# Automatic simplification of the visualization of functional expressions by means of fisheye views

J. Ángel Velázquez Iturbide[1]

**Abstract**

We face the problem of simplifying automatically the visualization of functional expressions. The problem rises when debugging in programming environments based on a rewriting model of expression evaluation, because very large intermediate expressions must be displayed. A simplification technique should filter those parts of an expression which are not interesting for debugging. We propose fisheye views as an adequate technique because they provide a balance between showing global context and local information; in particular, we have designed fisheye views of functional expressions to show the structure of the whole expression and subexpressions around its redex. The amount of information shown can be easily adjusted by the programmer, since it relates directly to the syntactic structure of expressions. The technique is general, since it can be used with any technique for visualizing expressions, from text pretty-printing to graphical representations. We also include in the paper several examples of application, discuss alternatives for several aspects of the technique and introduce a more general definition we are currently studying and experimenting.

*Keywords: functional programming, programming environments, visualization.*

## 1 Introduction

The evaluation of functional programs is based on a model of expression evaluation by rewriting. Consequently, the state of evaluation at any moment is given by the current intermediate expression resulting after previous rewritings. A debugger for functional programming adapted to such an evaluation model must allow the programmer to deal with intermediate expressions by means of adequate tools.

Intermediate expressions obtained during evaluation can be quite large, becoming incomprehensible. Therefore, the debugger should not always show the whole expression, but only relevant parts. Some traditional solutions, such as tracing, are good in the sense of allowing the programmer to focus on several points of interest. However, they are *ad hoc* solutions, not truly integrated with such operational view of evaluation. In addition, they are incomplete and rigid, because they do not usually allow to know other parts of the current expression, apart from the last function application and variable bindings.

We face this problem in a programming environment we have built with an educational purpose [9]. HIPE (*H*ope$^+$ *I*ntegrated *P*rogramming *E*nvironment) is based on the language

[1] Escuela de CC. Experimentales y Tecnología, Universidad Rey Juan Carlos,
Camino de Humanes 63, 28936 Móstoles, Madrid, Spain. E-mail: a.velazquez@escet.urjc.es

Hope$^+$ [6] and offers the programmer a rewriting model of expression evaluation.[2] The programmer can control the reach of evaluation from the current intermediate expression: to rewrite one or several steps, to evaluate completely either the expression or the redex, and to advance to a breakpoint (in HIPE, the application of any user-defined function selected by the programmer). In either case, the debugger shows the resulting expression (either intermediate or final). The programmer can also choose the evaluation strategy, either eager or lazy.

A solution to handle large expressions is browsing: the programmer can navigate through the expression, expanding and contracting different subexpressions according to their perceived degree of interest. Although this solution is simple and flexible, the programmer is burdened with determining what parts of an expression are interesting. Thus, browsing can be used as a complementary facility, but it should not be used as the primary solution to simplify visualization. We require automatic techniques which simplify the visualization of the current expression in the most comfortable and meaningful way to the programmer.

We have studied approaches to this problem in the functional programming community [2] and to the scalability problem in the visualization and the user interface communities (see e.g. [4, 8]). As a conclusion, we consider more adequate for our automation requirement those techniques which simplify the "logical space" of objects to show than those which simplify the "graphical space" of the visualization. That is, we think that the simplification must remove those parts in a functional expression which are not interesting for the debugging task. The technique that best fits our approach is *fisheye view*, a technique to filter information [3] (although it can also be used as a basis for defining graphical presentations [7]).

The name "fisheye view" derives from fisheye or wide angle lenses: these lenses provide a strategy of vision consisting in showing nearby places in great detail while still showing far places in less detail. Furnas [3] gives a rough idea of the technique by referring to the poster known as the "New Yorker's view of the United States", where Manhattan is shown street by street, New Jersey is given in much less detail, and the rest of the states are reduced to a few landmarks (Chicago, California, etc.). The basic idea behind fisheye views is to provide a balance between local detail and global context. Local detail is required for local interactions with the situation, while global context is needed to know where we are and even to know how to interpret local information.

We can highlight two relevant contributions in this paper. First, we study the adaptation and use of fisheye views to visualize functional expressions. An important advantage of fisheye views is that, given their generality, they are independent from any particular visualization. We illustrate this point by showing its use both with naive pretty-printing and with graphical displays. A second contribution is the generalization of the definition of fisheye views. Basically, this generalization comes from the convenience of defining several criteria for nearness within a functional expression, and to be able to define several focus of interest; some relevant cases of these variations are also described. The generalization allows to use different filters in different debugging situations within the same functional programming environment.

The paper is structured as follows. The following section describes fisheye views as defined by the human-computer interaction community. In the third section, we adapt fisheye views to functional expressions, defining all their constituent elements, and apply them as a filter for pretty-printing and for graphical displays. Section 4 generalizes the definition of fisheye views, motivating the utility of alternative definitions with several debugging needs. In

[2]The choice of the language is not important for the purpose of our paper. We still use Hope$^+$ because of its simplicity and adequacy for an educational use, and because its simple implementation allows us to reduce development and experimentation efforts.

the fifth section we explain our experience using fisheye views. Finally, a discussion and our conclusions are given.

## 2 Fisheye views

In his seminal work, Furnas [3] defines a fisheye view of any general structure by means of a function *Degree Of Interest* (DOI) which assigns to every point in the structure a numeric value representing its worthiness to the user. The visualization of a structure is a display formed by points whose degree of interest is greater than a threshold (an arbitrary integer quantity). A simple but successful definition of DOI, that considers both local and global interest, is:

$$DOI(x,f) = API(x) - D(x,f)$$

where DOI measures the degree of interest of a point $x$ of the structure when the focus of attention is point $f$. The functions API and D define the *A Priori Interest* of any point and the *Distance* between two points, respectively.

This definition of DOI can be customized in different ways to any structure. In particular, a "natural" definition $d$ of the distance function D for tree structures is the number of arcs in the path that links two points in the tree. In turn, a "natural" definition of API($x$) states that the closer a node to the root, the more interesting it is. In summary:

$$DOI(x,f) = -[d(x,root) + d(x,f)]$$

For instance, Figure 1 shows a simple ternary tree annotated with its DOI values, given a particular focus $f$.

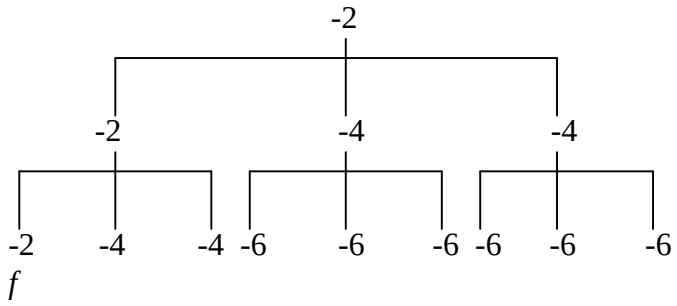


Figure 1. Ternary tree annotated with DOI values for a given focus $f$

The fisheye view of this tree varies with the adopted threshold $t$. Figure 2 shows two fisheye views, for values $t$=-2,-4. Notice $t$=-6 causes the whole tree to be displayed.

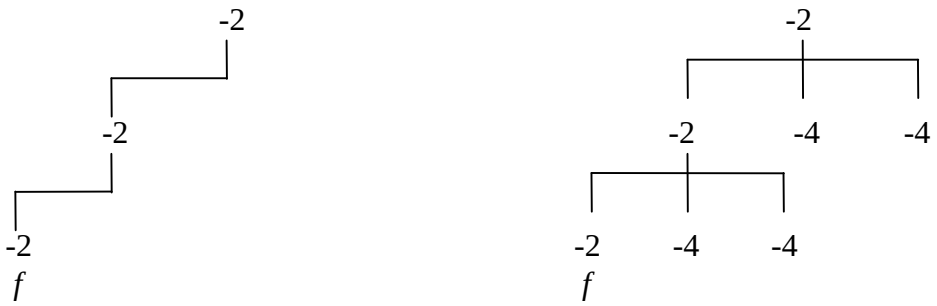


Figure 2. Two fisheye views of the ternary tree in Figure 1

Notice that an annotated tree has a spine of minimal values between its root and the focus. Sets of points with successively larger values are distributed in "concentric circles" around this spine. This property has a number of nice consequences with respect to efficiency [3] and ease of display. For instance, we can ignore any subtree whose root has a numeric value under the threshold.

# 3 Fisheye views of functional expressions

We explain in this section the adaptation of fisheye views to simplify the visualization of functional expressions. In the first subsection, we customize most of its elements. The second subsection defines the most subtle element, the distance function, delaying until the third subsection some additional details. Finally, subsection 3.4 contains two examples.

## 3.1 Definition of fisheye views of functional expressions

The application of fisheye views to visualize functional expressions requires the definition of several elements. Firstly, we must identify the structure on which to apply the fisheye technique. There is a canonical representation for syntactic structures in general, and functional expressions in particular: their abstract syntax tree (AST). It has the additional advantage for our purposes that it is a tree structure. We are not going to define explicitly the AST of Hope$^+$ expressions (see e.g. [1]), but we hope that it is clear from explanations and examples.

We need to specify two more elements in the DOI formulae above: what is the focus? what distance function to apply? The answers to these questions depend on the particular use we plan to make of the fisheye view mechanism. Remind here that we are interested in showing relevant parts of intermediate functional expressions during debugging. We propose the following answers:

- *Focus*. There are several interesting parts of an intermediate expression, for instance the subexpression resulting from the previous rewriting, and the next redex. The latter is in general the most useful choice since, when debugging, the programmer wants to know what will happen next. Notice that a normal form expression does not contain redex, so there are neither focus nor distance to it in this particular case.
- *Distance function*. The most direct option is the syntactic distance in the AST between nodes. However, we have found that comprehension is improved by modifying slightly this distance function, in a way that we explain in the two following subsections. Because of reasons that later will be apparent, we call it the *spatial distance function*.

We still have to define the display of the expression. To keep the discussion simple, we pretty-print naively the expression, with hidden subexpressions identified by elision '...'.

As an example, consider the following conditional expression:

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

that is produced after one step of eager evaluation of `fact 4`, given the usual recursive definition of *fact*. Notice that the redex is `4=0`. The abstract syntax tree of the expression can be seen in Figure 3, where each node is annotated with its API, D and DOI values (in this

order, without their minus sign, and enclosed by brackets). In addition, some nodes contain auxiliary information (e.g. identifiers for functions), and the redex node is underlined. The values of some nodes may result a bit surprising, but they are due to small modifications of the distance function that we explain in the next two subsections.

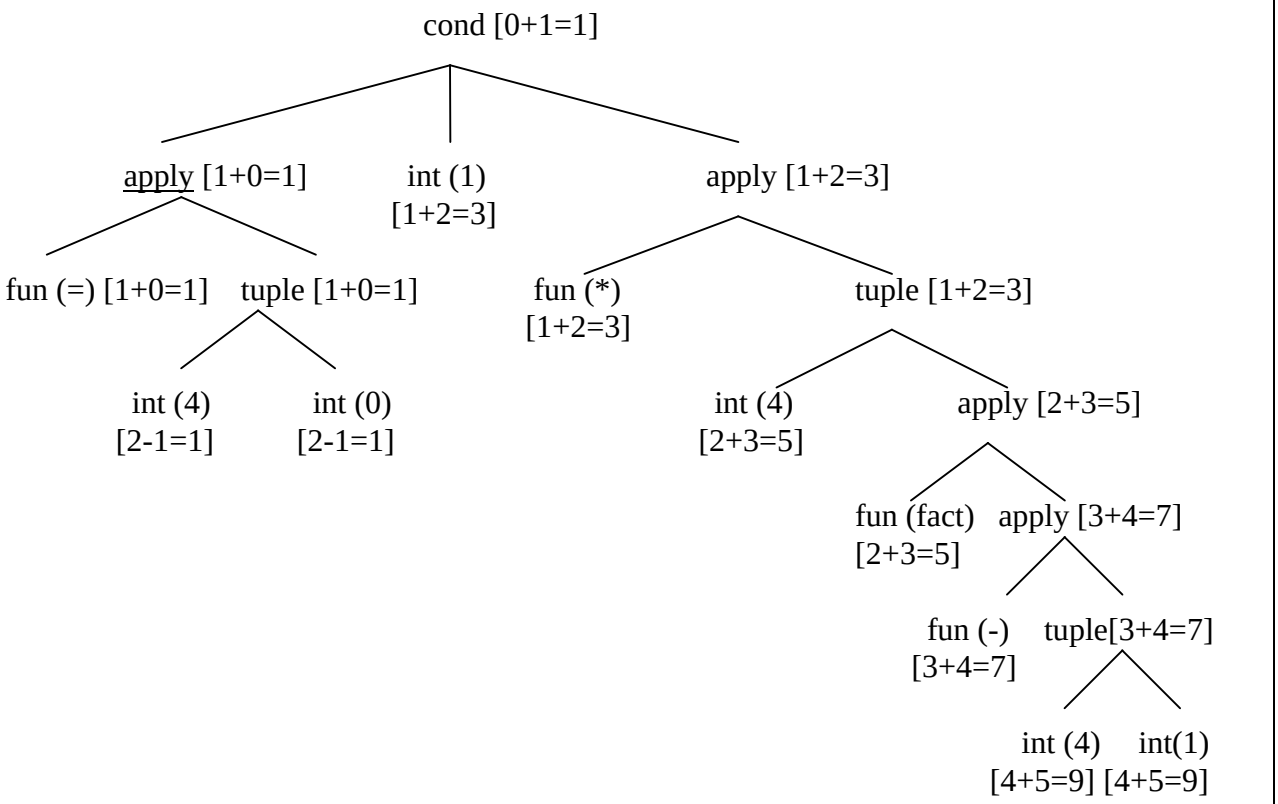


Figure 3. Annotated AST of a functional expression

Given different thresholds *t*, the resulting fisheye views of the expression are:

*t*=-1: `if 4=0 then ... else ...`
*t*=-3: `if 4=0 then 1 else ...*...`
*t*=-5: `if 4=0 then 1 else 4*fact ...`
*t*=-7: `if 4=0 then 1 else 4*fact(...-...)`
*t*=-9: `if 4=0 then 1 else 4*fact(4-1)`

## 3.2 Definition of the spatial distance function

The spatial distance function between two nodes in the AST is a variant of the function that computes the distance between two nodes in an arbitrary tree. We do not give the details here, but it just accumulates the number of arcs in the path that connects both nodes, i.e. the distance value for each arc is always 1. The only relevant variation we introduce on this simple algorithm is that some arcs are not counted, i.e. we consider that the distance of an arc is either 0 or 1. Distance zero is assigned to some arcs to achieve a more meaningful display.

Let us see the distance from each class of node in the AST to its direct descendants. We explain the rationale for such distance by thinking of the display we would like to obtain when each class of node is the frontier among visible and hidden nodes in the fisheye view. Notice that the definition of the distance function is influenced by the expectations about the amount of information to show.

- *Conditional expressions*. We show the expression keywords, but not its subexpressions: `if ... then ... else ...`. Therefore, the three descendants of the conditional node are at distance one.
- *Qualified expressions*. It is analogous, so they are written `let ... == ... in ...`, or `... where ... == ... end`, and the three descendants are at distance one.
- *Tuple*. We want to show that it is a tuple formed by a certain number of components, which are at the same level, for instance, `(...,...)` for pairs. Consequently, all the components of the tuple are at distance one.
- *Lambda expression*. It is analogous to the tuple case: we want to show its keywords and the number of lambda rules. For instance, for a lambda expression containing two lambda rules, it looks `lambda ... | ... end`. Thus, all the lambda rules are at distance one.
- *Application*. It has two descendant expressions, the function or constructor to apply, and the argument. As a representative of the application, we show the function to be applied, but the argument is still kept hidden. For instance, we display `f ...` for any function *f*. Thus, the first descendant of an application node is at distance zero, but the second is at distance one.
- *Lambda rule*. It is similar to the application case: in order to give more information than merely saying it is a lambda rule, we show one level of its pattern, but we do not show the corresponding expression. For instance, we see `nil => ...` for a pattern corresponding to the empty list, and `... :: ... => ...` for the non-empty list.
- *Equivalence*. It is analogous to the two previous cases. We show the variable and the equivalence symbol, but not the corresponding pattern, i.e. `v & ...` for a variable *v*.

## 3.3 Additional features

The spatial distance function defined above provides nice excerpts of functional expressions, but two additional features can improve the treatment of application expressions.

As we noticed in the previous subsection, the definition of the distance function is influenced by the expectations about the amount of information to show. The definition we made of the distance function for applications is incomplete, because it corresponds to the particular case of prefix functions. There are two other important subcases of the application expression with respect to pretty-printing: application of an infix function, and a list or string literal (formed by successive applications of the constructor *cons*, `::` in Hope$^+$). The details of distances associated to arcs are a little cumbersome, but easy to infer, so we only mention here the criteria adopted for showing subexpressions:

- *Infix function*. Remind what happens with a prefix function applied to two arguments. We want to see `f ...` in a first approach, later `f (...,...)`, and so we can go on showing parts of each component of the tuple. However, we expect to display the application of an infix function differently. We first show `... f ...` and then we begin to show its two arguments.

- *List or string*. The list or string notation is much more friendly than its plain syntactic structure. We adjust the distance function so that they are pretty-printed similarly to tuples: `[...,...,...]` or `"..."`.

A final improvement can be made in relation to the presentation of the redex; we motivate it with the case of the redex being an application node. The definition of the spatial distance given above for function application has the nice feature of showing something about an application, namely the function to be applied, but not the argument, for instance `fact ...` or `...-...`. However, this style of presentation is too poor for a redex, especially when it is the application of a primitive or recursive function. In effect, in these cases some knowledge of the redex argument should be given, for instance, `fact 3` or `4-1`. In summary, we want to "widen" the redex node with its direct descendants.

This change of presentation can be achieved by modifying the computation distances. Notice that, when the redex is an application node, we are trying to consider such node and its direct descendants (function and arguments) as a single node in the AST. The distance to the root already makes a difference among these three nodes. So, their union can only be achieved in an uncommon way: we set the distance from the arguments to the application node to be -1. Figure 3 is an example of this case, where the redex contains the equality function.

The redex is also widened for two other cases: the redex node being a conditional or a qualified expression. In this way, we avoid the possibility of simply showing the keywords for the redex (for example, `if ... then ... else ...`), because at least the boolean expression `true` or `false` and something about the two branches is shown.

## 3.4 Some examples

We finish this section by including two examples. First, we show how the evaluation process looks in a simple example, the factorial function. We only unfold completely the first application of *fact*, and later show the intermediate expressions whose redex is a recursive application. For every expression, we set a threshold value that restricts the display to the set of nodes with minimum DOI values, so that only the nodes in the path from the root to the redex are shown. We highlight redexes with bold typing.

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

The second example is used to illustrate the independence of the fisheye view technique with respect to the display format of expressions. We show an example of the simplification of an

expression involving trees, which can be displayed much more user-friendly in a graphical format than in a textual one. We use the same mixed display as in a previous work [5], where lists and trees are represented graphically and the rest of expressions as text.

Consider the following tree:

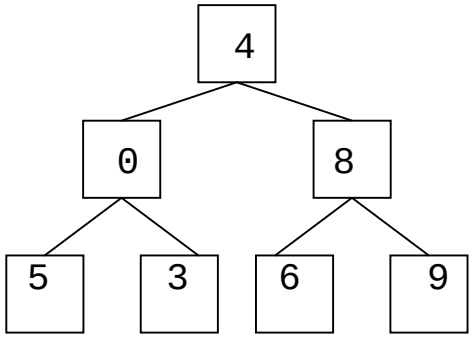


Figure 4. Graphical representatation of a literal tree

which is a graphical representation of the functional literal:

```
Node (Node (Empty,0,Empty),
      4,
      Node (Node (Empty,6,Empty),
            8,
            Node (Empty,9,Empty)));
```

We can apply a function *mirror* to reverse such a tree. The expression obtained after the first rewriting is a tree whose root contains the integer value 4 and its two subtrees have to be reversed yet. The left subtree is the next redex, so its fisheye view contains more details than that of the right subtree. Again, we use elision '...' to refer to hidden subexpressions and we bold the redex. In this case, the threshold is set to show the three sets of minimum DOI values.

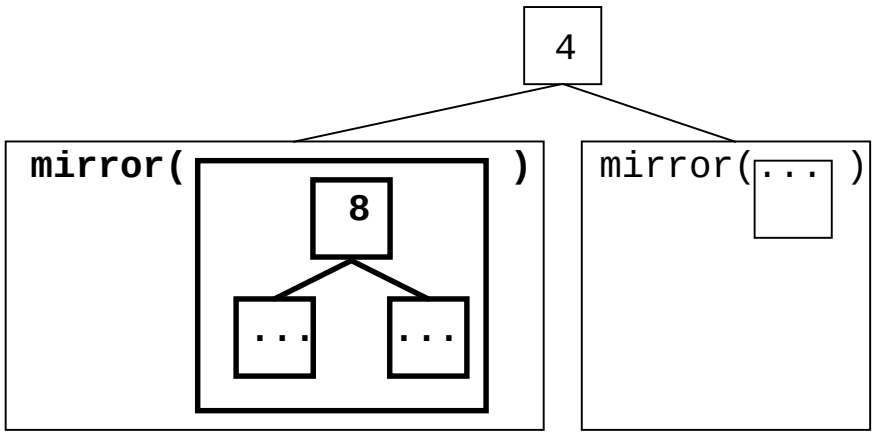


Figure 5. Graphical representation with fisheye views of an expression involving trees

# 4 Alternative fisheye views of functional expressions

The fisheye views of functional expressions given above are useful in general. However, there are many kinds of algorithms and programming errors, so it can be convenient to the programmer to have several strategies for filtering, rather than merely one. This can lead to variants on the previous definition of fisheye views; in the first subsection we describe one such variant. In order to be able to cope with alternative definitions, in the second subsection we generalize further the definition of fisheye views.

## 4.1 Temporal fisheye views

A first alternative on defining fisheye views consists in using a different distance function. Notice that we have emphasized the spatial dimension of functional expressions, considering that expressions which are at the same distance from the redex in the AST are equally important. However, this is not always true while debugging, because expressions do not emerge in isolation, but along a temporal process. Therefore, a different distance function can be defined, based on the "temporal distance" of a subexpression to the redex.

The temporal distance function D(*x*,*f*) is defined as the minimal number of rewriting steps required to evaluate the subexpression whose root is *x* when the current redex is *f*. In practice, this number of steps will commonly be larger, since new expressions will be introduced during the rewriting process.

For instance, let us consider the distance for the conditional expression. The condition is evaluated first, followed by the evaluation of either the subexpression *then* or the subexpression *else*. Therefore, we can say that the temporal distance from a conditional node to its condition is 1 and to the other two subexpressions is 2.

Notice that using this temporal distance function only makes sense when computing the distance to the redex, but not when computing the distance to the expression root. Besides, the temporal distance function can only be used successfully with eager evaluation, where the relative order of evaluation of all the subexpressions is known, but it is far more problematic with lazy evaluation.

## 4.2 General fisheye views

We have studied the possibility of using two different distance functions. We have also mentioned the possibility of not using the redex as the focus, but the subexpression resulting from the previous rewriting. We can also think of other variations in the definition of the DOI function, for example, we could be interested in highlighting all the recursive calls of a certain function by considering all of them as focus. In fact, a programmer in a debugging session would expect to have a versatile tool, where she can use the most adequate fisheye view to any particular problem or algorithm.

This situation leads us to generalize three parts of the definition of the function DOI:

- *Focus*. There can be an arbitrary number. The facilities of the debugger will determine the most convenient focus to use. In particular, we have already identified several useful ones: the next redex, the subexpression resulting from the previous rewriting, and the recursive calls of a given function.
- *Distance functions*. We have seen two useful distance functions: spatial and temporal. Although in principle distances and focus are independent, in practice each distance is most sensible for some focus than for others.

- *Sum of distances*. Remind the DOI function used for trees in previous sections:
  $DOI(x,f) = -[d(x,root) - d(x,f)]$
  where the right hand side comes from the expression $API(x) - D(x,f)$. This formula can be interpreted in a different way: there are two focus, the root of the AST and *f*. The formula can be generalized to an arbitrary set *F* of focus, resulting in:
  $DOI(x,F) = -[\sum D(x,f)]$, for all *f* in *F*
  In the formula, distances to focus are added, but more generally, they can be combined with an arbitrary combination function *Comb* (i.e. the product or minimum functions):
  $DOI(x,F) = -[Comb(D(x,f))]$, for all *f* in *F*

This generalized definition of the DOI function opens new possibilities to design fisheye views adequate to different purposes. Although they require more study, certain facts can already be identified. First, addition has a number of nice properties with respect to efficiency and ease of display, mentioned at the end of section 2. Other functions can be more convenient in some cases, but they have a cost.

A second consequence of our generalization is that the DOI function does not require the AST root to be a focus, but it is left to the choice of the programmer. In this case, the global context of the whole expression is lost, resulting views similar to traces. This contradicts our criterion of reconciling the use of global and local information. However, it also gives more flexibility to the programmer, while debugging in a structured, non *ad hoc* way.

# 5 Experience

We have integrated fisheye views in our programming environment pretty-printer. At the moment, we have implemented fisheye views based on the spatial distance function and for eager evaluation. The current version of the programming environment is similar to Turbo Pascal environment for MS-DOS, being available at the address `ftp://ftp.escet.urjc.es/programacion/thipe`.

Apart from the algorithm to annotate distances in the AST and the adaptation of the pretty-printer, it was necessary to give a simple user interface to the programmer, so that she could understand the basics of the mechanism without unnecessary and annoying details.

In the implemented definition of fisheye views, a simple solution was possible because their annotations exhibit the known property that the nodes with lowest values are along a spine which links the root and the redex. The rest of numeric values are distributed in "concentric circles", each one with a value smaller by 2. Each concentric circle in the AST is formed by the direct descendants of nodes in the contiguous inner circle.

Given this property, a simple interface can be provided to the user. She must only say the number of concentric circles that she wants to see. The lowest value is zero, indicating that only the spine will be shown. The smallest value that produces a display of the whole expression can be arbitrarily small or large, depending on the depth of the AST.

From the implementation point of view, the correspondence between the number of syntactic circles and the threshold to use is the following: the threshold is equal to the value associated to the redex (i.e. the lowest value at the AST) plus twice the number of circles.

Even with this simple interaction, not always the user knows the most adequate number to define the fisheye view. She has the possibility of changing it and redisplaying the expression. Initially, the programming environment sets the highest integer value to the threshold, so that, by default, complete expressions are shown.

# 6 Discussion

There are few experiences similar to ours to simplify *automatically* the visualization of functional expressions; in fact, we only know of that by Foubister and Runciman [2]. They define filters for the visualization of lazy programs written in a subset of Haskell. There are two kinds of filters, one to identify the intermediate expressions to show and another one to define the display of every such expression. A language, including a set of primitives, is provided that allows the programmer to define filters in a flexible way. However, the programmer must work hard to use them: she must learn a new programming language, she is not given hints on what filters are more adequate, and the interaction with the programming environment is complex. Our solution is much simpler from the point of view of the programmer. Therefore, their solution can be more adequate for professional programmers, but it is too complex for occassional programmers or for novices in an educational use.

The current implementation of fisheye views have some visualization and efficiency properties that make them appealing. The technique is a general filtering method, independent from the pretty-printing format: we showed in subsection 3.4 that it can even be used for graphical visualizations. In fact, we are now integrating it into a mixed text-graphics visualizer [5], where list and trees values are displayed graphically and the rest of expressions are displayed textually. It is being included in WinHIPE, a new version for Windows of the environment, available at the address `ftp://ftp.escet.urjc.es/programacion/winhipe`, which is in an advanced stage of construction.

We have noticed that the technique must be complemented with other techniques in order to make its best use. In particular, browsing is a good way to investigate in a particular subexpression without affecting the visualization of others. In addition, the best pretty-printing techniques we have (graphics, coloring, etc.), the best use of fisheye views can be made. Finally, highlighting the redex in any expression is very useful to understand the resulting view.

An interesting problem is its interaction with pretty-printing of infix functions. These functions are a user-friendly facility for the programmer, which make program text more complex to understand, but that, given adequate rules for priority and associativity, result in a more convenient notation. What we had not predicted is the interplay between elision '...' and infix operators: we tend to read elision as another infix operator, so it is not always easy to interpret text with both features. Coloring can be a good aid to eliminate these ambiguities.

A future line of work consists in further studying whether fisheye views indeed provide an improvement of debugging efficiency. We have studied their impact in many programs, including multiple recursive functions, but a more systematic assessment should be addressed. In any case, it is obvious to us that debugging is a complex activity where many facilities and aids must be provided. Any improvement in one facility easies the debugging process, but only by improving all these facilities, debugging can be dramatically eased.

Another interesting issue is the most adequate value of the threshold for different problems. We have simplified the user interaction to set it, and we have thought of some further interaction improvements. We have observed that usually the threshold value depends on the syntactic depth of the data type of parameters in recursive calls: it is quite different its value to fully view a number, a list or a tree. What remains an interesting problem is whether we can also provide automatic aids to set them.

Another future work is related to the linear nature of fisheye views. While debugging, the programmer often gives much more importance to some parts of a program than to others. We have to study whether this nonlinear interest can be simulated using variants of fisheye views

(e.g. multiple focus in recursive calls or different distance functions). The main problem for nonlinear selection is that it does have such good properties for pretty-printing simplicity and efficiency.

# 7 Conclusions

We have applied the fisheye view technique to simplify automatically the visualization of functional expressions in a programming environment. Fisheye views allow to trade-off the presentation of local detail (here, the redex) and global context (the whole expression) in order to provide a meaningful simplified expression. It can easily be adjusted by the programmer, since it relates directly to the syntactic structure of expressions. The technique is general, since it can be used with any technique for visualizing expression, from text pretty-printing to graphical representations. The success of the technique depends on the use of other related techniques, such as browsing or pretty-printing. We have also shown several alternatives and a more general definition we are currently studying and experimenting.

# Acknowledgments

The work here reported was made in July 1997 during a stay at the University of York, invited by the Functional Programming Group directed by Colin Runciman. I want to thank Cristóbal Pareja Flores, Ricardo Jiménez Peris, Marta Patiño Martínez and Colin Runciman for some discussions about the topic, and two anonymous referees for their comments.

# References

[1] A. J. Field and P. E. Harrison, *Functional Programming*, Addison-Wesley, 1988

[2] S. P. Foubister and C. Runciman, "Techniques for simplifying the visualization of graph reduction", *Functional Programming, Glasgow 1994*, Springer-Verlag, 1994, pp. 66-77

[3] G. W. Furnas, "Generalized fisheye views", *ACM SIGCHI'86 Conference on Human Factors in Computing Systems*, pp. 16-23

[4] L. M. Gómez Henríquez, *Sistematización y uso de las técnicas de visualización de programas concurrentes*, PhD Thesis, Facultad de Informática, Universidad Politécnica de Madrid, Spain, May 1995

[5] R. Jiménez Peris, C. Pareja Flores, M. Patiño Martínez and J. Á. Velázquez Iturbide, "Graphical visualization of the evaluation of functional programs", *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (ITiCSE'96)*, ACM Press, pp. 36-38

[6] N. Perry, *Hope*$^+$, Technical Report IC/FPR/LANG/2.5.1/7, Dept. of Computing, Imperial College, University of London, October 1989

[7] M. Sarkar and M. H. Brown, "Graphical fisheye views", *Communications of the ACM*, 37(12):73-84, December 1994

[8] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, 3rd ed., 1998

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