Graphical visualization of the evaluation of functional programs

Ricardo Jiménez-Peris,[1] Cristóbal Pareja-Flores,[2]
Marta Patiño-Martínez,[1] J. Ángel Velázquez-Iturbide[1]

[1] Depto. de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática. Universidad Politécnica de Madrid
Campus de Montegancedo, s/n. 28060-Madrid. SPAIN
{rjimenez, mpatino, avelazquez}@fi.upm.es

[2] Depto. de Informática y Automática,
Escuela Universitaria de Estadística. Universidad Complutense de Madrid
Avda. Puerta de Hierro, s/n. 28040-Madrid. SPAIN
cpareja@eucmax.sim.ucm.es

Abstract

The increasing interest in functional programming for computer science education demands adequate programming environments. Our work is based on an integrated programming environment where the evaluation of functional expressions can be seen as a term rewriting process. Our goal is to facilitate the understanding of this process. We propose an innovative way to display the evaluation of functional expressions that combines text and graphics. Lists and trees constructors are displayed graphically, while the remaining expressions are shown as text. An adequate format for graphics and pretty-printing of text gives a very clear presentation of the evaluation of expressions.

1 Introduction

There is an increasing interest in functional programming for computer science education, mainly for programming education (e.g. see [4]), but also for other subject areas [1]. One of the problems functional programming faces for a wider use in computer science education is the lack of adequate programming environments. They should be state-of-the-art ones, i.e. integrated environments with user-friendly interfaces. In addition, these environments should not mirror directly tools for imperative programming, but should adapt them to the characteristics of functional programming, or even to provide new tools.

We have developed an environment [10] for the functional language Hope$^+$ [5], that fulfils the previous features. In particular, it provides the programmer with a view of the evaluation of functional expressions as a term rewriting process according to an operational semantics for the language. The user can observe intermediate expressions obtained during the evaluation process.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

Integrating Tech. into C.S.E. 6/96 Barcelona, Spain
© 1996 ACM 0-89791-844-4/96/0009...$3.50

Some of the facilities provided to the programmer are: the ability to choose the evaluation strategy (either eager or lazy), and the ability to control the progress of the evaluation of expressions. An expression can be evaluated in three ways: completely, one rewriting step (step breakpoint), or until the application of a particular function (a functional breakpoint).

An important problem is the overwhelming amount of information that can result during the evaluation process. In this paper, we briefly describe these problems and several solutions, but we concentrate on the most innovative technique. It consists of the presentation of expressions yielded during an evaluation, in a mixed format, both textual and graphical, that provides an intuitive feeling of which parts of the expression are functional values and which are not, as well as how the evaluation progresses.

2 Managing intermediate expressions

The control of intermediate expressions exhibits several problems, each deserving a separate solution. We only point out some of these problems and outline the solutions proposed for our environment:

- Too many intermediate expressions.

The programmer must be able to control the evaluation process, so that just some expressions are shown. The environment offers three facilities, as mentioned above, and we foresee the incorporation of others existing in debuggers.

- Presentation of large expressions.

Only those parts relevant for our purposes should be shown. There are some mechanisms for hiding certain subexpressions, e.g. syntax-directed editors [7] use elision '...' to represent hidden parts.

- Discriminating among subexpressions.

This is important for several purposes, such as readability or knowing the next subexpression to be evaluated. Simple solutions are pretty-printing or highlighting subexpressions.

- Presentation of structured values.

It is a particular case of the last problem, since the structure of data involved in the computation is not clear, e.g. it is hard to read an expression denoting a tree. We propose an innovative solution which displays subexpressions in a mixed fomat, graphical and textual. The former is used for showing structured values; the latter is used for the remaining kinds of subexpressions.

3 Visualization of expressions

The representation of structured values by means of expressions is a powerful feature of functional programming, not available in standard imperative languages. However, such expressions are often hard to read: the programmer must 'parse' the expression and build, either mentally or on paper, a picture of the denoted structure. This problem is illustrated by means of an example. Consider a function that mirrors a tree:

```
data tree(alpha)
    == Empty
    ++
     Node(tree(alpha)#alpha#tree(alpha));

dec mirror : tree(alpha) -> tree(alpha);
--- mirror (Empty)
      <= Empty;
--- mirror (Node (t1, x, t2))
      <=      Node(mirror     (t2),     x,
     mirror(t1));
```

An excerpt from the sequence of expressions produced during an evaluation is:

```
    mirror (Node (Node (Node (Empty, 5,
    Empty), 3, Node (Node (Empty, 6,
    Empty), 4, Empty)), 1, Node (Empty, 2,
    Empty)))
→   Node (mirror (Node (Empty, 2, Empty)),
    1, mirror (Node (Node (Empty, 5,
    Empty), 3, Node (Node (Empty, 6,
    Empty), 4, Empty))))
→   ...
→   Node (Node (Empty, 2, Empty), 1, Node
    (Node (Empty, 4, mirror (Node (Empty,
    6, Empty))), 3, mirror (Node (Empty,
    5, Empty))))
→   ...
→   Node (Node (Empty, 2, Empty), 1, Node
    (Node (Empty, 4, Node (Empty, 6,
    Empty)), 3, Node (Empty, 5, Empty)))
```

The previous problem is not exhibited by other classes of expressions, at least so crudely. For instance, function application is better understood by the text denoting it, e.g. `summation (1,10,sqr)` for representing the summation of the square of numbers between 1 and 10, inclusive. Consequently, we have adopted a mixed format, textual and graphical.

The incorporation of this style of presentation in the environment will be available both to display expressions and to input the expressions for evaluation with a graphical editor. We have prototyped this mixed presentation style in the environment, restricting it to lists and trees, with a prefixed format. In our prototype, the previous evaluation is shown as in figure 1. Notice that bounding boxes are displayed to indicate a change from function to values and vice versa.

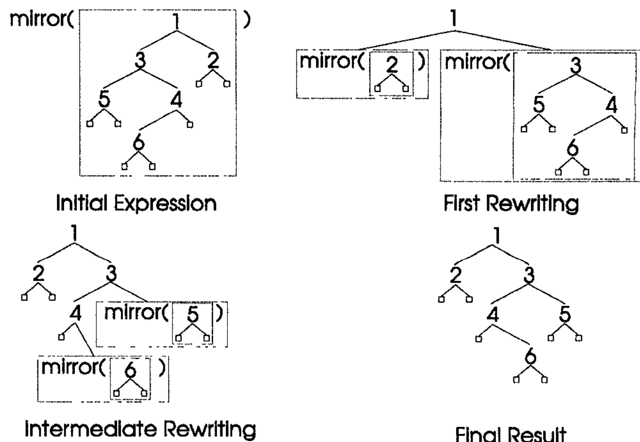

Figure 1

The usual list notation using brackets is also readable (e.g. see the list [1, 2, 3]), but it may also become obscure within large expressions. Consequently, we have also adopted a graphical visualizations of lists. Figure 2 shows four snapshots of the preorder traversal of a tree, storing it in a list.

This kind of presentation has several benefits:

- It associates a substantially different presentation for constructors (graphics) and for other expressions (text). This helps to distinguish intuitively between values and general expressions, mainly between values and functions.

- Box nesting is clearer than text nesting. It helps to identify clearly the syntactical structure of expressions when values and other classes of expressions are mixed.

- The evaluation progress is clearly shown, e.g. the movement of function applications towards the leaves in trees or inward in lists is easily seen, reinforcing the intuition behind algorithms.

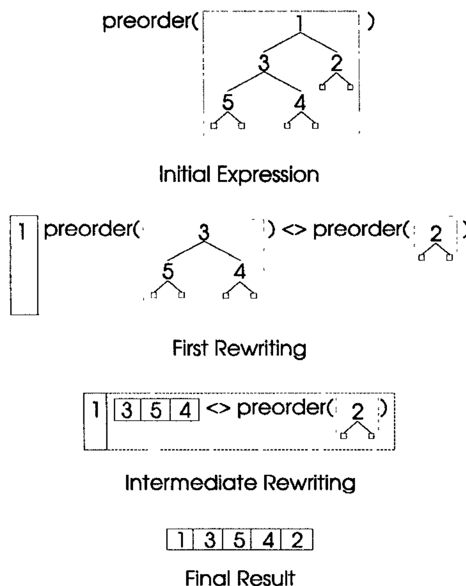

Figure 2

4 Related work

There are a number of systems, similar in several ways, to the one described in the paper. Algorithm animation systems [6, 8] provide powerful facilities to make visualizations but they require a substantial programming effort in contrast to our automatic displays. Other related works are those about debugging functional programs [3], but they do not include graphical presentations. Two exceptions are the works [2] and [9]. The former visualizes lazy functional programs, providing valuable help in tracing the often unpredictable behaviour of lazy evaluation. However, they show a low level representation of expressions, syntactic graphs. This model is useful for professionals, not for students. The second work is based on the visualization of expressions, as is our environment, providing some graphical aids for control, but not for data.

5 Conclusions

We have presented an innovative way to display the evaluation of functional expressions that combines text and graphics. Lists and trees constructors are shown graphically, while the remaining expressions are shown as text. An adequate format for graphics and pretty-printing of text gives a very clear presentation of the evaluation of expressions. We think that the integration of this mixed presentation style with the rest of the partial solutions described in section 2 becomes a powerful tool for understanding the evaluation of expressions based on the operational semantics of the language.

In the future, we plan to extend this mixed presentation style to any data type, built-in or user-defined. We will provide a comprehensive set of graphical formats so that the programmer can choose one and associate it with a data type declaration. Expressions will also be input in the same format with a graphical editor; thus, testing of programs will be easier, keeping a uniform and friendly interface.

References

1 First International Symposium on Functional Programming Languages in *Education (FPLE'95)*. Nijmegen-Plasmolen, The Netherlands, December 1995.

2 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*, Glasgow 1994, Springer-Verlag, 1995, pp. 66-77.

3 Johnson, M. S. (ed.). *ACM SIGSOFT/SIGPLAN Software Engineering Simposium on High-Level Debugging*. Pacific Grove, California, March 1983.

4 *Journal on Functional Programming*, January 1993. Special issue on education.

5 Perry, N. Hope. Technical Report IC/FPR/LANG/2.5.1/7, Dept. of Computing, Imperial College, University of London, October 1989.

6 Price, B., Small, I. and Baecker, R. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(1), September 1993, pp. 211-266.

7 Reps, T. and Teitelbaum, T. Language processing in program editors. *Computer*, 20(11), November 1987, pp. 29-40.

8 Roman, G. C. and Cox, K. A taxonomy of program visualization systems, *Computer*, 26(12), December 1993, pp. 11-24.

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

10 Velázquez-Iturbide, J. A. Improving functional programming environments for education. In M. D. Brouwer-Janse and T. L. Harrington (eds.), *Human-Machine Communication for Education Systems Design*, Springer-Verlag, 1994, pp. 325-332.

Acknowledgements

This work was partially supported by the Spanish agency CICYT, under project TIC95-0967-C02.