Program Visualization for the Functional Paradigm

J. Urquiza-Fuentes, J.A. Velázquez-Iturbide
Universidad Rey Juan Carlos. Móstoles. Spain.

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

1 Introduction

One of the definitions for visualization is to give a visible appearance to something that has not it, thus it is 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 efective use of computer software". Also, *program visualization* is defined as "the visualization of actual program code or data structures in either static or dynamic form".

We focus on 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 system.

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

In section 2 particular aspects of the functional paradigm are introduced. In section 3 systems will be briefly described. The visualization of each particular aspect identified will be studied in section 4. Finally we draw our conclusions in section 5.

2 Features of the functional paradigm

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 one is a set rules. 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 wich some of the functions of the program is called. Each execution step is a rewriting step applied on an expression, and its result will be another expression. Here is seen 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 it is seen in the previous example, the rewriting steps could be aplied 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. An important aspect to visualize will be the evaluation of (sub)expressions, its corresponding redexes and the (sub)results obtained.

Other feature to visualize is the way function calls are executed. The evaluation of its parameters and how pattern matching is used to select the appropiate rule in the body of the function.

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

There are two ways (also called strategies) of executing a functional program: as the previous example, which is called an eager execution; or with a lazy execution, where an expression is only evaluated if it is needed to be done, for instance, the expression: `fact(4+2)` would be reduced to: `if((4+2)=1) then 1 else (4+2)*fact((4+2)-1);`. Instead of evaluating the subexpression `4+2`, the body of `fact` has been applied, so `4+2` will appear 3 times, this another feature of functional programming to be visualized: expression sharing when lazy evaluation is used.

3 Systems studied

Normally, the features visualized and the way it is achieved, depend on the class of the system being used, so we have classified systems into four categories: integrated development environments, debuggers, teaching systems and visualizing systems.

Integrated development environments use to have a number of tools integrated under the same interface. *CIDER* (Hanus and Koj, 2001) uses the lazy language Curry; it integrates edition, program analisys tools, a graphical debugger, and a dependency graph drawing. Its execution data are collected in a trail. When debugging it allows to use breakpoints and changes the exeution's direction. *WinHIPE* (F. Naharro-Berrocal et al., 2002) uses the language Hope. Programs are executed in an eager way. It allows editing and general options on debugging as execute one or n steps, to the next breakpoint, evaluate the redex or go back. It also builds animations from the static visualizations generated. *ZStep* (Lieberman and Fry, 1998) is a Lisp integrated environment. Its debugger allows two directions execution, evaluate the selected expression and execute until the end; also speed execution control and a tree function calls are provided. Again execution data can be found in a trail. *ZStep* shows simultaneously 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), although it is not a functional program environment; it is a rewriting term system (which is the basis of functionl program execution). It has been developed with SML/NJ (a functional language), and allows to transform *TERSE* programs in SML programs. During the execution it allows to select among all redexes available, which one will be reduced. Also, it permits to choose the rule to apply and execution strategy. It shows a global vision of the tree and a zoomed vision of a particular area of it, also it generates rewriting secuences

As debuggers, we have studied six systems. *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) uses a subset of Haskell too. 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 (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 not to change the of the execution.

Teaching systems, usually focus its attention to particular aspects. *Evaltrace* (Touretzky and Lee, 1992) uses Lisp, its visualizations are documents generated with LaTeX. This system is focused in differentiate between aplying and evaluating actions, also it visualizes macros and side effects. It is integrated in a programming environment. *KIEL* (Berghammer and Milanese, 2001) works with a subset of the language *Standard ML*. It only uses first order functions. It allows to change the execution strategy and execute a number of rewriting steps. *DrScheme* (Findler, 2002) uses the language Scheme. It allows to use 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 calls graphs. They can generate snapshots of each visualization and secuences of them. They use trails generated by the application *FranzLisp* and they are integrated in a programming environment too.

Visualization system studied are two. *GHood* (Reinke, 2001), which shows graphically 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 changed. *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 Particular Aspects of Functional Programming to Visualize

4.1 (sub)expression Evaluation, (sub)results Obtainned and Redex

A functional expression has a tree structure, so internally all systems work with expressions as if they were trees (or directed graphs in lazy functional languages). Also, many systems visualize expressions as trees. These are *CIDER*, *KIEL* which allows to interact directly with the tree, *Prospero*, Hint and *TERSE* (giving different representation to constructors, variables and functions).

When an expression is large, its visualization can be confusing, some tools make a compact version of the expression. *Prospero* and *Hint* allow aplying filters (spatial and temporal ones), moreover *Hint* provides a metalanguage to define those filters. *TERSE* transforms subtrees into tree nodes. *WinHIPE* simplifies the visualization of an expression aplying fish-eye views. *ZStep* allows filter expressions by defining conditions. *Evaltrace* compacts trivial evaluations steps, for instance, in the evaluation of `sumlist([])`, first the parameter `[]` is evaluated into itself.

All except *Hood & GHood* and *Evaltrace* highlight the redex. *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.

4.2 Function Calls, Application of Functions to its Parameters and Pattern Matching

The visualization of function calls is carried out by *KAESTLE & FooScape* drawing a static flow diagram where functions are represented as ellipses, and function calls are represented as arcs from the caller to the callee. The user can choose to hide calls from a function, so the diagram is compacted. It allows a dynamic visualization too, by highlighting functions that have not finished its execution. *Evaltrace* is focused on differentiating evaluation and application of functions to their arguments. It 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. *Visual Miranda* shows all the pattern matching process, trying to match the rule and detailing if the match fails or is correct. Finally, *Hood* allows showing function calls a their result if the observation is located in the definition of the function.

4.3 Environment of Variables and Complex Data Structures

The environment of variables (also called contour) and their values are visualized in several ways. All the systems show variable values. *Hood* is a special case, because it shows values in a particular location in the source code, so in the body of a function, a variable could be visualized 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, but if the line is not solid, then the parent will be the previous contour, so the variable is local. *DrScheme* connects variables and their occurences with lines. *Visual Miranda* shows, for each variable its value, before evaluating a (sub)expression.

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

4.4 Subexpression Sharing in Lazy Evaluation Strategy

Lazy systems should visualize subexpressions shared. *CIDER* only reduces shared subexpressions in the same reduction. *Hat* does not visualizes subexpressions shared until they are reduced, then it highlights all occurences of the shared subexpression. *Prospero* and *Hint* only visualize a shared subexpression one time, the rest of occurences in the graph are connected with the first one by arcs.

5 Conclusions

A more general solution is needed. Although there are three systems (*ZStep*, *DrScheme* and *WinHIPE*) covering several features of functional programming, more efforts are needed to reach a general solution. Our approach is based on joining: *debugger solutions*, like algorithmic debugging and error visualization, which help in solving errors; *stepper solutions* which help in understanding execution steps; and *graphical visualization solutions* which use crafts to enhance visual representantions of the features of functional programming paradigm (alternative visual representations of complex structures, animation generation facilities, large expressions compacting, etc.)

Acknowledgements

This work is supported by the Rey Juan Carlos University under contract GCO-2003-11.

References

M. Auguston and J. Reinfelds. A visual Miranda machine. In *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 *WFPL 2001*, pages 365–368. Springer-Verlag, 2001.

H.D. Boecker, G. Fisher, and H. Nieper. The enhancement of understanding through visual representations. In *SIGCHI'86*, 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.

F. Naharro-Berrocal, C. Pareja-Flores, J. Urquiza-Fuentes, J.A. Velázquez-Iturbide, and F. Gortázar-Bellas. Redesigning the animation capabilities of a functional programming environment under an educational framework. In M. Ben-Ari, editor, *PVW 2002*, pages 60–69, HornstrupCentret, Denmark, June 2002. DAIMI PB - 567 / ISSN: 0105-8517.

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

S.P. Foubister and C. Runciman. Techniques for simplifying the visualization of graph reduction. *Functional Programming*, pages 65–77, 1995.

A. Gill. Debugging Haskell by observing intermediate data structures. In *4th Haskell Workshop*, 2000.

M. Hanus and J. Koj. CIDER: An Integrated Development Environment for Curry. In *WFLP 2001*, pages 369–373, Christian-Albrechts-Universitt zu Kiel, Kiel, Germany, 2001.

N. Kawaguchi, T. Sakabe, and Y. Inagaki. TERSE: TErm Rewriting Support Environment. In *ACM SIGPLAN Workshop on Standrad ML and its Applications*, pages 91–100, 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.

B. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *SIGCHI'86*, pages 59–66, Boston, MA, April 1986. ACM Press.

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

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

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 *ACM SIGPLAN Haskell Workshop 2001*, pages 121–149, Firenze, Italy, September 2001.

J. Sparud and C. Runciman. Tracing lazy functionals computations using redex trails. In H. Glasser, P. Hartel, and H. Kuchen, editors, *PLILP 1997*, volume 1292, pages 291–308. Springer. LNCS, September 1997.

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

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