WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization

Cristóbal Pareja-Flores, Jaime Urquiza-Fuentes & Jesús Ángel Velázquez-Iturbide

ACM SIGPLAN Notices Volume 42 Issue 3, March 2007

DOI: http://dx.doi.org/10.1145/1273039.1273042

WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization

Cristóbal Pareja-Flores
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Avda. Puerta de Hierro s/n
28040 Madrid, Spain
cpareja@sip.ucm.es

Jaime Urquiza-Fuentes
Departamento de Lenguajes y Sistemas Informáticos
Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
jaime.urquiza@urjc.es

J. Ángel Velázquez-Iturbide
Departamento de Lenguajes y Sistemas Informáticos
Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
angel.velazquez@urjc.es

ABSTRACT

The article describes an IDE for functional programming, called WinHIPE. It provides an interactive and flexible tracer, as well as a powerful visualization and animation system. The former tool is based on the rewriting model of evaluation, and the latter provides automatic generation of visualizations and animations, friendly support for customization, maintenance and exportation of animations to the Web, and facilities to cope with large scale. Its main advantage over other visualization systems is an effortless approach to animation creation and maintenance, based on generating visualizations and animations automatically, as a side effect of program execution. Finally, we briefly describe our experience using the system during several years in educational settings.

Keywords

Functional programming, programming environments, expression evaluation, term rewriting, tracing, program visualization, program animation.

1. INTRODUCTION

In the functional paradigm, a program typically consists of a set of equations defining data types and functions, both first and higher order. The programmer can define and use algebraic data types, polymorphic data types and functions. Functions are defined by a list of equations, each one with a different pattern.

For instance, the program in Fig. 1 defines a function that rebuilds a tree from its preorder and inorder traversals, assuming that all of its elements are different. This algorithm solves a known exercise proposed by Knuth [9], which is most typically used in the classroom as an example of tree or divide and conquer algorithms.

```
dec length: list(α) -> num;
--- length (nil)  <= 0;
--- length (_::l) <= 1 + length(l);

dec splitElem: α X list(α) -> list(α) X list(α);
--- splitElem(x,y::ys) <=
      if x = y then (nil,ys) else (y::ys_1,ys_2)
      where (ys_1,ys_2) == splitElem(x,ys)
      end;

dec splitLen: num X list(α) -> list(α) X list(α);
--- splitLen(0,ys)    <= (nil,ys);
--- splitLen(n,y::ys) <= (y::ys_1,ys_2)
      where (ys_1,ys_2) == splitLen(n-1,ys)
      end;

dec split: list(α) X list(α)
        -> list(α) X list(α) X list(α) X list(α);
--- split (x::xs,ys) <=
      let    (ys_1,ys_2) == splitElem (x,ys)
      in let (xs_1,xs_2) ==
                        splitLen (length(ys_1),xs)
      in     (xs_1,xs_2,ys_1,ys_2);

data TREE(α) == Empty ++
                Node (TREE(α) X α X TREE(α));

dec rebuild: list(α) X list(α) -> TREE(α);
--- rebuild (nil,nil) <= Empty;
--- rebuild (x::xs,ys)<= Node(rebuild(xs_1,ys_1),
                              x,
                              rebuild(xs_2,ys_2))
      where (xs_1,xs_2,ys_1,ys_2) ==
                                  split (x::xs,ys)
      end;

dec main: TREE(char);
--- main <= rebuild
```

Fig. 1. Rebuilding a tree from its pre- and in-order traversals.

Let us shortly review its strategy. We want to rebuild a tree *t* = *Node*(*t*1,*x*,*t*2) whose traversals are:

preorder (*t*) = "iynuocprbhgatle"
inorder (*t*) = "uncopyrightable"

It is obvious that the root is *x* = 'i' (the first element in the preorder traversal). This fact allows us to split the inorder traversal into "uncopyr" ++ "i" ++ "ghtable", resulting in the inorder traversals of subtrees *t*1 and *t*2:

inorder (*t*1) = "uncopyr"
inorder (*t*2) = "ghtable"

We also know the sizes of *t*1 and *t*2, so it is trivial to split the preorder traversal of *t* into the respective preorder traversals of *t*1 and *t*2:

preorder (*t*1) = "ynuocpr"
preorder (*t*2) = "bhgatle"

Functional programs neither contain variables nor assignments. Therefore, the notion of state is absent and functions are state-independent. This property is known as transparency referential. An immediate consequence is that expressions can be evaluated in any order, or even in parallel, without affecting the result (modulo termination).

These features make functional paradigm a very high level model, where a program is decomposed into pieces (functions) that glue other functions together, fostering a well structured, clean style, and yielding to reusable software. As a consequence, the functional paradigm currently plays an important role in CS education, allowing the student and the instructor to focus directly on algorithmic issues avoiding technical overhead.

On the other hand, functional compilers have successfully overcome drawbacks from their early stages, such as inefficiency, thanks to research in static analysis, efficient garbage collection, etc., and in optimizing compilers in general. Thus, this programming model has come out of the theoretical and academic scope, being frequently used by professionals for rapid prototyping and application development.

Currently, there are many educational environments for a wide variety of languages and platforms, ranking from simple command line compilers or evaluators to complex integrated development environments (IDEs). Learning environments may show intermediate states of programs under execution, as well as providing visualizations as a more expressive representation of control and data structures. Unfortunately, control and data structures are not separated in the functional model, so tracing and visualization are non-trivial tasks.

WinHIPE fulfils all of these goals in the functional programming model. It is an IDE for the functional paradigm [22] that provides an interactive and flexible tracer, as well as a powerful visualization and animation system. The former is based on the rewriting model of evaluation, and the latter provides automatic generation of visualizations and animations, friendly support for customization, maintenance and exportation of animations to the Web, and facilities to cope with large scale. All of these features make WinHIPE unique for teaching and learning, as well as for professional programming. Its main advantage over other visualization systems is an effortless approach to animation creation and maintenance [19][25], based on generating visualizations and animations automatically, as a side effect of program execution.

The article has the following structure. Section 2 describes the tracing facilities of WinHIPE. The third and fourth sections describe its visualization and animation facilities. Section 5 describes how WinHIPE handles different situations where large sizes can be cumbersome for the user. The sixth section briefly reports on our experience using the IDE. Finally, we discuss related work and give our conclusions.

2. TERM REWRITING

The evaluation model underlying functional programming is term rewriting. In this model, running a program consists in a process of "re-elaborating" a starting term, the main expression, step by step, until a normal form is reached. The main replacing step (namely β–reduction) applies a directed equation as a rewriting rule on a reducible expression (namely *redex*). For instance,

```
length ("ACM")
↓
1 + length ("CM")
↓
1 + (1 + length ("M"))
↓
1 + (1 + (1 + length ("")))
↓
1 + (1 + (1 + 0))
↓
1 + (1 + 1)
↓
1 + 2
↓
3
```

(The redexes of intermediate expressions are underlined.)

WinHIPE allows controlling and inspecting the evaluation process. The environment provides several ways to control the pace of evaluation:

- Performing one rewriting step, as is the case of steps in the example above.
- Performing *n* rewriting steps:

```
1 + length ("CM")
↓3
1 + (1 + (1 + 0))
```

- Performing the evaluation of the redex:

```
1 + length ("CM")
↓r
1 + 2
```

- Performing the evaluation of the whole expression:

```
1 + length ("CM")
↓*
3
```

One could reasonably argue that real programs are not so simple, and tracing a program by counting the number of steps is not a practical solution. Consequently, WinHIPE allows using breakpoints:

- Progressing until a breakpoint is reached. The absence of state forces us to make a new definition of breakpoint. Currently, WinHIPE considers a breakpoint the actual application of any function so identified by the programmer. For instance:

```
rebuild ("iynuocprbhgatle","uncopyrightable")
↓p
Node(rebuild("ynuocpr","uncopyr"),
     ´i´,
     rebuild("bhgatle","ghtable"))
↓p
Node(Node(rebuild("nuocp","uncop"),
          ´y´,
          rebuild("r","r")),
     ´i´,
     rebuild("bhgatle","ghtable"))
```

provided *rebuild* has been marked as a breakpoint.

The environment allows the programmer to control the pace of evaluation by successively selecting actions until the evaluation process is complete or it is abandoned. After each action, the resulting expression is displayed.

Testing and debugging are common activities, both for students and professionals. These activities are harder in the functional model than in the imperative one, because intermediate terms include data and control together. In last years, different approaches have been proposed for lazy and strict functional semantics [13]. In some of them, the graph (i.e. intermediate code in functional programming) produced for the traced program is different from the original graph without traces (i.e. its abstract syntax tree). WinHIPE lies in a second, cleaner approach: it regards traces as pure observations of the program, keeping the original graph.

Instrumentation is the first step to trace a program, and so to visualize it [14]. We have considered two ways of obtaining program traces, which differ on what element is instrumented to produce the trace: the interpreter or the user's program. The later is a quite simple approach. It is often implemented as an API that can be used in programs. Thus, programmers can define which elements of their program are traced. A drawback of this approach is that the program is changed, and its execution could then be altered. The former approach, interpreter instrumentation, is a complex task that implies rebuilding part, if not all, of the interpreter. Here, the interpreter generates the traces, keeping invariant the user's program and the semantics of the language. This allows producing traces for any program executed, without extra work for programmers. We implemented an instrumented interpreter in WinHIPE. Thus, a clean separation between the trace and the program being observed is established. Therefore, correctness is guaranteed, as its normal execution process is not altered.

3. EXPRESSION VISUALIZATION

A key decision in the design of WinHIPE was to display the whole of each intermediate expression. As a consequence, the programmer knows the current global state of the evaluation.

Typically, an intermediate expression is pretty-printed, i.e. it is displayed in textual format, according to the definition of the language. WinHIPE provides an alternative, graphical representation for lists and binary trees, 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.

Each expression is automatically visualized according to a number of formatting choices made by the programmer. The programmer can customize textual representations (font, tabulations, indenting, etc.) by demonstration. He/she can also customize all the typographical elements of list and tree graphical representations (colors, lines, backgrounds, etc.) by means of interactive dialogs. Finally, the redex can be highlighted with respect to the rest of the expression by means of two effects: it is colored differently and a thick, horizontal line is drawn above it.

As an example, **¡Error! No se encuentra el origen de la referencia.** contains 12 visualizations generated during an evaluation of the *rebuild* function, defined in Section 1, composed of 348 steps. As mentioned above, functional expressions contain both control and data structures. Therefore, the visualizations in **¡Error! No se encuentra el origen de la referencia.** contain both a fragment of the final value of the expression and suspensions (expressions waiting for their turn to be evaluated, including the redex).

Notice that the third visualization contains a graphical representation of a binary tree (not yet completely computed) and four short lists. Different choices of indenting, forms and colors facilitate discriminating the different parts in such a complex expression. Also notice that the second and third visualizations give more details of program execution, while the remaining visualizations give a more global view of the algorithmic process.

4. ANIMATION OF EVALUATIONS

Animations are displayed in WinHIPE as a discrete sequence of visualizations. The "animation window" shows either one or two (consecutive) visualizations at a time. In the case of two consecutive visualizations, an arrow is displayed in between the visualizations and it is labeled so that the evaluation actions that led from the former to the later expression are identified.

Fig. 2 shows the animation window containing the third visualization of the evaluation given in the previous section. Only one expression has been simultaneously included due to its large size.

The animation window contains a 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 visualizations relative to the total number of visualizations is also displayed, as is the animation speed.

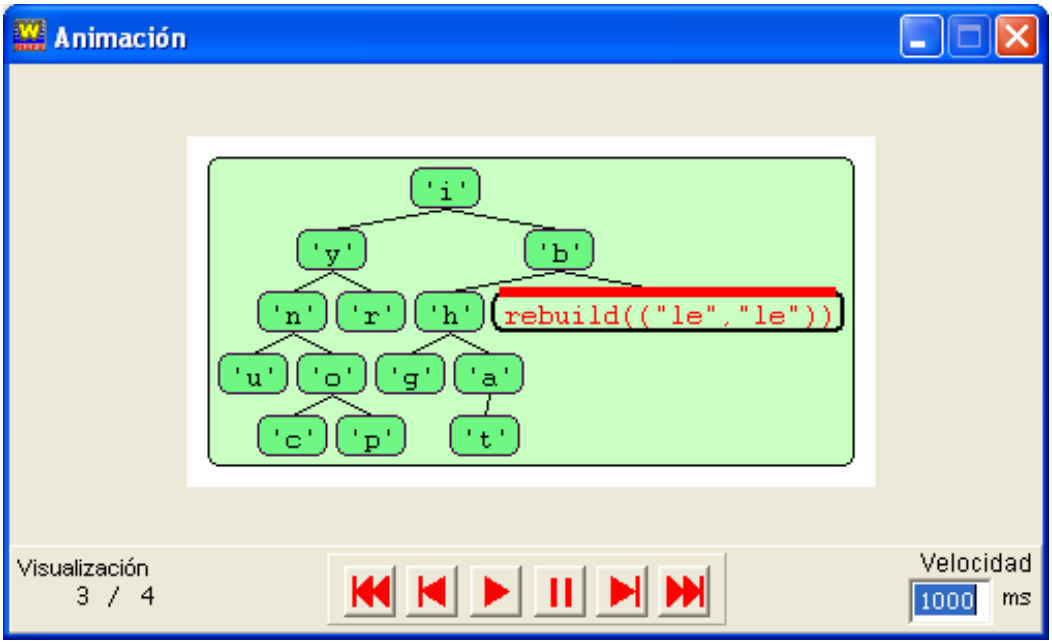

Fig. 2. The animation window.

4.1 Animations as Documents

Our aim has been to allow the construction of animations with low workload for the user, while being able to make them valuable. We use a clear and simple framework for constructing animations based on a metaphor of office documents. An important feature of office applications is their facility to produce documents without the necessity to learn any language. We consider animations as "documents" delivered by our programming environment. Following the document metaphor, animations are easy to construct and modify with a graphical user interface, and the user also is able to store and retrieve them by means of an atomic action provided by the environment menu system.

4.2 Construction of Animations

The process to create an animation consists of two steps [25], generation and selection of visualizations that will be played in the animation. In both steps the user only has to click menu options and checkboxes.

In the first step, an expression e_1 is typed and evaluated. The user controls the evaluation using the actions he/she considers most appropriate (a_1, …, a_{n-1}), thereby yielding intermediate expressions (e_2, …, e_n). For each expression, a visualization is automatically generated as a side-effect (v_1, …, v_n). This sequence of visualizations, separated by labeled arrows that denote the actions performed (as shown in Sections 2 and 3), is displayed in the "visualization window".

In the second step, the user must select the subset of those visualizations that will form the animation (v'_1, …, v'_m where $m \leq n$). Each original visualization v_i has an adjacent check box (in the visualization 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 criterion).

Once these steps have been performed, the user can play the animation. This can be made in two ways: either playing it within the programming environment (using the "animation window") or after generating a dynamic web page (as explained below in Section 5). Both kinds of animations have different aims. The first format is typically used for animations of programs in progress, whereas the second one is used for animations of finished programs, thereby allowing the user to complement them with code and explanations, i.e. for documentation.

One factor that must be taken into account to keep consistency on 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 several visualizations are omitted. For instance, three expressions e_1, e_2 and e_3 that are derived by applying the one-step rewriting action are 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 necessary steps. A simple algebra is used to maintain consistency in arrows.

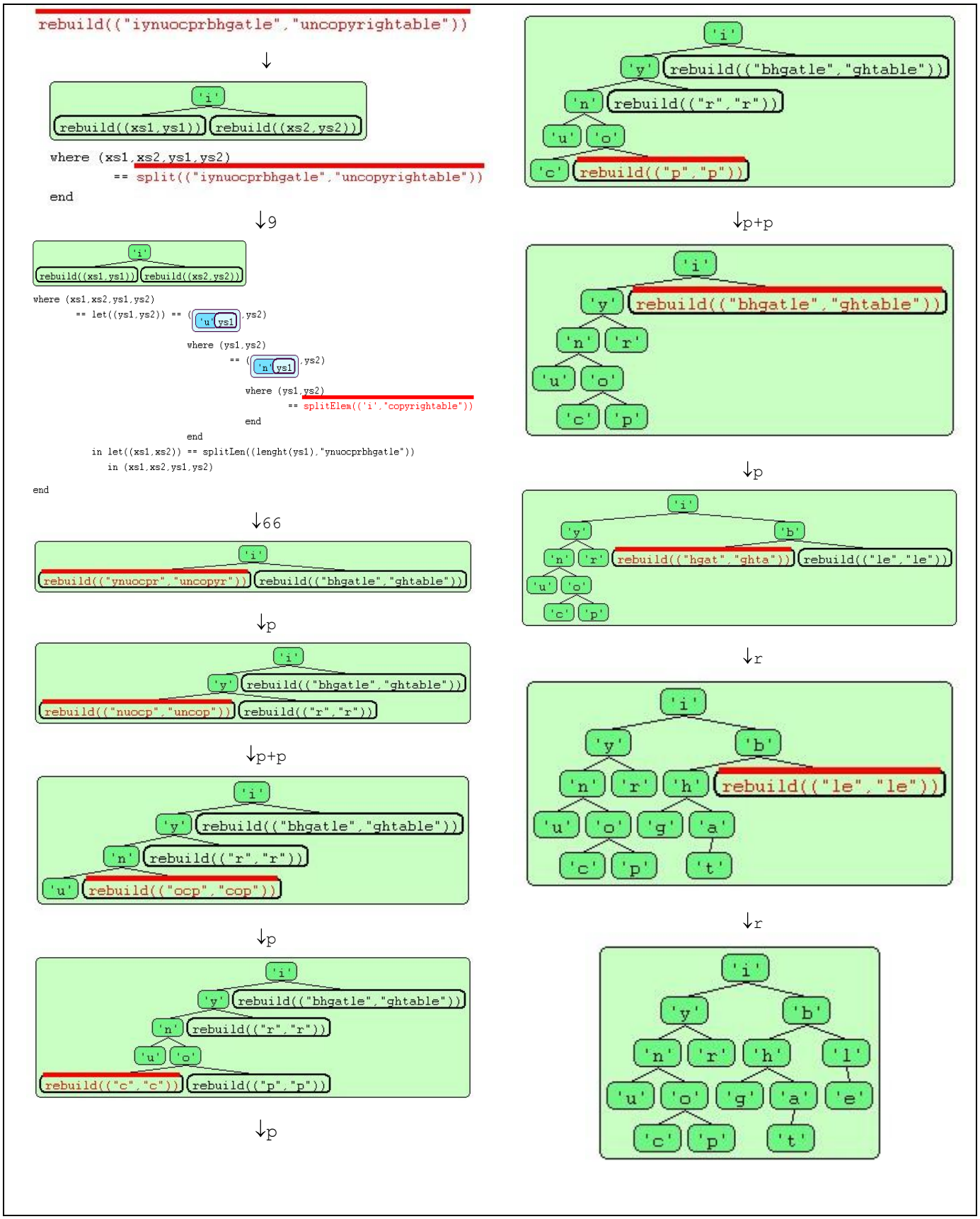

Fig. 3. Visualizations generated during an evaluation of rebuild.

5. EXPLANATION OF EVALUATIONS

In addition to the arrow representation of the steps of an evaluation, WinHIPE allows enriching web animations with textual explanations that help in understanding program behaviour. Textual contents consist in three parts: the source code of the program, and two descriptions about the program. Both descriptions can be used for different purposes. In a professional scope, they can be used for documentation, e.g. to document program tests, describing the test objective and its results. Most commonly, textual descriptions can be used in an academic scope to state a problem and to describe the algorithm developed to solve it. In particular, teachers may use them to support lessons and students either to document assignments or to reinforce self-study.

All the information regarding an animation, i.e. textual explanations, source code of the program, and animation, can be integrated into a web page [19]. The animation is also based on a video player interface. Web pages have the advantage of being platform independent, easy to publish and use.

We have designed four different formats of presentation of these web pages. One format uses a plain web page with local links (see Fig. 4 for the length of a list). Two additional formats are inspired by general principles of web pages usability requirements. The fourth format represents a decentralized solution, where the different parts of the animation are displayed in different windows.

WinHIPE allows friendly creation and maintenance of web animations by means of simple dialogs. It uses an underlying XML-based representation that is transformed into web pages on demand.

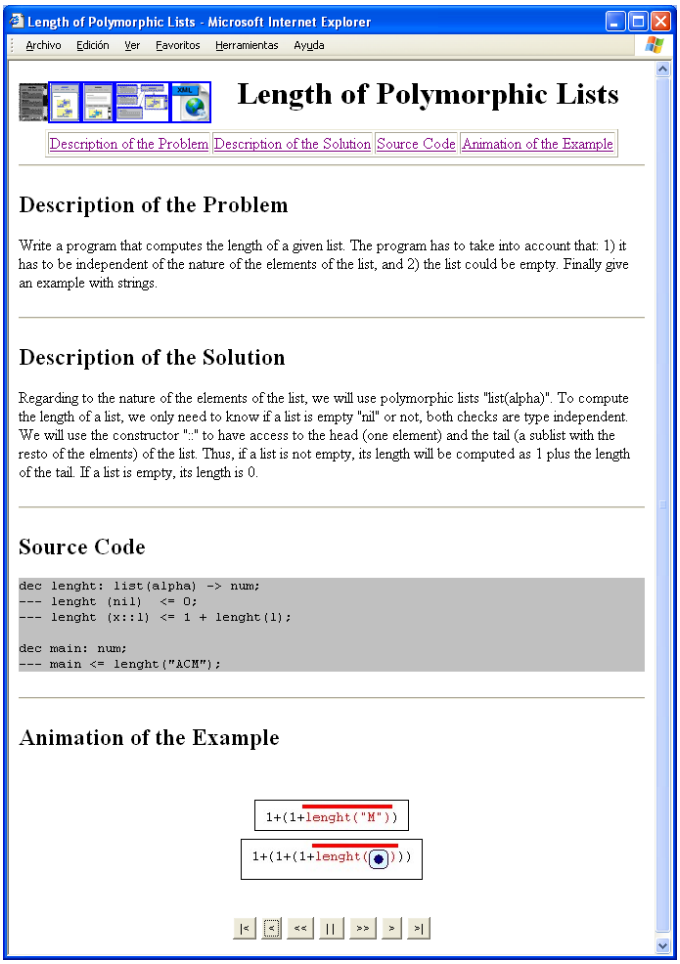

Fig. 4. A web animation presented with plain format.

6. HANDLING LARGE SIZES

The facilities described in the previous sections are effective for small expressions, short evaluations and a small number of animations. When their respective size, length or number grows substantially, they cannot be handled effectively. As a consequence, the IDE must provide more advanced facilities, which are described in this section.

6.1 Large Expressions

Some programs may produce large intermediate expressions during evaluation. This can be due to the program itself, to the computational process being long (e.g. in multiple recursive functions) or to the presence of large data structures.

We have designed a “context+focus” or “fish-eye view” technique [24] that shows the parts most relevant to an understanding of the current stage of evaluation. It produces a simplified visualization of the current intermediate expression that provides a balance between displaying the whole expression and only showing the subexpression containing the redex. As a consequence, the current focus of interest is displayed while also displaying the context where it has been yielded. The amount of information to be shown of any expression is controlled by the user via a natural number, and can be modified at any moment.

For instance, consider the following expression:

```
if 4=0 then 1 else 4*fact(4-1)
```

It can be successively simplified on user's demand, resulting in the following alternative displays:

```
if 4=0 then 1 else 4*fact(...-...)
if 4=0 then 1 else 4*fact ...
if 4=0 then 1 else ...*...
if 4=0 then ... else ...
```

where each occurrence of elision (i.e. '. . .') represents a hidden subexpression. Notice that the amount of information to hide/display is based on the syntactic structure of the expression.

Fig. 5 and Fig. 6 respectively show the visualizations of another expression example with and without applying fish-eye views. Notice subexpressions in the path from the expression root to the redex are shown, while adjacent subexpressions are elided.

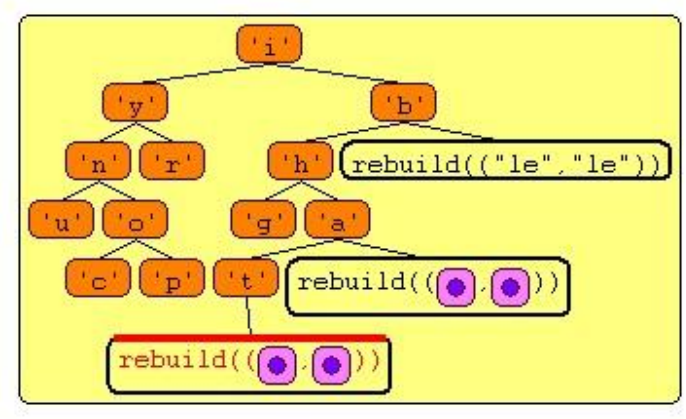

Fig. 5. Visualization of an expression without applying fish-eye views.

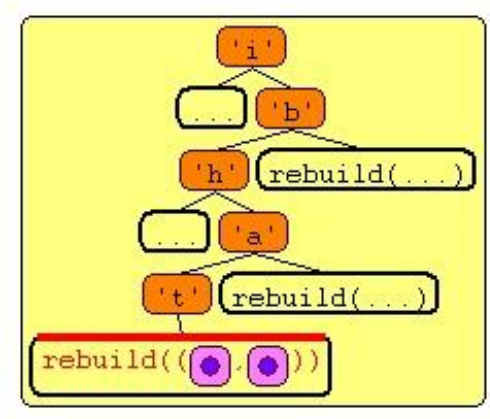

Fig. 6. The same visualization after applying fish-eye views.

6.2 Large Number of Expressions

The evaluation of an expression can produce numerous intermediate expressions and therefore numerous visualizations. In this case, it is very difficult for the programmer to choose the visualizations that will form the animation. On the one hand, the programmer must have a global view of the evaluation to produce a meaningful animation. On the other hand, the programmer probably wants to view a particular visualization in detail to decide if it is worth to select it for the animation.

We have designed R-Zoom [21], an information visualization technique based on focus+context principles. R-Zoom simultaneously provides a global view of the evaluation of an expression and a detailed view of a particular visualization selected by the programmer.

R-Zoom places the visualizations in a special window in a left to right, top-down fashion. Visualizations are displayed in reduced size and keeping their relative proportions of height and breadth. The user can also select any visualization (the focus), which is then displayed in full size. The visualizations in the same row as the focus are distributed in two rows so that minimum usage of screen space is made. The technique allows flexible interaction to the user, including customizing the window size or the maximum size of reduced visualizations. Fig. 7 shows an example of the technique.

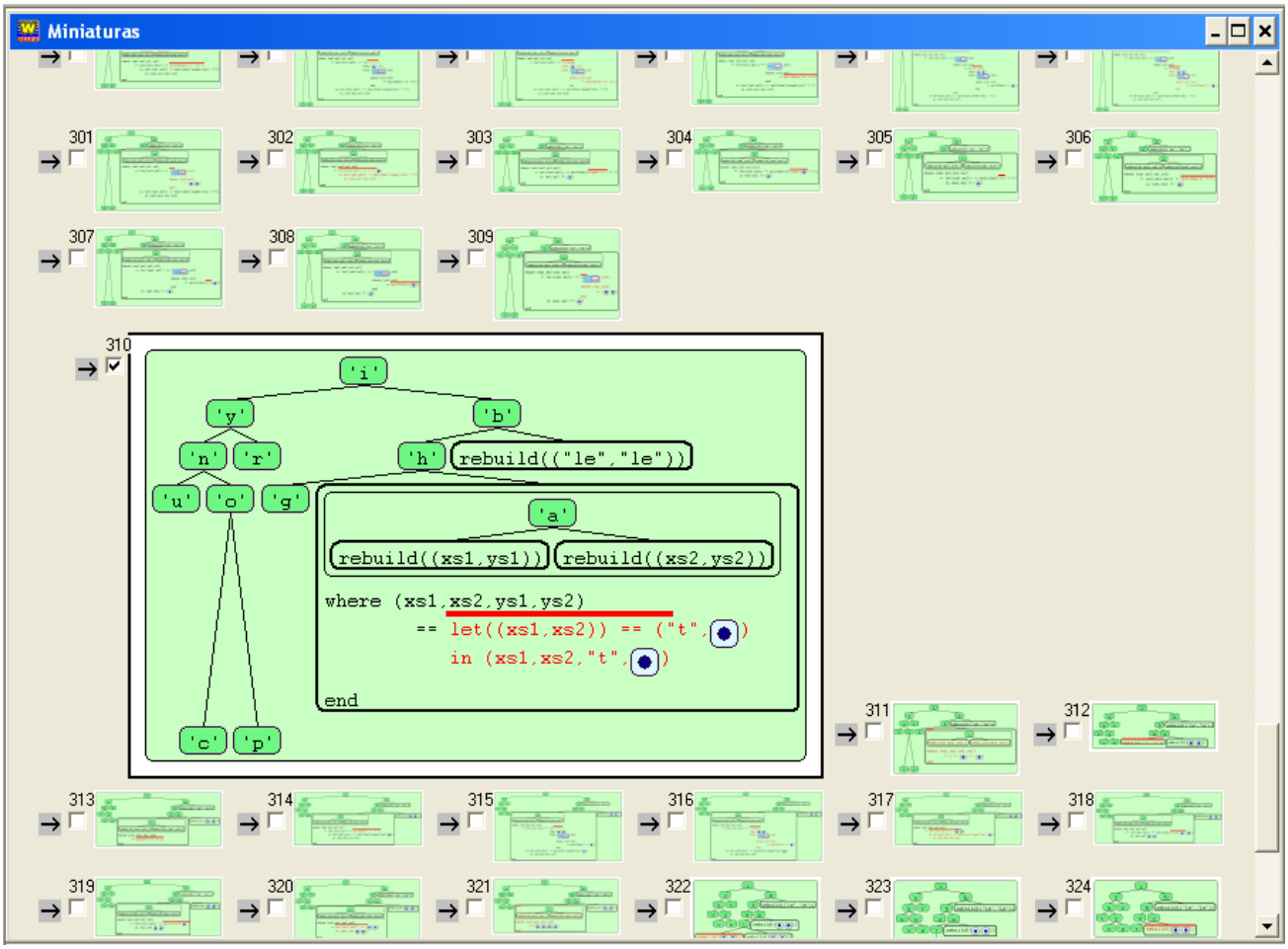

Fig. 7. A window displaying visualizations with the R-Zoom interaction technique.

6.3 Collections of Web Animations

Typically, an instructor will construct more than one or two animations. Therefore he/she will wish to take advantage of this effort by organizing and reusing the animations.

WinHIPE provides an interface to manage collections of web animations [19]. A collection is formed by a set of web animations about a common theme: a course, a class of problems, etc. The user may generate as many independent collections as he/she wishes. A given collection is organized hierarchically. Management of a collection consists in managing its hierarchical structure, web animations, and their look. A web animation is handled in a collection as a single document.

7. EXPERIENCE

WinHIPE has been used in a programming languages course since 2001. It is a course for freshmen [23] with a strong component of functional programming. The functional language is taught to illustrate the functional paradigm and also as the leading language to describe different programming language constructs.

During these years, we have evaluated WinHIPE. A usability evaluation was conducted during Spring 2004 [11]; attitude towards WinHIPE was positive but some difficulties regarding the animation construction process were detected. As a consequence, we developed R-Zoom, also evaluated during Spring 2005 with excellent results [21]. Finally, we have conducted a pedagogical evaluation during Fall 2005, again with successful results [20].

8. RELATED WORK

Many efforts have been dedicated to go beyond the typical editor-compiler tandem used by programmers; they have resulted in different advanced tools that support the coding tasks. We classify the functional programming literature on these tools into two categories: debugging tools and IDEs.

Debugging tools offer a number of facilities focused on handling the evaluation of expressions, e.g., controlling the evaluation steps, visualizing the trace, or assisting in locating errors. IDEs provide support for different tasks from a more general point of view, such as: edit programs and expressions (with advanced edition facilities), manage projects with multiple files, or provide debugging facilities.

8.1 DEBUGGING TOOLS

There exist two main approaches to debug functional programs: declarative and semantic. The semantic approach is a more human directed method, where locating errors is a task carried out by the programmer and supported by a number of tools that enables browsing the expression evaluation. The declarative (also called algorithmic) approach is an automatic and machine-directed method that, interacting with the programmer, is able to locate errors in the source code.

Semantic debuggers produce and show the evaluation trace. Some of them build the trace using breakpoints, as *Prospero* [18] for the Miranda language, *Hint* [4] for a subset of Haskell, or *Hood* [5] for Haskell. The latter has an extension called *GHood* [16] that provides a graphical representation of the trace.

Freja [12] and *Buddha* [14] are two examples of declarative debuggers for different subsets of the Haskell language. They use the dependency reduction graph to guide the user while debugging. *COOSy* [2] is another declarative debugger for the functional logic programming language Curry, it extends the observations technique used in *Hood* and also produces graphical visualizations of the trace.

Other debuggers allow browsing the redex trail. *Hat* [17], based on a subset of Haskell, generates a trail of reduced redexes and allows browsing it in a graphical way, also it provides both algorithmic and semantic debugging. A trace browser for a lazy functional language [26], designed for semantic debugging, allows browsing the trail of redexes linking them to the source code. Finally, *Visual Miranda* [1], that can be used for semantic debugging, connects each expression with its subexpressions and finally with its result, also showing the pattern matching process.

8.2 IDEs FOR THE FUNCTIONAL MODEL

IDEs commonly integrate a number of tools under the same interface. Let us review the most relevant ones.

CIDER [6] is based on the lazy language Curry. It integrates an editor, program analysis tools, a graphical debugger, and a dependency graph drawing. Execution data are collected in a trail. The debugger supports breakpoints and changing the execution order.

ZStep [9] is a Lisp integrated environment. Its debugger supports execution in both forward and backward direction, evaluating the selected expression and executing until the end. Control of speed execution and trees of function call also are provided. Execution data can be found in a trail. ZStep simultaneously shows the source code and the execution code. Execution errors are located in the same place where the correct values should be located.

DrScheme [3] is based on the Scheme language. It is an educational IDE that allows using four different subsets of the language. Its debugging facilities locate the function call that produces errors. It has a static debugger which, using type inference, can predict potential errors. In addition, it provides program browsing, profiling and testing facilities.

The last environment is called *TERSE* [7]. Properly speaking, it is not a functional program environment, but rather a term rewriting system (which is the basis of functional program execution). It has been developed with Standard ML/NJ, and allows transforming TERSE programs into Standard ML programs. During execution it allows selecting, among all redexes available, the one that will be reduced. Also, it permits to choose the rule to apply and the execution strategy. It shows a global view of the expression, represented as a tree, and a zoomed view of a particular area of it, and also generates rewriting sequences.

9. CONCLUSIONS

In last years, functional programming has come out of the theoretical and academic scope and is currently being used both in industry and in CS education. The main contribution of WinHIPE consists in gathering different tracing and visualization features:

- It provides a flexible set of advance actions based on the term rewriting model. Tracing is based on pure observations and does not change the normal execution process.
- It provides a fine, easily customizable, mixed graphical-textual representation for expressions.
- It includes an effortless framework for constructing and reconstructing discrete animations.
- It facilitates the generation of electronic documentation about the program. It allows integrating the delivered animations with code and free text, and exporting them to the web.
- It provides several techniques to handle large scale in several dimensions: expression size, number of expressions in an evaluation, and number of animations in a collection.

We have successfully used WinHIPE in the classroom for several years in a programming languages course. In addition, we have formally evaluated different parts of the IDE with successful results, too.

WinHIPE is available at http://vido.escet.urjc.es/winhipe/. The evaluation of the *rebuild* algorithm used in this article also can be found at this website in two formats: the full step-by-step evaluation and the simplified evaluation included in Section 3.

In comparison to other systems, WinHIPE is unique as it integrates coding tools with animations in a simple, effortless framework. This feature is valuable for both professional and academic usage.

Many tools were designed during the last decades for program and algorithm visualization. Sami Khuri [8] identifies several key requirements to achieve effectiveness. Some of them were cited above. Besides, WinHIPE supports the following ones: minimizing users' startup time, providing a standard GUI to handle the tools, and including a number of worked examples.

10. ACKNOWLEDGMENTS

The work is supported by the Spanish Ministry of Education and Science under projects TIN2004-07568 and TIN2006-15578-C02.

11. REFERENCES

[1] Auguston, M. and Reinfelds, J. A Visual Miranda Machine. In *Proc. of the Software Education Conference (SRIT-ET'94)*, 1994, 198-203.

[2] Braßel, B., Chitil, O. Hanus, M. and Huch, F. Observing Functional Logic Computations. In B. Jayaraman (ed.), *Proc. of the 6th International Symposium, PADL 2004*, Springer-Verlag, LNCS 3057, 2004, 193-208.

[3] Findler, R.B. DrScheme: A programming environment for Scheme, *Journal of Functional Programming*, 12(2), 2002, 159-182.

[4] Foubister, S.P. and Runciman, C. Techniques for simplifying the visualization of graph reduction. In K. Hammond, D.N. Turner, and P.M. Sansom (eds.), *Functional Programming*, Springer, 1995, 65-77.

[5] Gill, A. Debugging Haskell by observing intermediate data structures. In G. Hutton (ed.), *Proc. of the 2000 ACM SIGPLAN Haskell Workshop*, ENTCS 41, Elsevier, 2000.

[6] Hanus, M. and Koj, J. Cider: An integrated development environment for Curry. In *Functional and (Constraint) Logic Programming, WFPL 2001*, Christian-Albrechts-Universitt zu Kiel, Report No. 2017, 2001, 369-373.

[7] Kawaguchi, N., Sakabe, T. and Inagaki, Y. Terse: Term rewriting support environment. In *Proc. of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*, ACM Press, 1994, 91-100.

[8] Khuri, S. Designing effective algorithm visualizations. In Sutinen, E. (ed.), *Proc. of the 1st Program Visualization Workshop*, University of Joensuu, Finland, 2001, 1-12.

[9] Knuth, D.E. *The Art of Computer Programming, vol. 1, Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997

[10] Lieberman, H. and Fry, C. ZStep95: A reversible, animated source code stepper. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price (eds.), *Software Visualization*, MIT Press, 1998, 277-292.

[11] Medina-Sánchez, M.A., Lázaro-Carrascosa, C.A., Pareja-Flores, C., Urquiza-Fuentes, J. and Velázquez-Iturbide, J.A. Empirical evaluation of usability of animations in a functional programming environment, Universidad Complutense de Madrid, Technical Report 141/04, 2004.

[12] Nilsson, H. and Sparud, J. The evaluation dependence tree as a basis for lazy functional debugging, *Automated Software Engineering*, 4(2), 1997, 121-150.

[13] Pareja-Flores, C., Peña, R., Rubio, C. and Segura, C. Adding traces to a lazy monadic evaluator. In *Computer Aided Systems Theory - EUROCAST 2001*, Springer-Verlag, LNCS 2178, 2001, 627-641.

[14] Oudshorn, M.J., Widjaja, H.and Ellershaw, S.K. Aspects and taxonomy of program visualisation. In P. Eades and K. Zhang (eds.), *Software Visualisation*. World Scientific Publishing, 1996, 3-26.

[15] Pope, B. Buddha: *A Declarative Debugger for Haskell*. PhD thesis, Department of Computer Science, The University of Melbourne, Australia, 1998.

[16] Reinke, C. GHood: Graphical visualisation and animation of Haskell object observations. In R. Hinze (ed.), *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of ENTCS, Elsevier, 2001, 121-149.

[17] Sparud, J. and Runciman, C. Tracing lazy functional computations using redex trails. In H. Glasser, P. Hartel, and H. Kuchen (eds.), *Proc. of the 9th Intl. Symp. on Program- ming Languages, Implementations, Logics and Programs (PLILP'97)*, Springer-Verlag, LNCS 1292, 1997, 291-308.

[18] Taylor, J.P. Presenting the evaluation of lazy functions. PhD thesis, Department of Computer Science, Queen Mary University of London and Westfield College, UK, 1995.

[19] Urquiza-Fuentes, J. and Velázquez-Iturbide, J.Á. Effortless construction and management of program animations on the Web. In R.W.H. Lau, Q. Li, R. Cheung and W. Liu (eds.), *Advances in Web-Based Learning – ICWL 2005*, Springer-Verlag, LNCS 3583, 2005, 163-173.

[20] Urquiza-Fuentes, J. and Velázquez-Iturbide, J.Á. An evaluation of the effortless approach to build algorithm animations with WinHIPE, In *Proc. of the 4th Program Visualization Workshop (PVW 2006),* 2006, 29-33.

[21] Urquiza-Fuentes, J. Velázquez-Iturbide, J.Á. and Lázaro-Carrascosa, C. Design and evaluation of R-Zoom, a new focus+context visualization technique. In *Proc. of the VII Congreso Internacional de Interacción Persona-Ordenador (INTERACCIÓN 2006),* 2006, 79-88.

[22] Velázquez-Iturbide, J.Á. Improving functional programming environments for education. In M.D. Brouwer-Janse and T.L. Harrington (eds.), *Man-Machine Communication for Educational Systems Design*, Springer-Verlag, 1994, 325-332.

[23] Velázquez-Iturbide, J.Á. A programming languages course for freshmen. In *Proc. of the 10th Annual Conference on Innovation and Technology in Computer Science Education*, ACM Press, 2005, 271-275.

[24] Velázquez-Iturbide, J.Á. Principled design of logical fisheye views of functional expressions, *ACM SIGPLAN Notices*, 41(8), 2006, 34-43.

[25] Velázquez-Iturbide, J.Á., Pareja-Flores, C. and Urquiza-Fuentes, J. An approach to effortless construction of program animations, *Computers & Education*, in press.

[26] Watson, R. and Salzman, E. A trace browser for a lazy functional language. In *Proc. of the 20th Australian Computer Science Conference*, 1997, 356–36