LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1999

A debugging environment for lazy functional languages

GUY LAPALME (*lapalme@iro.umontreal.ca*)

Département d'informatique et de recherche opérationnelle
Université de Montréal
CP 6128, Succ "A"
Montréal Québec Canada
H3C 3J7

MARIO LATENDRESSE (*latendresse@crim.ca*)

Centre de recherche informatique de Montréal
3744 Jean-Brillant
Bureau 500
Montréal Québec Canada
H3T 1P1

Keywords: Debugging, Lazy functional language

Abstract. This paper describes a new approach for debugging lazy functional languages. It rests on the fact that a functional program is the transformation of an expression; one debugs a program by investigating the syntactic form of the expression and by stopping the reduction process at given points. We show what problems are involved and our approach to solving them in a prototype implementation.

1. Introduction

Functional programming languages using lazy evaluation have shown a great potential for all kind of applications. New implementation techniques based on graph reductions of combinators [12] and supercombinators [1] allow an efficient use of the computer for such programs. As they involve some modifications of the underlying graph structure of the programs, classical debugging tools such as traces or stack frames printouts are inadequate. Lazy evaluation delaying the computation of an expression until it is needed, the evaluation sequence is usually quite hard to predict. The current implementations of lazy pure functional languages (Miranda, LML or Haskell) do not provide debugging tool such as traces, breakpoints and stack inspection facilities commonly found in non-lazy functional languages (Lisp, Scheme and ML). This could be attributed to the "laziness" (pun intended) of the implementors but, as we show in this paper, the problem is more deeply rooted in the subtleties of lazy evaluation. A debugging tool should not change the semantics of a program; but in itself, printing

of information might involve evaluating an expression just for the sake of printing it and in doing so, it might change the evaluation order. Another point is the fact that in a pure functional language, no side effect are permitted, but printing such information is such a side-effect. The way around this problem, is to modify the function so that it returns not only the result but also the debugging information. But this in turn changes the type of the function and every call site has to be modified to take that information into account.

The goal of this paper is to make clear the issues involved in presenting a "non-intrusive" debugging environment for a lazy functional language and use it to debug a small program.

Currently, no well established tool can help a programmer debug a program that uses lazy evaluation. Hall[2] and Runciman[10] have given a few ideas but no definitive system is yet accepted. ML[7] designers have even rejected lazy evaluation on the ground that they were unable *"to see the consequences of lazy evaluation for debugging"* [6]

As we are convinced of the advantages of lazy evaluation, we think that it is more appropriate to find an original approach to develop and debug lazy functional programs because, as we show in the next section, debugging methods for eager evaluation are not appropriate in this case. Section 3 presents our approach to debugging and shows its application within a prototype. We then discuss its implementation and we compare it with previous work.

2. Approaches to Debugging

Functional languages (Lisp in particular) usually have well designed break packages that can stop the execution of a program at a point and permit the "inspection" of stack frames to see the current values of the variables; in some cases, it is possible the execute arbitrary commands and functions in the context of the "broken" program in order to find more easily the cause of the erroneous behavior. Traces of functions can be built upon that facility by breaking a function at its start and/or at its end, printing the values of the parameters and/or the result and then continuing with the execution of the program after the function call.

This works well in a language with eager evaluation (all parameters being completely evaluated before a function is called) because the stack frames contain fully evaluated values that can be inspected without changing their values. Unfortunately, this approach cannot be used with a language that implements lazy evaluation unless good care is taken to display partially evaluated expressions; in the Lisp debuggers these kind of values only occur when closures are involved which are then usually show as "black boxes"

such as `#<closure ...>`. This simple minded approach to partially evaluated values should not be used in a language with a lazy evaluation order because such values occur frequently; they are the rule and not the exception. Lazy evaluation also permits the natural definition of infinite or cyclic data structures that have to be shown in meaningful way and in terms of the original program in order to be helpful. In Lisp, these values can only be created using data structure "surgery" techniques like `rplaca/rplacd` which then have to be printed with special care (e.g. setting the `*print-circle*` flag).

Lazy evaluation in a pure functional language is often implemented using graph reduction where the notion of environment and stack frames is not relevant; it should not be imposed only for the purpose of debugging.

For all these reasons in a lazy functional language we cannot rely on the primitives of a language to obtain the trace of the flow of control. This conclusion has also been reached by Peyton-Jones:

> *"It seems that existing debugging techniques (like putting print statements in the suspected functions, or examining dumps) are inappropriate due to the absence of side effects, and the peculiar evaluation order caused by lazy evaluation".[8]*

In a functional language, all the information given by the application of a function is its value. For example, given the following Haskell[3] function definition: [1]

```
fact n | n==0 = 1;
fact n        = n*fact (n-1)
```

we cannot simply insert a primitive to print its argument upon entry to the function because that would create a forbidden side-effect. In a functional language, printing is done when an expression returns a character string to the "top-level"; this means that for a function to show intermediate results, it has to return these intermediary results. This changes the effect of the original function. This is an annoyance for debugging because that function changes but also all other expressions referencing it (they all have to deal with intermediary results). Hall[2] describes a system that implements this transformation automatically.

Another problem is the order of evaluation. In a classical language, control flow is part of the semantics of the language. Each instruction is executed in a strict order that is well known and specified by the programmer. Showing an intermediary result relies upon this fundamental principle

[1] As we were mainly interested in the run-time aspects of the functional language, we restricted ourselves to a small subset of Haskell but that includes all the "interesting" implementation problems. The appendix describes the language we used in our prototype.

of execution flow. Lazy evaluation can delay the evaluation of parts of expressions and so the control flow is very hard to follow. For example, in the expression `f[2+4,3*4]`, we cannot determine if `2+4` is evaluated before `3*4` or even if it is evaluated at all, unless we are aware of the implementation of `f`; a debugging method that would put too much stress upon the evaluation order would be very delicate to use.

The fact that lazy evaluation can propagate partially evaluated expressions all along the execution of a program further complicates the matter. We cannot force the evaluation of an expression only for the sake of seeing its value for debugging because that could change the semantics of the program (the "Heisenbug effect"). So that implies being able to show a partially evaluated expression and not only final values.

A last important fact is related to the graph reduction used for the implementation of lazy evaluation. This means that the internal representation of the graph of the expression is continually modified and in some cases it amounts to modifying the original script of the program. So special care has to be taken when printing values and relating them to the program.

21 Previous works

Hall and O'Donnell[2] describe an implementation independent method for printing intermediary results. For example, `fact` described above would be transformed to give a form analogous to the following one:

```
fact' n debug | n==0 = (1,[n,1]);
fact' n debug        = (result*n,[n]++debug_out++[result*n])
                       where
                       {(result,debug_out)=fact' (n-1) debug};
```

Evaluating `fact' 3 []` gives the following result:

```
(6,[3,2,1,0,1,1,2,6])
```

which is pretty printed to display this information in a more readable manner such as:

```
fact 3
  fact 2
    fact 1
      fact 0
      1
    1
  2
6
```

Hall and O'Donnell show how to automatically transform a functional program in this manner but they have no easy way of stopping the evaluation at one point. They also present another method of debugging but it relies on the ability of calling the evaluator at run time. For reasons given above, we choose not to go into that direction.

Runciman and Toyn[10] use a "snapshot" to show the sequence of transformations that lead to a result and in this respect their work is closer to ours. The evaluation of an expression can be either be stopped by the user or by the system when a run-time error occurs (e.g. a division by 0 or an overflow). Here is what a snapshot would look if a user interrupted the computation of `fact 4`:

```
fact 4 <<< interrupted >>>
fact (n -> 4) -> (n->4)*(fact (n-1 ->3)) ->
    fact (n ->3) ->(n->3)*(fact (n-1 -> 2)) ....
```

This method displays a set of transformations and not only the state of the expression at the moment of the interruption. It is implemented by using special annotated nodes in the program graph. An annotated node is used by the transformations to keep track of the derivations that gave the resulting value. When an interruption occurs, these annotations are printed in a meaningful way.

Snyder[11] describes a method called "lazy debugging" for deferring debugging decisions until run-time in lazy functional programs. The execution model uses the Turner combinators [12]; by keeping appropriate information, it reconstructs a "meaningful source-like representation" of the original program. The approach is interesting and suggests the use of traces and breakpoints similar to the ones we have defined, unfortunately his current system does not yet make a clear relation between the current expression and the original script. This point is of fundamental importance for a debugging tool and is surely much more than a simple user interface design issue as he suggests.

22 Our debugging tool

Before presenting our debugging tool, we give the design criteria that such a tool should possess:

- the programmer should not have to understand the implementation methods of the language. We do not think it is appropriate to show the state of evaluation by presenting the argument stack. It should not be necessary to know the order of evaluation to use the debugging tool.
- the mechanism should be integrated in the functional language. It should not be necessary to know many semantic and syntactic details that are outside the functional language itself.
- the debugging tool must not change the semantics of the language; evaluation must remain fully-lazy.
- the implementation of such a mechanism should not imply a great loss of efficiency and none at all when debugging is not "enabled".

Our debugging tool is based on the simple principle of the equivalence of a symbolic expression. For example, given the following definition:

```
gcd a b | a == b = a;
gcd a b | a > b  = gcd (a-b) b ;
gcd a b | a < b  = gcd a  (b-a)
```

A good way to understand the behavior of that function is to follow the sequence of reductions that occur in the transformation of an expression that uses it. Imagine that we can see the state of the computation at certain point in the reduction of an expression. For example, `gcd 11 5` is transformed in `gcd (11-5) 5`, then in `gcd 6 (6-5)`, `gcd (6-1) 1`, `gcd (5-1) 1` ... until we arrive at `gcd (2-1) 1` that finally gives `1`. These transformations do not change the final value of the expression; at each point the value is the same but expressed in different forms.

This idea of stopping and showing the current expression forms the basis of our debugging tool. We build a mechanism that can stop the evaluation before or after certain parts of the script of the program and that shows the state of the transformations in order to understand what is causing the wrong behavior.

To do this, we introduce two "break operators" [2] to indicate a break point at an expression; `brkBef` indicates a stop before the evaluation of

[2] Haskell syntax does not allow unary operators, so we use function names instead. We call them "break operators" to indicate that they are not real functions because they are treated in a special way by the interpreter. Their functional reading is the same as the identity function.

the expression whereas `brkAft` indicates a stop after evaluation. Defining a unary operator makes it very simple to embed the debugging tool in the language; no special syntax is needed and guarantees that the expression to be traced is always correctly formed. For example, in the expression `3*2 - brkBef(4*7)` execution stops before reducing `(4*7)`.

To obtain a trace, we use `brkAll` which can be seen as the combination of the `brkBef` and `brkAft` operators; it stops before each intermediary evaluation and after evaluation. We also define `brkOff` to disable a break point and `brkOn` to reactivate it. The next section give examples of their use.

3. Our prototype environment

This section presents in more details the use of the debugging operators. Their use is illustrated in the context of our prototype implementation originally developed in Common Lisp on a Xerox 1109 Lisp Machine; it has now been ported in Allegro Common Lisp on a Macintosh.

Figure 1 shows the environment composed of five text editing windows each having a specific role in debugging:

- script window (top left) holding the program; syntax errors are indicated in this window. As we are interested in the run time aspects of the execution, we take for granted that there are no type errors in the program.
- starting expression (top right) that can reference functions defined in the script.
- current expression (middle) whose value is equivalent to the starting expression and represents the state of the transformations done upon it.
- current sub-expression (bottom) is the expression having as root the current reduction point. This part of the expression will be modified in the following transformations.
- final result (middle right) appears in a special window when evaluation of the starting expression is finished.

The last three windows are alternative views of the transformations that occur on the starting expression, but the experience showed that it was useful to separate them.

Break operators can be inserted either in the script or in the starting expression and reductions are performed until a break point is encountered.

Figure 1: The five windows, at a break point, as seen on the Macintosh.

The user then sees the state of the current sub-expression partially evaluated; the corresponding part in the script and the starting expression are emphasized using a bold font in the text editor windows (see figure 1). The expressions are written using as much as possible the Haskell syntax of the original programs taking care of showing shared sub-expressions by introducing `where` clauses (see the current expression window on middle of figure 1).

3.1 Using the break operators

3.1.1 Break before

The operator `brkBef` is used to indicate a break point before reducing a sub-expression. For example, in figure 1, we added a `brkBef` in the `grtr` function to see the expression at that point. We see the different expressions at some point in the reduction process and their relation with the original program. By placing this operator at the start of each right hand side of the equations we can visualize the arguments at each entry of the function. It can even be put within the guards to obtain a trace at the time of their evaluation.

3.1.2 Break after

The operator `brkAft` shows the value of sub-expression once it has been reduced. Why do we need such an operator? If we only kept the break before, then we would need to know the order of evaluation to see the result of a sub-expression. For example, in `2 - g 5 - f 6` if we want to know the value of `g 5`, we could put a break point before `f 6` but that would imply knowing the evaluation order. By breaking after the evaluation with `2 - brkAft(g 5) - f 6`, we can be certain to have a break when the final result of `g 5` is computed without being aware of this order.

In the case of a result that is not a list or a tuple, the break occurs when the expression does not contain any application (i.e. it is in weak head normal form). In the case of a list or tuple, we chose to be more "selective" because for example in `brkAft[2-3,4*5]` the stop would occur immediately after recognizing that it is a list so the value would be equivalent to the starting one. So we stop only when the whole list has been produced; not all elements have been necessarily evaluated but the whole list has scanned until the end. There are cases shown in [4] where this choice is debatable but on the whole it usually gives the intended effect. Of course, no break point is ever met in the case of infinite structures.

3.1.3 Trace Operator

To continuously follow the transformations of an expression, we introduce `brkAll`, a debugging operator that stops before and after each intermediary

evaluation of an expression. It can be seen as a combination of `brkBef` and `brkAft` but in this case, we do not stop on the reduction of a constant.

3.1.4 Disabling Break Points

The preceding operators (especially the trace one) can generate quite a lot of output and so it is important to have a way for dynamically disabling the static break points that we put in the script. We define `brkOff` for this. Suppose that we want to trace a function `h` but not the computation of its argument given by `g 23`; we simply write `brkAll(h (brkOff (g 23)))`. In this expression, `brkOff` disables all break points within `g` but not the ones in `h`. There are three reasons for having such a disabling mechanism:

- functions implemented by another programmer or predefined in the environment should appear as indivisible entities and not be traced.
- a programmer might want to disable break points in certain parts of scripts to be able to concentrate better on the "suspect" parts. There are even cases where it is impossible to syntactically remove break points to obtain the same effects as the dynamic operator `brkOff`.
- we might want to remove a break point incurred by recursion from another level.

We have also defined the `brkOn` operator to dynamically reactivate break points in the context of `brkOff`.

3.1.5 Methodology of use

By combining these simple operators and by inserting in the appropriate places in the script we can obtain almost any information that would be interesting in a program. Table 1 gives a summary of the use of the operators given the place where the break operator is inserted . Latendresse[4] gives more examples of the use of these operators for debugging a functional program. Of course, this environment showing the state of the transformations on the current expression is also available when a "real" error occurs (e.g. `head` of an empty list) and that in itself is a big improvement on the current practice of only printing a cryptic error message.

3.2 Permanence of break points and CAF

The break operators can be used either in the script or in the expression to be evaluated but there is a fundamental difference between these uses because, in the script, the operator has a certain permanence and can be reused many times. A `brkBef` in the body of a function is used at each call of the function; of course, this is a consequence of the instantiation of the body of the function where it appears as any other operator. But there

	`brkBef`	`brkAft`	`brkAll`
f $\mid g_1 = rhs_1$ $\ldots$ $\mid$ **brk** $g_i = rhs_i$ $\ldots$ $\mid g_n = rhs_n$	stopping before the ith equation and showing the values of the arguments of f	stopping before the call to the ith equation	trace of the guard of the ith equation
f $\mid g_1 = rhs_1$ $\ldots$ $\mid g_i =$ **brk** rhs_i $\ldots$ $\mid g_n = rhs_n$	stopping before the ith equation and showing the values of the arguments in the right hand side	stopping before returning from the call to the ith equation	trace of the ith equation
f $\mid$ **brk** $g_1 = rhs_1$ $\ldots$ $\mid$ **brk** $g_i = rhs_i$ $\ldots$ $\mid$ **brk** $g_n = rhs_n$	stopping before the call to the function and showing the arguments of f	indicates the chosen right hand side without showing the values of the arguments	trace of the guards of the function
f $\mid g_1 =$ **brk** rhs_1 $\ldots$ $\mid g_i =$ **brk** rhs_i $\ldots$ $\mid g_n =$ **brk** rhs_n	stopping before calling the function and showing the values of the arguments	stopping before returning from the call	trace of the function

Table 1: Effects of the placement of break operators denoted here by **brk**.

are some functions that are not instantiated when they are used and so a break operator within such a function disappears. A simple example is

```
f = brkBef(1:f)
```

that seems to imply that a break will occur for each new element in the `f` list. But the definition of `f` is modified when evaluated to give an infinite list of `1`s and making `brkBef(1:f)` disappear. The corresponding supercombinator (see next section) for `f` is removed after the creation of the first two `1`s of the list. So the `brkBef` disappears and the use of `f` does not incur further break point. This behavior will occur for any CAF (Constant Applicative Form)[9, p. 224]. So the permanence of break operators are not guaranteed even in the script and this is consistent with the concept of lazy evaluation although it can be a source of confusion to the unwary. This disappearance is less obvious for supercombinators which were developed in order to find the maximum number of CAF. For example,

```
f x = x + (2 + brkBef 4)
```

is translated into two supercombinators

```
$f x = x + $cl
$cl = 2 + brkBef 4
```

now the evaluation of `f 1 - f 2` only causes one break point because `$cl` is only reduced once as necessary to keep full laziness.

This problem is a very difficult issue that is at the root of full laziness. Should we choose not to generate a CAF that includes a break operator, we would change the behavior of the program. We decided to keep the original semantics so that a part of a script that is evaluated only once is traced only once. Another approach would have been to keep the break operators permanent while keeping the full laziness by modifying the implementation of the reduction. Latendresse[4] discusses the consequences of that choice that was considered but not implemented in our prototype.

4. Implementation of the Break Operators

4.1 The supercombinator compiler

The supercombinator compiler transforms functions and expressions in such a way that fully lazy evaluation becomes easily feasible with a simple interpretation of the resulting code. The actual Lisp function that interprets the supercombinators is only 3 pages long, including instantiation of supercombinators and execution of the list and arithmetic operators. It is implemented using the well known techniques described in [9].

Our compiler creates a distinction between supercombinators that have the same names as user defined functions and intermediary supercombinators generated for the sake of full laziness. This distinction permits us to display user defined functions names and avoid the display of intermediary supercombinators.

To be able to display comprehensible expressions at break points, it is necessary for the compiler to keep the names of global and local functions and the parameters for local functions. This slight overhead provides enough information for good display of the current expression at break points (see figure 1).

42 Reduction in the case of break operators

As said before, we use operators to introduce break points in a script. These operators are similar to other functions found in the language, they are therefore treated in a similar fashion by the compiler and the interpreter. The full effects of the break operators are treated by the interpreter. For the compiler, a small difference is made between an ordinary function and a break point operator.

We now give a brief discussion of the interpretation of break operators.

In the case of the trace operator `brkAll` it is necessary to include a new parameter to the interpreter that controls the display of traces. The other two operators `brkBef` and `brkAft` do not need to be dealt specially by the interpreter. We added a parameter to the interpreter to implement the operators `brkOff` and `brkOn`. This parameter indicates if break points are active or not.

The operator `brkBef` is easily implemented as it requires a simple stop of the interpreter. The operator `brkAll` recursively calls the interpreter with a demand for a trace. The trace is interpreted as a break point before each evaluation of operators and calls to functions. Note that we keep track of which supercombinators represent functions.

The operator `brkAft` deals with compound objects in a special way. To perform its task correctly, it reconstructs a node passing along itself on the other part of the compound object. It is only when it cannot do this operation that a break point is effectively done. In this way, a break point really occurs when all elements of the list have been seen. This works properly as the ith element cannot be accessed without scanning the first $i-1$ elements.

The compiler avoids generating supercombinators in the case of break point operators applied to a constant applicative form (CAF). A supercombinator is generated for the CAF but the break operator is not included in this CAF. Simply put, the presence of a break operator does not impel the creation of a supercombinator to share its result. In this way the operators

remain in the compiled script and their permanence is considered a benefit for debugging. This is in contrast with section 3.2 that discussed the case of a CAF that surrounds a break point.

43 Algorithm to display the current expression

As can be seen from figure 1, the environment displays the current expression at break points. The algorithm to accomplish this is very similar to the interpreter itself!

The main difference is that no operator is ever interpreted and some supercombinators are not instantiated. The occurrence of a supercombinator is treated in three different ways depending on its type. If it is directly related to a global function, the name of that function and the arguments along the spine are displayed.[3] If it is directly related to a local function, a unique name is displayed[4] with the arguments along the spine. In this case, an instantiation is performed on a copy of the supercombinator of the local function to recreate a local definition to display in the `where` clause; it is necessary to include identifiers for the unbound parameters. Finally, if it is an intermediary supercombinator, a full instantiation on a copy of the body is performed as all necessary arguments are present along the spine. Further local definitions occur in the "where clause" to represent sharing of sub-expressions. These were detected as part of the algorithm to avoid cycles.

44 Implementation of the Prototype

The prototype to test our ideas on debugging operators was originally developed on a Xerox 1109 Lisp machine. It requires some 2000 lines of Common Lisp in about 75 functions. We have now ported this implementation on the Macintosh using Macintosh Allegro Common Lisp.

This environment has been used for teaching and for debugging small programs. It has been very useful in these cases but it could not be used for debugging "big" programs mainly because we only deal with a small subset of a "real" language and we have not implemented type checking. But this prototype has established a methodology and identified the important points that should be adressed in an "industrial" lazy functional language environment.

When a break point is encountered, some parts of the script is underlined and/or made bold. This operation is implemented using the resident text editor Lisp functions that perform these operations. The information

[3] The number of arguments is not always the same due to currying. Therefore, the number of parameters of that function should not be used for such a display.

[4] We use the name of the local function suffixed by a unique integer for the current display.

needed to operate on the text is kept in the nodes of the graph. For example, in the expression `20 * brkBef(3 + 4)` the node that describes the constant `20` keeps track of the window and the positions in which it appears; this is also true for `3` and `4`, and the applications of `*` and `+`. This information is gathered by the compiler throughout the script.

Although the graph is continually changed, the positions in the original text of the script or the starting expression of the evaluation is passed along when an instantiation occurs, but it is destroyed when an operator is applied.

5. Further work

Our prototype only deals with a subset of Haskell but it has shown the principles of the break operators. The first extension would be to deal with all the language constructs; we are convinced that the basic approach would be the same but perhaps some adjustments would be needed for the list comprehensions and the different constructors.

The current implementation shows the expressions in the Haskell syntax but it can often display very large structures and it would be important to find a way to shorten this information. A simple way would be to define the equivalent of `*print-level*`, `*print-length*` found in Common Lisp to control until what level nested expressions are printed and how many elements to be printed at each level.

Following Lieberman[5], it would be interesting and possible to go back in the history of the reduction after a break point. It would seem that this would be very expensive in time and space, but as functional languages are side-effect free, this overhead could be manageable.

6. Conclusion

We have presented in this paper the use of break operators for debugging functional languages; they are simple to use and they integrate well syntactically and semantically in the underlying language. This mechanism keeps the properties of the lazy evaluation and can be used without any knowledge of the implementation or the evaluation order. At a break point, the current expression is shown in terms of the syntax of the original program and relations are made between this expression and the program script. We have also introduced a dynamic disabling mechanism to reduce the number of steps to be seen before arriving at the error. These principles and ideas, implemented within our prototype, have been shown to be quite useful and efficient and do not require great modifications to an implementation based on supercombinators.

References

1. Augustsson, L. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1984) 218–227.

2. Hall, Cordelia V. and O'Donnell, John T. Debugging in applicative languages. *Journal of Lisp and Symbolic Computation*, 1, 1 (1988).

3. Hudak, P. and Wadler (editors), P. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1).* Technical Report, Yale University, Department of Computer Science (August 1991).

4. Latendresse, M. *Un environnement de mise au point de programmes écrits dans un langage fonctionnel à évaluation paresseuse.* Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal (1990).

5. Lieberman, Henry. Steps towards better debugging tools for Lisp. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1984) 247–255.

6. Milner, R. How ML evolved. *Polymorphism*, 1, 1 (1983) 1–6.

7. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press (1990).

8. Peyton-Jones, S. L. *Directions in Functional Programming Research.* Technical Report INDRA note 1575, Department of Computer Science, University College, London (1985).

9. Peyton-Jones, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall (1987).

10. Runciman, C. and Toyn, I. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4 (1986) 339–363.

11. Snyder, Robin M. Lazy debugging of lazy functional language. *New Generation Computing*, 8, 139-161 (1990).

12. Turner, D. A. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9 (1979) 31–49.

7. Appendix: Syntax of our language

For our prototype, we chose a small subset of Haskell but that is sufficient to include all implementation problems dealing with the run-time aspects of a lazy functional language.

The program is checked for syntax errors but we take for granted there are no type errors. In fact, we do not even accept type definitions. Of course, a full implementation would have to deal with this, but as it is done before run-time, no modification would be needed for our proposition. For type checking, the break operators are simply polymorphic functions of the same type as the identity function.

A function is written as recursive equations separated by `;`. For simplicity, our lexical analyzer does not deal with the layout rules. For example:

```
gcd a b | a == b = a;
gcd a b | a > b  = gcd (a-b) b ;
gcd a b | a < b  = gcd a  (b-a)
```

In each equation, the left hand side gives the name of the function `gcd`, the formal parameters `a` and `b` and the guard after '`|`' giving the condition to use this right hand side of the equation. No pattern matching is allowed for the parameters. Expressions are arithmetic, boolean or characters constructed with the usual operators. List are built using either the cons (`:`) or brackets (`[1,2,3]` is equivalent to `1:(2:(3:[]))`). Arithmetic sequences like `[1..5]` can be used, sequences can also be infinite like `[1..]`. `head` and `tail` are functions giving respectively the first element of a list or the list after the first element is removed. Tuples are created by putting their elements in parenthesis. Functions can be currified and local definitions can be introduced by a `where` clause.

List or array comprehensions and sections were not introduced in the language. No modules nor input/output request are permitted and we do not provide any standard library of functions.