A Survey of Program Visualizations for the Functional Paradigm

Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide
Universidad Rey Juan Carlos, Madrid, Spain

{j.urquiza,a.velazquez}@escet.urjc.es

1 Introduction

One of the definitions for visualization is to give a visible appearance to something, making it easier to understand. In Price et al. (1998), *software visualization* is defined as "the use of crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software". *Program visualization* is defined as "the visualization of actual program code or data structures in either static or dynamic form".

We focus on the functional programming paradigm here. We study crafts used to visualize functional program code and data structures. The study has been done over sixteen systems. These systems can be categorized in multiple ways (Price et al., 1993; Myers, 1986; Brown, 1998). Although we do not want to make a new taxonomy, we differentiate among integrated development environmens, debuggers, teaching systems and visualization systems.

We have made a compilation of information about functional visualization systems (this information is very dispersed). In general, most systems are partial solutions to the main problem; the visualization of functional programs. Our ultimate aim is to find a more general solution.

In section 2 particular aspects of the functional paradigm are introduced. Section 3 briefly describes the systems we studied. The visualization of each particular aspect identified is presented in section 4. In section 5 the evaluation of some systems is described. Finally we draw our conclusions in section 6.

2 Features of the Functional Paradigm

The functional programming paradigm has some particular features that are needed to be visualized to understand the execution of a program. In functional programming the source code of a program is formed of bodies of functions. Each function is a set of rules. In the following, a program to compute the addition of elements in a list is shown:

```
fun sumlist list(int) -> int
    | sumlist([]) = 0
    | sunlist(head::rest) = head + sumlist(rest);
```

The execution of a functional program begins with an expression in which some of the functions of the program are called. Each execution step is a rewriting step applied on an expression, and its result will be another expression. The following are all the rewriting steps of the execution of `sumlist([3,5,2])`.

```
[sumlist([3,5,2])] ⇒ sumlist(3::[5,2])
3 + [sumlist([5,2])] ⇒ sumlist(5::[2])
3 + 5 + [sumlist([2])] ⇒ sumlist(2::[])
3 + 5 + 2 + [sumlist([])] ⇒ sumlist([])
3 + 5 + [2 + 0]
3 + [5 + 2]
[3 + 7]
10
```

As shown in the previous example, the rewriting steps are applied to parts of the whole expression (framed code in the example). For each step, the next subexpression to rewrite (or reduce) is called the *redex*. Each rewriting step is related to the evaluation of a (sub)expression, and each evaluation gives (sub)results. Therefore, important aspects to visualize are the evaluation of (sub)expressions, its corresponding redexes and the (sub)results obtained.

Another feature to visualize is the order in which function calls are executed. Two important details are the evaluation of parameters and how pattern matching is used to select the appropiate rule in the body of the function to be applied.

The environment of variables (also called contour) fixes their values, so it will be important to clearly visualize those environments. Moreover, if complex data structures are used in a program, as lists or trees, it will be desirable to work with special visualizations for them.

There are two ways (also called strategies) of executing a functional program. The previous example shows eager execution. Alternatively, lazy execution evaluates an expression only when necessary. For example, the expression `fact(4+2)` is reduced to `if((4+2)=1) then 1 else (4+2)*fact((4+2)-1)`. The function `fact` is applied before evaluating the argument `4+2`. In order to avoid inefficiency, the subexpression `4+2` does not appear 3 times, but it is unique and shared among the three places. Therefore, expression sharing is an important feature of lazy evaluation to visualize.

3 Systems Studied

Normally, the features to be visualized and the way this is achieved depend on the class of the system being used. We have therefore classified systems into four categories: integrated development environments, debuggers, teaching systems and visualizing systems.

Integrated development environments use to integrate a number of tools under the same interface. *CIDER* (Hanus and Koj, 2001) uses the lazy language Curry. It integrates edition, program analysis tools, a graphical debugger, and a dependency graph drawing. Execution data are collected in a trail. Its debugger supports breakpoints and changing the execution direction.

WinHIPE (Naharro-Berrocal et al., 2002) uses the language Hope. Programs are executed under the eager stategy. It shows the set of expressions resulting during an evaluation. Its debugger provides general options such as executing one or n steps, evaluating to the next breakpoint, evaluating the redex or backtracking to a previous expression. It shows graphically lists and trees and supports a wide range of customizations, including graphical format, typographic characteristics and subexpression visibility. From the static visualizations generated, it allows building animations that can be saved and loaded for educational use.

ZStep (Lieberman and Fry, 1998) is a Lisp integrated environment. Its debugger supports execution in both directions, evaluating the selected expression and executing until the end. Speed execution control and a tree function calls are also provided. Execution data can again 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.

The last environment is called *TERSE* (Kawaguchi et al., 1994). 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 vision of the expression, represented as a tree, and a zoomed vision of a particular area of it, and also generates rewriting sequences.

Debuggers adapted to functional programming are commonly called *steppers*. We have studied six debuggers. *Freja* (Nilsson and Sparud, 1997) and *Buddha* (Pope, 1998) use subsets of the language Haskell. They are algorithmic debuggers, and use the dependency reduction

graph to guide the user while debugging.

Hat (Sparud and Runciman, 1997) also supports a subset of Haskell. It generates a trail of reduced redexes, allowing to browse it in a graphical way.

Hood (Gill, 2000) uses the whole Haskell language. To visualize the execution, the source code must be modified, inserting calls to the visualization system where a visualization is needed (either a function or a data structure). The visualization is obtained as a result of the execution of the program.

Prospero (Taylor, 1995) and *Hint* (Foubister and Runciman, 1995) are very similar systems. *Prospero* uses the language Miranda and *Hint* uses a subset of Haskell. Both generate and use a trail. While debugging, they allow using breakpoints, but do not allow changing the direction of the execution.

Teaching systems usually focus the user's attention on particular aspects of programming languages in order to gain understanding. *Evaltrace* (Touretzky and Lee, 1992) uses Lisp. Its visualizations are documents generated with LaTeX. This system is focused on differentiating between applying and evaluating actions. It also visualizes macros and side effects. It is integrated into a programming environment.

KIEL (Berghammer and Milanese, 2001) works with a subset of the language Standard ML, were only first order functions are allowed. It allows changing the execution strategy and executing a number of rewriting steps.

DrScheme (Findler, 2002) uses the language Scheme. It allows using four subsets of the language. When a error is produced, *DrScheme* locates the function call that produced it. It has an static debugger which, using type inference, can predict potential errors.

KAESTLE & FooScape (Boecker et al., 1986) work with Lisp too, but they only visualize data structures and function call graphs. They can generate snapshots of each visualization and sequences of them. They use trails generated by the *FranzLisp* system and are also integrated in a programming environment.

We have studied two *visualization systems*. *GHood* (Reinke, 2001), which graphically shows the observations made by *Hood*. It has typical VCR controls and possible EPS output of its graphs. It generates animations where the speed can be controlled. *Visual Miranda* (Auguston and Reinfelds, 1994) uses the language Miranda. It generates a textual trail, but it can be shown in a graphical way.

4 Partial Visualizations of Functional Programs

In this section, we describe how the systems cited above support the visualization of the different aspects (partial visualizations) of functional programming mentioned in the second section. Four partial visualizations and some existing combinations of them are considered.

4.1 (Sub)expressions, Redexes and (Sub)results

A functional expression has a tree structure, so all systems work internally with expressions represented as trees (or directed graphs in lazy functional languages). Many systems also visualize expressions as trees (see Fig. 1). These are *CIDER*, *KIEL* (which allows interacting directly with the abstract syntax tree), *Prospero*, *Hint* and *TERSE* (which gives a different representation to constructors, variables and functions).

When an expression is large, its visualization can be confusing. Therefore, some tools make a compact version of the expression (see Fig. 2). *Prospero* and *Hint* allow applying filters (spatial and temporal ones). Moreover, *Hint* provides a metalanguage to define those filters. *TERSE* transforms subtrees into tree nodes. *WinHIPE* elides the visualization of less important subexpressions by applying fish-eye views. *ZStep* allows filtering expressions by defining conditions. *Evaltrace* compacts trivial evaluation steps; for instance, in the evaluation of `sumlist([])`, the evaluation of the parameter `[]` into itself is trivial.

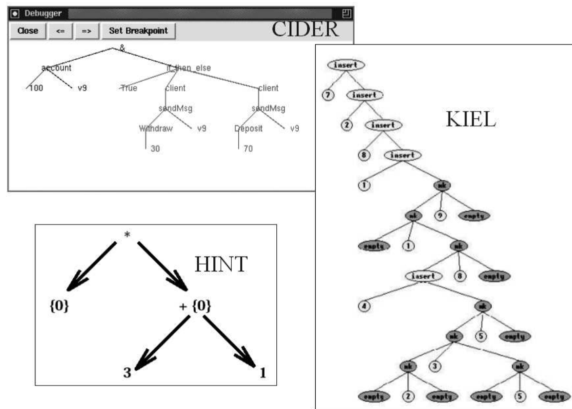

Figure 1: Expressions represented as trees

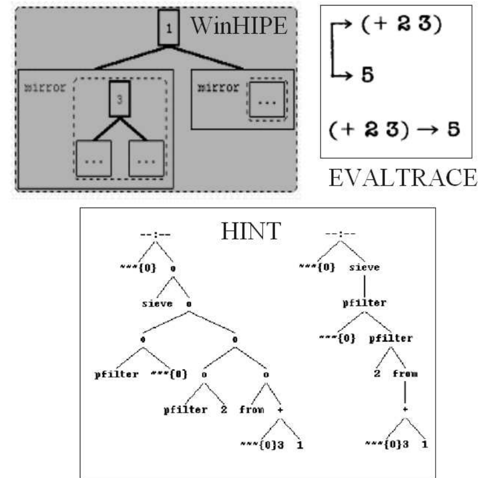

Figure 2: Compression of large expressions

All systems except *Hood & GHood* and *Evaltrace* highlight the redex (see Fig. 3). *Hood & GHood* only show the value of a variable marked as observable.

Hat, *Freja*, *Buddha* and *Evaltrace* connect each subexpresion and the result of its evaluation. *DrScheme* shows simultaneously the current expression, its reduction and the function definition used in the rewriting step. *Visual Miranda* connects the expression with its subexpressions and finally with its result (see Fig. 4).

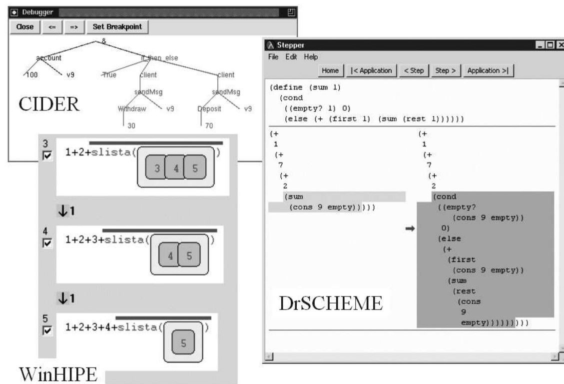

Figure 3: Redex highlighting

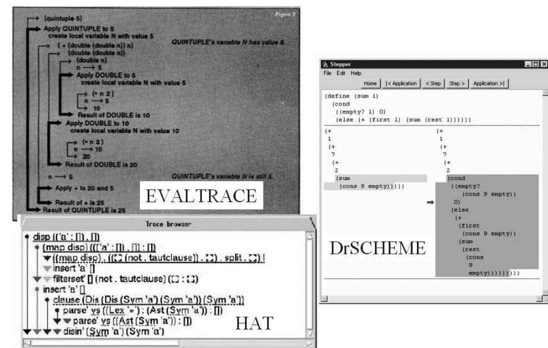

Figure 4: Connecting expressions & results

4.2 Function Calls, Function Application and Pattern Matching

The visualization of function calls is carried out by *KAESTLE & FooScape* by drawing a static flow diagram where functions are represented as ellipses, and function calls are represented as arcs from the caller to the callee (see Fig. 5). The user can choose to hide calls from a function, compacting the diagram. It allows a dynamic visualization too, by highlighting functions that have not finished their execution. *Evaltrace* focuses on differentiating evaluation and application of functions to their arguments. This is done by drawing different lines while evaluating the parameters of a function call or while applying the function: a thin line and a thick line respectively. Also, lines connect the evaluation of parameters with the function application and the result (see Fig. 4). *Visual Miranda* shows the full pattern matching process, trying to match the rule and detailing if the match fails or succeeds (see Fig. 4). Finally, *Hood* allows showing function calls and their result if the observation is located in the definition of the function (see Fig. 6).

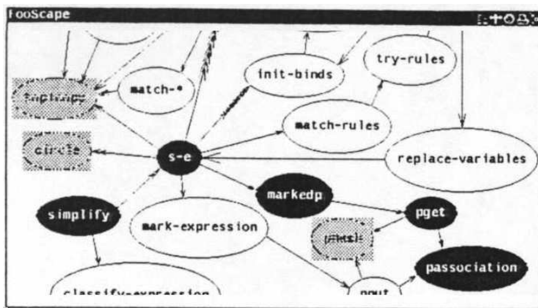

Figure 5: A function call graph in FooScape

```
last = observe "last" last'
last' (x:xs) = last xs
last' [x] = x
---last
   { \ (_ : _ : []) -> trhow <Exception>
   , \  ( _ : []) -> trhow <Exception>
   , \  [] -> trhow <Exception>
   }
```

Figure 6: A *Hood* observation

Function calls may also be used as an auxiliary element, even though they do not play an important element in visualizations. Thus, *WinHIPE* uses function calls as breakpoints, but the visualization displays the current expression as a whole.

4.3 Variables and Data Structures

The contour of variables and their values is visualized in several ways. All the systems show variable values. *Hood* is a special case, because it shows values in a particular location of the source code, so in the body of a function, the user can choose to visualize a variable in a rule and not in others. *Evaltrace* identifies a contour by connecting the begining and the end with a thick line. If this line is solid, then the global contour is the parent of the present contour. Otherwise, there is a local variable definition and the parent contour is the closest enclosing one. *DrScheme* connects variables and their occurences with lines (see Fig. 7). *Visual Miranda* shows the value for each variable before evaluating a (sub)expression (see Fig. 4).

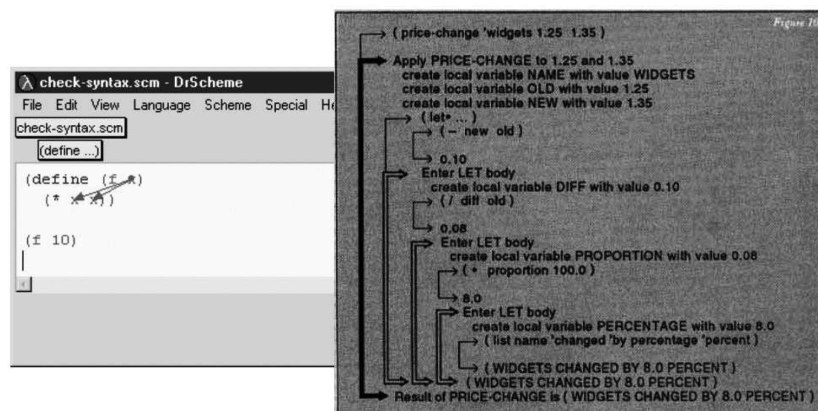

Figure 7: Contour visualization with *DrScheme* and *Evaltrace*

Only two systems allow alternative visualizations of complex structures (see Fig. 8). *WinHIPE* permits to customize the visualization of tree and lists, by identifying constructors used (the predefined constructors of the language for lists; `Node` and `Empty` for binary trees), and assigning to it the corresponding shapes, line styles, background and foreground colours and dimensions defined by the user configuration. *KAESTLE & FooScape* visualize lists by drawing their elements into squares, putting one after another or connecting them with arcs as needed. It allows modifying the layout of each list visualized and its contents.

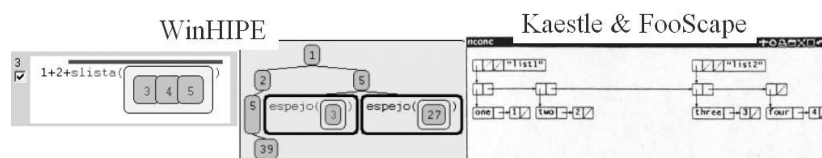

Figure 8: Alternative representations for complex data structures

4.4 Subexpression Sharing in Lazy Evaluation

Systems supporting lazy evaluation should visualize shared subexpressions. *CIDER* and *Hat* do not visualize shared subexpressions until they are reduced, then they highlight all occurences of the shared subexpression. *Prospero* and *Hint* (see Fig. 9) only visualize once a shared subexpression, being connected the rest of occurrences to the first one by arcs or labels.

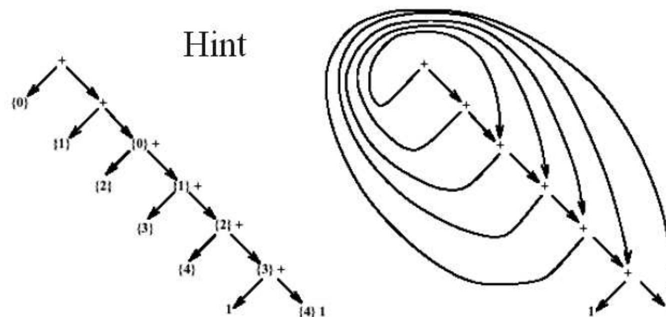

Figure 9: Visualization of shared subexpressions in lazy evaluation

4.5 Combining Partial Visualizations

Some systems combine some of the previous partial visualizations. While *Hood & GHood* are able to show values of variables and results of function calls, and *KAESTLE & FooScape* display a function call graph and current state of lists in the program, the rest of systems tend to blur the separation of code and data in functional programs. *WinHIPE*, *DrScheme* and *Visual Miranda* display values and data structures integrated into expressions. *Evaltrace* does it too but in a different way, by integrating values of variables into a pretty-printed textual description of the execution of the program. In addition *DrScheme* and *Evaltrace* show the contour of variables, and *Visual Miranda* is able to display the full pattern matching process.

5 Systems Evaluation

We have only found two documented experimental evaluations of systems. The first (Chitil et al., 2001) is a comparative study of three systems: *Freja*, *Hat* and *Hood*. The study is focused on their tracing and debugging facilities. A number of criteria are evaluated for each system: readability of expressions, the process of locating an error, redexes and language constructs, and modification of the program. This study identifies strengths and weakness of each system and then suggest how the systems can be improved.

The second documented evaluation (Medina-Sánchez et al., 2004) is of the *WinHIPE* environment. This evaluation is focused on effortlessness and usability of animations and their construction process. The experiment was done with students and its results show that animations are easy to use, its construction process is easy to learn, and are understood as a help to complete other tasks, such as debugging or program understanding.

There are two more evaluations but they are documented in a rather informal way. In the section 6 of Findler (2002) some experience with *DrScheme* is briefly described and in section 5 of Reinke (2001) some details are shown about experience with *GHood*.

6 Conclusions

We have given a survey of visualizations of functional programs provided by sixteen different systems. In order to make the exposition more meaningful, we have given a two dimension classification. On the one hand, we have classified the systems into four categories (programming environments, debuggers, teaching systems, and visualization systems). On the other

hand, we have considered the most important partial views provided about functional programs (expression evaluation, function calls, values, and subexpression sharing). Our selection of these dimensions has been pragmatical: we do not pretend these dimensions to be the most important ones, but we found them especially clarifying to us. In Price et al.'s taxonomy, the corresponding categories are program (B.1) and purpose (F.1).

In spite of this variety of visualizations, none provides comprehensive visualizations, with multiple views of all the aspects. There are several systems covering several features of functional programming, more comprehensive visualizations are still lacking. We advocate for a comprehensive approach that would make use of solutions given by current systems. A comprehensive approach could offer current partial views, but it would also offer more powerful and flexible visualizations on the different features of functional programs.

Such a comprehensive visualization still has to be designed. However, notice that the first identified feature, namely expression evaluation, is the basic element of the functional paradigm. Consequently, it should be the basis if the new visualization. Two other features (function calls and values) are partial views that mimic our understanding of program execution derived from the imperative paradigm. Therefore, they should be integrated in the expression model. Finally, subexpression sharing is a particular and important aspect of lazy evaluation.

Acknowledgements

This work has been supported by projects GCO-2003-11 of the Universidad Rey Juan Carlos and TIN2004-07568 of the Ministerio de Educación y Ciencia.

References

M. Auguston and J. Reinfelds. A Visual Miranda Machine. In *Proceedings of the Software Education Conference (SRIT-ET'94)*, pages 198–203. IEEE Computer Society Press, 1994.

R. Berghammer and U. Milanese. Kiel - a computer system for visualizing the execution of functional programs. In *Functional and (Constraint) Logic Programming, WFLP 2001*, pages 365–368, Christian-Albrechts-Universitt zu Kiel, 2001. Report No. 2017.

H.D. Boecker, G. Fisher, and H. Nieper. The enhancement of understanding through visual representations. In *Proceedings of the ACM SIGCHI'86 Conference on Human Factors in Computing*, pages 44–50. ACM Press, 1986.

M.H. Brown. A taxonomy of algorithm animation displays. In J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 35–42. MIT Press, 1998.

O. Chitil, C. Ruciman, and M. Wallace. Freja, Hat and Hood - A comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Selected Papers*, volume 2011 of *LNCS*, pages 176–193. Springer, 2001.

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

S.P. Foubister and C. Runciman. Techniques for simplifying the visualization of graph reduction. In K. Hammond, D.N. Turner, and P.M. Sansom, editors, *Functional Programming*, pages 65–77. Springer, 1995.

A. Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *ENTCS*. Elsevier, 2000.

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

N. Kawaguchi, T. Sakabe, and Y. Inagaki. Terse: Term rewriting support environment. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*, pages 91–100. ACM Press, 1994.

H. Lieberman and C. Fry. ZStep95: A reversible, animated source code stepper. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 277–292. MIT Press, 1998.

M.Á. Medina-Sánchez, C.A. Lázaro-Carrascosa, C. Pareja-Flores, J. Urquiza-Fuentes, and J.Á. Velázquez Iturbide. Empirical evaluation of usability of animations in a functional programming environment. Technical report, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Madrid, Spain, 2004. Ref. 141/04.

B. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceeding of the ACM SIGCHI'86 Conference on Human Factors on Computing Systems*, pages 59–66. ACM Press, 1986.

F. Naharro-Berrocal, C. Pareja-Flores, J. Urquiza-Fuentes, J.Á. Velázquez-Iturbide, and F. Gortázar-Bellas. Redesigning the animation capabilities of a functional programming environment under an educational framework. In M. Ben-Ari, editor, *Proceedings of the Second Program Visualization Workshop*, pages 60–69, University of Aarhus, Department of Computer Science, 2002. DAIMI PB - 567.

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

B. Pope. *Buddha - A Declarative Debbuger for Haskell.* PhD thesis, Department of Computer Science, The University of Melbourne, Australia, June 1998.

B.A. Price, R. Baecker, and I. Small. An introduction to software visualization. In J. T. Stasko, J. Domingue, M. H. Brown, and B.A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 3–27. MIT Press, 1998.

B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualisation. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

C. Reinke. GHood: Graphical visualisation and animation of Haskell object observations. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*, pages 121–149. Elsevier Science, 2001.

J. Sparud and C. Runciman. Tracing lazy functionals computations using redex trails. In H. Glasser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, volume 1292 of *LNCS*, pages 291–308. Springer, 1997.

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

D.S. Touretzky and P. Lee. Visualizing evaluation in applicative languages. *Communications of the ACM*, 35(10):49–59, October 1992.