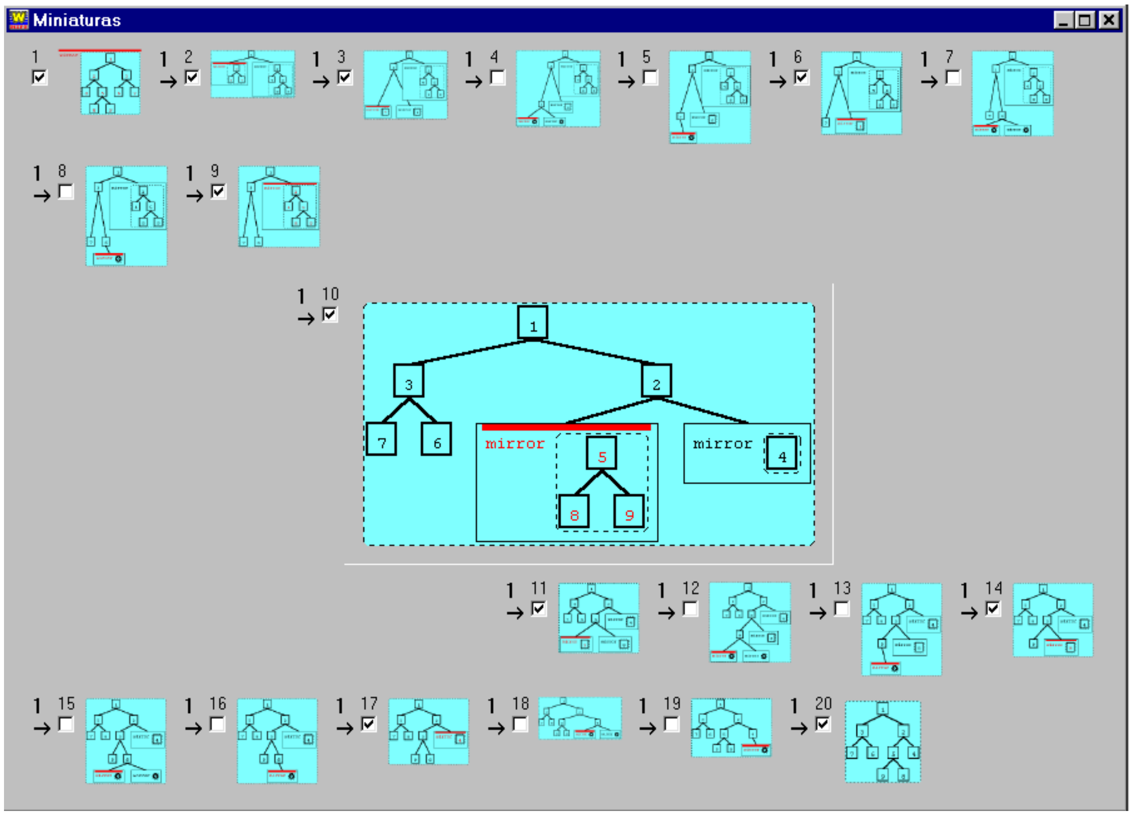


Fig. 4 The miniatures window

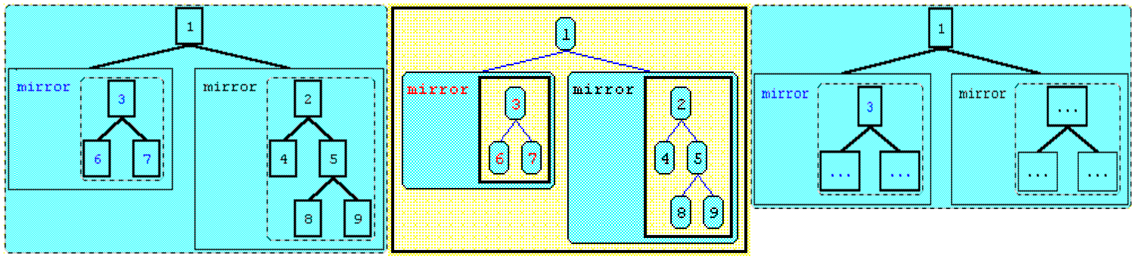


Fig. 5 Three visualizations of the same expression