13th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP04), Aachen, Germany — June 1, 2004.
Technical Report, Department of Computer Science of RWTH Aachen, AIB-2004-05, June 2004, pp. 4-18.

TeaBag: A Functional Logic Language Debugger*

Sergio Antoy Stephen Johnson

Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,stephenj}@cs.pdx.edu

Abstract. We describe a debugger for functional logic computations. The debugger is an accessory of a virtual machine currently under development. A distinctive feature of this machine is its operational completeness of computations, which places novel demands on a debugger. We give an overview of the debugger's features, in particular the handling of non-determinism, the ability to control non-deterministic steps, to remove context information, to toggle eager evaluation, and to set breakpoints on both functions and terms. We briefly describe the debugger's architecture and its interaction with the associated virtual machine. Finally, we describe a short debugging session of a defective program to show in action debugger features and window screenshots.

1 Introduction

Functional logic programming joins in a single programming paradigm characterizing features of functional and logic programming. There are a number of languages with this aim, e.g., Curry [23], Escher [26], Le Fun [2], Life [1], Mercury [36], NUE-Prolog [29], Oz [35] and Toy [27], to name a few. These languages support user-defined functions and the subsequent evaluation of expressions involving these functions. Debugging functional computations of this kind is a non-trivial, but well-studied problem [5,14,15,18,31,32,37,38,40,42]. These languages also support the use of logic variables. Debugging programs with the combination of user-defined functions and logic variables is much more challenging for reasons that will be discussed shortly.

Indeed, programming with the combination of user-defined functions and logic variables is the subject of active research even for its most fundamental aspects, e.g., the formulation of both adequate semantics and efficient implementations. A significant problem of combining functions and logic variables is what to do when the execution of a program leads to the evaluation of a functional expression containing uninstantiated logic variables. This problem is solved by either residuation or narrowing [19]. *Residuation* delays the evaluation by transferring control to some other portion of the program in hopes that the variables will be instantiated by a predicate so that the evaluation of the functional expression can continue. *Narrowing*, instead, guesses instantiations of variables which allow the evaluation to continue. Thus, the result of evaluating by narrowing an expression produces both the value of the expression, generalizing a functional computation, and a substitution of some variables of the expression, generalizing a logic computation. The details of this computation are quite technical and outside the scope of this discussion. Examples will be provided in the next section.

Narrowing introduces non-determinism in the sense that distinct instantiations of a variable in an expression may be equally plausible and different instantiations may lead to different values. This suggests to allow functions (including constants seen as functions of zero arguments) that for the same arguments return different results. Obviously, these "things" are not functions in the mathematical sense, but they are defined and used as ordinary functions in a program. The semantics of a functional logic program is often formulated by seeing the program as a first-order rewrite system. Higher-order and partially applied functions are eliminated by a transformation referred to

* This research has been supported in part by the NSF grants CCR-0110496 and CCR-0218224.

as *firstification* [8,41]. Then, the execution of a program consists in the evaluation by narrowing of an expression using the rules of the rewrite system.

The characteristics discussed above pretty much shape a debugger for a functional logic language. The debugger has the features generally found in tracing debuggers for functional languages. It shows evaluation steps as reductions. In the case of a functional logic language, rather than the β-reductions of the λ calculus, these reductions are narrowing steps of a rewrite system—for first-order, variable-free expressions the difference between the two is small. In addition, the debugger must handle logic variables and non-determinism. How this is done depends on what a functional logic language provides. Our work is centered on Curry [23] and on an implementation of Curry, referred to as the *FLVM*, currently under development [7]. Curry offers both narrowing and residuation and the *FLVM* offers a complete implementation of non-determinism. How these characteristics affect a debugger will be discussed at length in the following sections.

Section 2 contains background information on Curry and the *FLVM*. Section 3 presents an overview of the significant features of our debugger. Section 4 sketches the architecture of the debugger and how it interacts with the *FLVM*. Section 5 shows an example of a debugging session. Section 6 discusses related work. Section 7 offers our conclusion.

2 Background

Curry provides built-in types, such as numbers and characters; user-defined algebraic data types; functions, including higher-order and non-deterministic ones, defined by pattern matching; lazy evaluation; logic variables; and built-in search. The syntax is Haskell-like. An example of a complete program follows. The numbers to the left are not part of the program. They are used for reference purposes only.

```
 1  data Color = red | white | blue

 2  mono _ = []
 3  mono c = c : mono c

 4  solve flag | flag =:= x ++ white:y ++ red:z
 5             = solve (x ++ red:y ++ white:z)
 6             where x,y,z free
 7  solve flag | flag =:= x ++ blue:y ++ red:z
 8             = solve (x ++ red:y ++ blue:z)
 9             where x,y,z free
10  solve flag | flag =:= x ++ blue:y ++ white:z
11             = solve (x ++ white:y ++ blue:z)
12             where x,y,z free
13  solve flag | flag =:= mono red ++ mono white ++ mono blue
14             = flag
```

Line 1 defines the type `Color` whose instances are three constants. Lines 2 and 3 define a non-deterministic function, `mono`, that takes an argument (of type `Color`) and returns a list whose elements are all equal to the argument. The textual order of the rules is irrelevant. Lists of any length can be returned. The remaining lines define a function, `solve`, that "solves" the *Dutch National Flag* problem in the spirit of [16], i.e., by swapping pebbles out of place.

The function `solve` is defined by conditional rules of the form:

$$f\ t_1 \ldots t_n \ \texttt{|}\ c\ \texttt{=}\ e \quad \texttt{where}\ vs\ \texttt{free}$$

The conditions are *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 evaluate to unifiable data terms. Free variables, introduced by the `where` clause, in a condition may be instantiated by narrowing steps, if this is useful to satisfy the condition.

In contrast to other declarative programming languages, e.g., Haskell, where the first matching rule is applied, in Curry all matching (to be more precise, unifiable) rules are non-deterministically applied to support complete computations. This enables the definition of non-deterministic functions, such as `mono` and `solve`, which may have more than one result on a given input. As an example of solving constraints, consider the evaluation of the following expression:

```
solve [white,red,blue,white]
```

Both the first and third rule of `solve` can be fired, because the conditions of these two rules are satisfiable. For example, the first condition holds if `x` $=$ `[]`, `y` $=$ `[]` and `z` $=$ `[blue,white]`. These instantiations of `x`, `y` and `z` are computed by the evaluation of the constraint. With these instantiations, the expression is rewritten to `solve [red,white,blue,white]`.

A crucial design decision of the implementation of the language is how to handle the fact that *two or more* rules are applicable to the same expression. One common strategy is to select one rule and delay the application of the others until the selected rule yields either a result or a failure. This is a simple strategy adopted, e.g., by some implementations of Curry [20,28] and other functional logic languages [27]. But it is unsatisfactory because if the application of the selected rule leads to a non-terminating computation no other rule that could yield a result is ever applied.

Another strategy is to fork the computation for every applicable rule and to execute fairly and independently all the results. This is the strategy adopted by the *FLVM*. This design decision is more satisfactory because it ensures the operational completeness of the language implementation. However, it also introduces novel problems for a debugger, since the trace of a computation is no longer the traditional linear sequence of steps, but it has a tree-like structure. A distinctive feature of our debugger is the handling of this structure.

3 Features Overview

In this section we present some characterizing features of our debugger, called TeaBag (The Errors And Bugs Are Gone!). These features are realized by several interactions with a computation. A synopsis of these features follow: *computation structure* is a window that visualizes the non-deterministic steps of a computation; *choice control* is an option for the early elimination of undesired non-deterministic steps; *context hiding* is an option for displaying only a subterm of the term being evaluated and/or only steps that affect this subterm; *eager evaluation* is an option to eagerly evaluate and/or replace a subterm of a term; *runtime debugging* is a debugging mode that supports non-terminating computations and runtime selection of non-deterministic steps; *breakpoints* is an option to set breakpoints not only on functions, but also on terms; *highlighting* is the use of colors and other visual clues to ease understanding.

There are several classes of debugging tools for declarative languages. This subject will be further discussed in Section 6. To understand some of the following features, we recall the difference between a tracer and a runtime debugger. These terms are not formally defined and our descriptions are only a subjective point of view to aid comprehension. A tracer executes a computation

and when the computation terminates it displays some representation of the computation, e.g., the computation steps. A runtime debugger executes a computation and if some events occur it displays information about these events. The events generally include the termination of the computation, runtime errors, and the invocation of certain functions selected by the user. A runtime debugger can provide useful information about non-terminating and/or failing computations. It can also be helpful when debugging code that interacts with the outside world, e.g., the program directly paints to the screen or uses a socket. However, a runtime debugger generally provides less detailed information about ordinary computations.

3.1 Computation Structure

A computation is the set of the narrowing or rewriting steps performed on the term being evaluated. Computations in deterministic languages are a linear sequences of steps. In a non-deterministic language a computation is a tree sometimes called the narrowing space. The narrowing strategy executed by the *FLVM* is essentially an implementation of the *inductively sequential narrowing strategy* [6] with some adjustments to support residuation. In this framework, narrexes and possibly redexes can have more than one replacement. When this happens, a trace forks into several paths—one for each replacement. In other words, a computation has the structure of a tree and a trace is a path in this tree.

TeaBag provides a view of the tree structure of a computation. This view does not show all the rewriting and narrowing steps of the computation. It just shows a tree in which a branch represents a non-deterministic step and a leaf represents the endpoint of a trace, which is a term totally or partially evaluated. In this view, a sequence of deterministic steps is shown simply as an arc from a parent to a child in the tree. This view highlights the current path through the tree that the user is looking at in the trace browser. The trace browser, discussed in section 3.2, shows all the rewriting and narrowing steps of a trace.

Having a computation structure is very important to understand how a result is obtained. Without the computation structure it is difficult to know where a rewriting or narrowing step fits into the overall computation. The computation structure lets the user know which non-deterministic steps were made to get to any rewriting or narrowing step in a trace. In deterministic computations, where traces are linear, this contextual information can be obtained with just a counter, but this is impossible in non-deterministic computations. An example of the computation structure is in figure 3.

A variable is displayed with both its source code name v and a unique internally generated number n in the format $v|n$. The name aids the user in relating steps of the computation to the source code. Since different variables may have the same name because of either recursion or the scoping rules, the unique number allows the user to distinguish different variables with same name.

3.2 Trace Browser

The trace browser shows the rewriting and narrowing steps for the selected path in the computation structure. There are two ways to view the trace. The first way is as a table of all rewriting and narrowing steps. This view is convenient for getting a “birds eye” view of the trace. However, it is not suitable for examining individual rewriting or narrowing steps of the trace. The second way to view a trace is by examining each rewriting and narrowing step. The user can choose to view one or two terms of the trace at a time. Each term in the trace is obtained from a rewriting or narrowing step on the previous term. The trace browser includes buttons to move to the next, previous, first, and last steps. The user can also select a particular step number to jump to.

The trace browser interacts with the computation structure. The node or edge in the computation structure corresponding to the current step in the trace browser is highlighted. When a non-deterministic step is displayed in the trace browser the user is given the option of which branch to follow. The selected path is highlighted in the computation structure. The user can also select a path in the computation structure and the trace browser will be updated to display that path. An example of the trace browser is in figure 1.

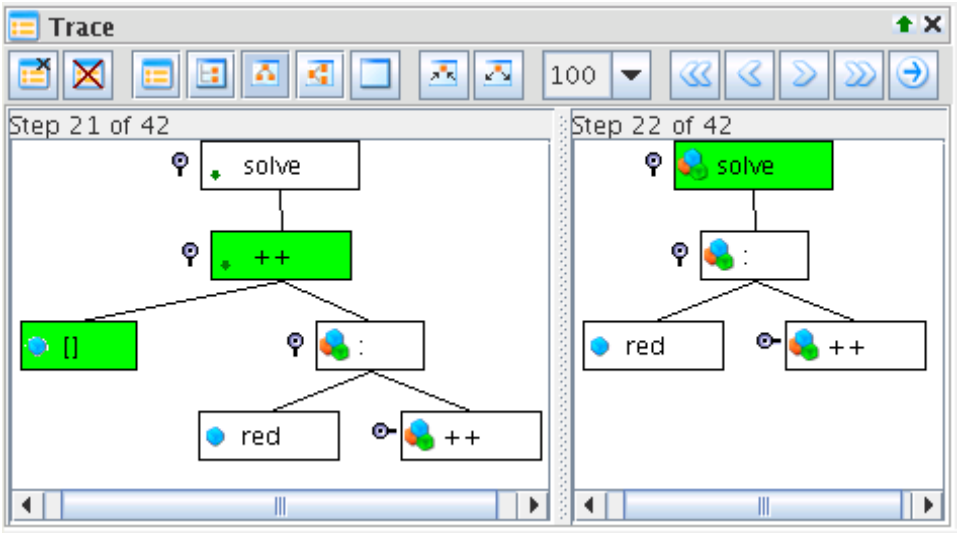

Fig. 1. Example of Trace Browser. The terms are displayed as trees. The user can expand and collapse subtrees. Nodes in the tree are symbols and branches are arguments.

3.3 Choice Control

TeaBag includes computation management to control subcomputations originating from non-deterministic steps made during runtime debugging. When a computation executes a non-deterministic step, the *FLVM* evaluates fairly and independently all the results of this step. This is essential for ensuring the operational completeness of a computation. This view allows the user to kill, pause, and activate the subcomputation of any individual result. Often, the user is interested only in a subset of all the choices of a non-deterministic step. Since there can be an exponential growth of non-deterministic steps, being able to pause and kill subcomputations toward the beginning of a computation can greatly reduce the total number of non-deterministic steps made. This makes it easier for the user to debug computations that would normally produce too many steps to examine.

3.4 Context Hiding

Even for small programs, the sheer volume of data to be displayed and analyzed for an execution may become a serious obstacle to debugging. TeaBag alleviates this problem with two features intended to suppress unwanted information.

Term Size Lazy evaluation has a propensity for creating large terms during a computation. Large terms are not displayed easily and they make it hard to find subterms of interest. Often, the programmer is interested in examining a subterm nested somewhere in a large term. To assist the user

in focusing on this term, TeaBag can be instructed to display a subterm of a term of a computation. The subterm to be displayed is selected by the user. An option, allows the user to select only the redex or narrex of each step, i.e., to eliminate the portion of a term above the subterm replaced by a step. Also, TeaBag will expand terms only as much as is needed to display redex, narrex, and created positions, i.e. to eliminate the portion of the term below the parts of the term that were replaced. The user has the option to further expand the term to see more. Hiding the context of a term makes finding pertinent information easier for the user.

Trace Length Likewise, the number of steps in a trace can be very large. Many times the user will only want to look at a subset of the trace steps, in particular at all the steps that affect a particular subterm of the term being evaluated. For example, this may be convenient for terms rooted by a function thought to be defective. TeaBag lets the user choose which terms to trace. The trace for that term will only have the rewriting and narrowing steps performed on that term or one of its subterms. This feature limits the number of rewriting and narrowing steps that the user needs to look at. This makes it possible for the user to examine traces that would normally be too long to look at.

The programmer may want to look at a portion of a trace after thousands or millions of steps since the beginning of a computation. Displaying or even recording all the steps of a computation can be very time consuming or even infeasible. TeaBag lets the user set breakpoints so that no step is displayed or recorded until the breakpoint. The user can set breakpoints in a program and choose to display only the steps that are executed on a subterm between breakpoints or until the subterm is in normal form.

3.5 Eager Evaluation

In a lazy language, the arguments of a function application $f\ t_1\ \ldots\ t_n$ are evaluated, as the name says, lazily. For example, if t_i is needed to compute the application of f, it will be evaluated to a constructor head normal form. Then steps may be executed on other terms unrelated to f and t_i, but it is possible that either t_i or some other argument of f will need to be further evaluated to compute the application of f. In short, the arguments of f may be computed in stages. This back and forth switching between arguments may occur repeatedly. It may be difficult for the programmer to understand the behavior of f until some of its arguments are sufficiently evaluated, but it is time consuming and distracting to interleave the evaluations of these arguments with the evaluation of other unrelated terms.

On demand eager evaluation is a feature that lets the programmer override the default lazy evaluation of a term. In this context, eager evaluation means evaluation to normal form. Any term displayed in a window can be interactively selected. By default, the result of eagerly evaluating a term is only displayed and does not replace the term. This lets the user see what the term evaluates to without changing the lazy behavior of the program. An option, allows the user to replace a term with its eagerly evaluated result. This is another device to compress the information displayed to the user. Obviously, an attempt to eagerly evaluate a term may result in a non-terminating computation even for a trace that terminates.

3.6 Runtime Debugging

TeaBag is not just merely a tracer. It is also a runtime debugger. TeaBag allows a programmer to see the rewriting and narrowing steps of a computation at runtime. The unique feature of TeaBag's

runtime debugging environment is that it interacts with the tracer. The tracer will reflect the computation(s) the programmer activated during runtime. It will also compress in a single step the optional eager evaluation of a term. Choice Control, the feature described in Section 3.3, is available during runtime debugging and can effect what steps are generated for a trace. The runtime debugging environment is shown in figure 2.

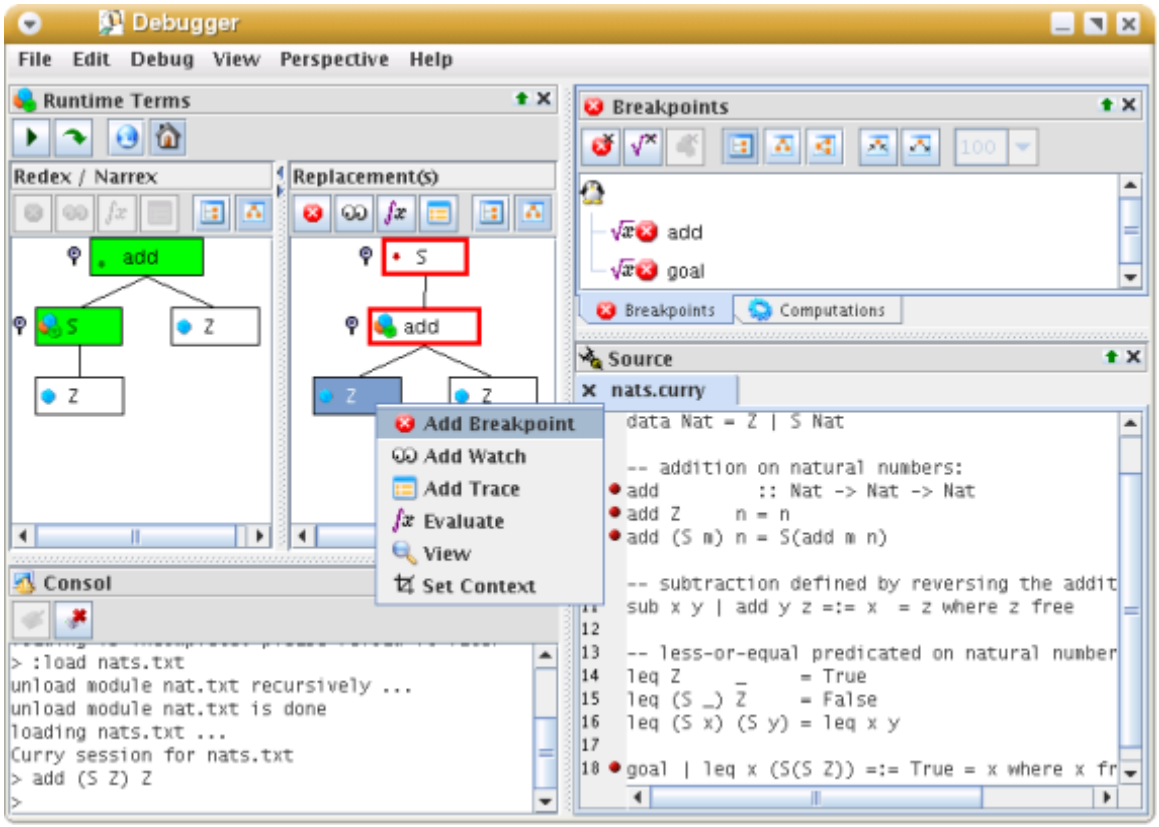

Fig. 2. Runtime Debugging Environment

3.7 Breakpoints

TeaBag lets the programmer set a breakpoint on a function during runtime debugging. Whenever a rewrite rule defining that function is applied, the *FLVM* will be paused and the rule application will be displayed. This is convenient to debug non-terminating computations. TeaBag also lets the programmer attach a breakpoint to an *individual* term during runtime debugging. When that one term is replaced, the *FLVM* will be paused and the step from which the replacement originates will be displayed. This is convenient when the programmer does not understand when or why a term is evaluated. To the best of our knowledge this is the only debugger that allows breakpoints to be set on individual terms.

3.8 Highlighting

Highlighting uses colors, icons, and other visual clues as aids to understand the large amounts of displayed information.

Term Replacement When a rewriting or narrowing step is displayed both the redex pattern and created positions are highlighted. This information is intended to speed up the perception of how a rewriting or narrowing step changes a term. It also enables the viewer to determine which of the possibly many rewrite rules of a function has been applied in the step.

Variable Binding When a variable is bound, both the variable and its instantiation are highlighted. This makes it easier to detect that a step involves a binding and to determine both the variable being bound by the step and the step's substitution.

Source Code When available, the source code executed to perform a step is highlighted. Information about the source code is optionally added by the compiler to the generated bytecode. Providing source code highlighting makes it easy for users to correlate the rewriting or narrowing steps with their code. Being able to relate information displayed by the debugger to source code is considered important [38] since ultimately a bug in a program will be fixed by changing the source code.

Non-Deterministic Choice Selection Source Code Highlighting When the user selects a non-deterministic step to follow in the trace browser, the source code for that selection is highlighted if it is available. For example, in figure 4 the non-deterministic choice selection in the trace browser corresponds to the third rewrite rule for `solve` as highlighted in the source code.

4 Architecture

In this section, we give a few details on the architecture of our debugger. A significant aspect of our architecture is that the debugger interface is entirely separated from the *FLVM*. The debugger is implemented in Java. Java is a friendly, portable language with excellent graphical libraries. The *FLVM* is implemented in Java as well, but this may change in the future. The efficiency of the *FLVM* is obviously a concern and the size of its code is small, thus a conversion to a different language is feasible.

A potential problem of most debuggers is scalability. Generally, one must consider both large programs and programs that execute a large number of steps. In our case, one should also consider programs with a large degree of non-determinism. Often, scalability is in conflict with both providing detailed information and presenting information in a form visually rich, e.g., by means of colors, fonts, and options. We have chosen to provide detailed, visually rich information and have also implemented several features, described in Section 3.4, which should help in debugging realistic programs.

Top Level Architecture The debugger interface is decoupled from the *FLVM* by running both in separate processes and communicating over sockets. This allows the debugger to work with any virtual machine that implements the socket interface. Having the *FLVM* in a separate process makes it easy for the debugger to kill and restart it which is especially useful when the *FLVM* executes a non-terminating computation.

Debug Events The *FLVM* communicates with the debugger by sending it debug events over a socket. The debugger has a thread listening for these events. When data becomes available on this socket the debugger parsers the event and dispatches it to the event thread for processing.

Debug Commands The *FLVM* understands both user commands, such as "`:load`" and "`:quit`" to respectively load a module and terminate an execution, and debugging commands, such as the request to eagerly evaluate a term. User commands are given interactively to the command line interpreter. Debugging commands for the *FLVM* have a textual representation that could be typed in by a user. This allows the *FLVM* to redirect the socket of communication to standard input and run as if there was a user typing in the commands. When a program is executed under the debugger, the debugger acts as a proxy for the *FLVM* for all user input.

Tracing The information for a trace is written to a file during runtime. The trace browser reads data from that file to display the steps in the trace window. Since, the user can choose which terms to trace, not all steps of a computation are recorded to this file. Whenever a term is set to be traced, the *FLVM* creates a chain of responsibility structure [17] among its subterms via a listener. Then, when a subterm is replaced, it propagates this information up the chain of responsibility. Any term along this chain that has a trace set on it fires a trace step event to the debugger. When the debugger gets the trace step event, it writes the event to a file. In an attempt to make the changes to the *FLVM* as simple as possible the debugger handles the files associated with tracing. However, marshaling and unmarshaling the event is time consuming. One optimization we foresee is moving the file handling to the *FLVM* and having it write the trace steps directly rather than through the debugger.

To minimize file sizes only the first step of the file contains the entire term. Subsequent steps contain the difference from the previous term represented as a position and replacement. To get a step from the file the first term is parsed out. Then for each step up to the desired one the given position in the term is replaced with the replacement. Since this can be time consuming (it can take as long as the entire computation) the files are broken up so that they contain at most 50 steps.

The execution of a non-deterministic step fires a non-deterministic trace step event to the debugger. The debugger creates new traces for each of the possible replacements. A non-deterministic trace step is just a collection of traces.

Breakpoints Each time the *FLVM* replaces a term it checks if either the term or the symbol (a function) at the root of the term has a breakpoint set on it. If it does then the *FLVM* fires a breakpoint event to the debugger, suspends the thread that evaluates terms, and wakes up the thread that reads input from the user.

Eager Evaluation To perform eager evaluation of a term a new evaluation thread to work on a deep copy of the term to be evaluated is created. Creating the deep copy makes sure that there are no shared terms between the copy and any other term in the *FLVM*. This is necessary to preserve the lazy evaluation. When the newly created evaluation thread finishes evaluating the term to normal form an evaluation event is fired to the debugger. The *FLVM* holds on to the term to evaluate and its replacement until it knows if the user has selected to replace the term or not.

5 Example

The following example demonstrates the computation structure of TeaBag. Consider the *Dutch National Flag* program of Section 2 where lines 10-12 are replaced by:

```
10    solve flag | flag =:= blue:y ++ white:z
11               = solve (white:y ++ blue:z)
12               where y,z free
```

i.e., the prefix, `x`, of a blue-white pair of pebbles out of place has been forgotten. When `solve [white,red,blue,white]` is run using the above program the result is a failure. To find this bug we first generated a trace of `solve [white,red,blue,white]`. Part of the structure for this trace is shown in figure 3. We decided to follow the path through the computation structure that

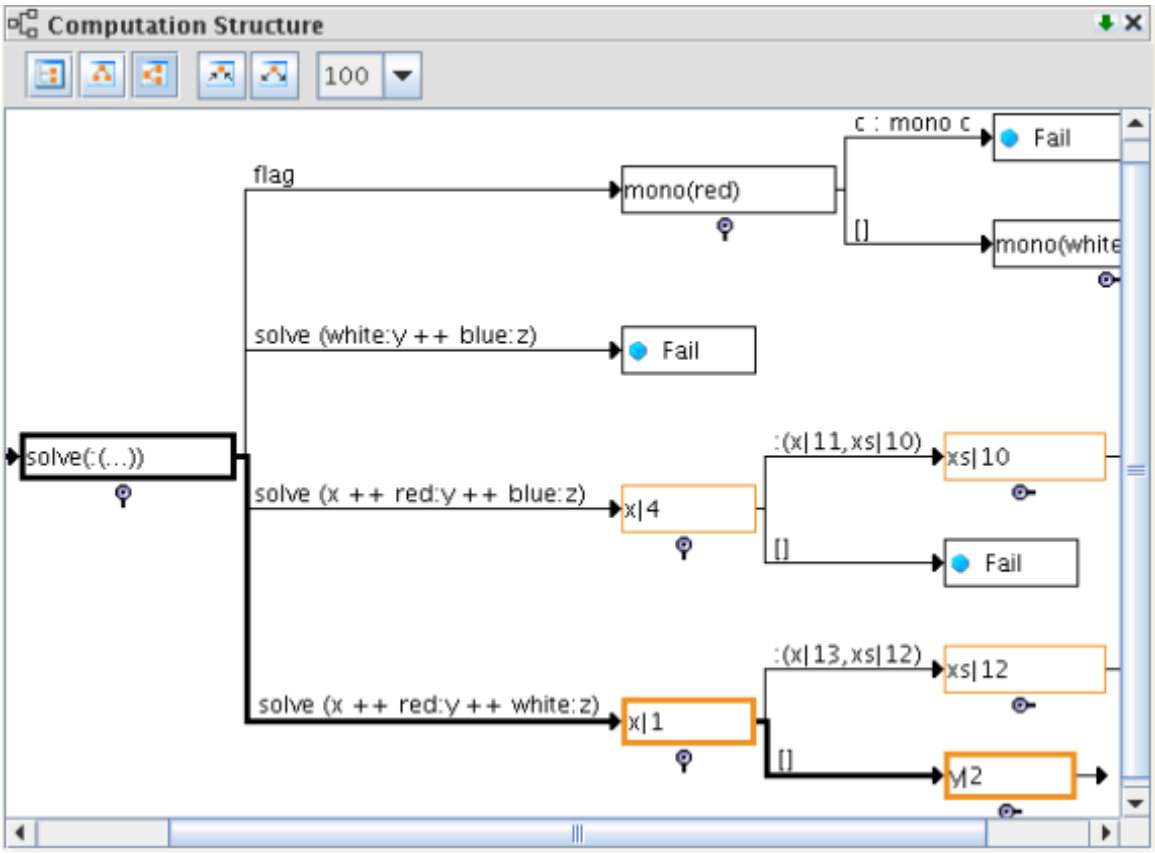

Fig. 3. Computation Structure for Buggy Dutch National Flag Program

we thought should have led to a solution. Since the choices along this path should have led to a solution, examining the rewriting and narrowing steps on this path will tell us where the bug is located. In this example we realized that to get a solution the first and third rules of `solve` would need to be executed. The first rule should swap `red` and `white` and the third rule should swap `blue` and `white`. Either order of applying these rules should led to a solution. We arbitrarily decided to look at the path that is generated from applying the first rule and then applying the third rule. In order for the first rule to swap `red` and `white` it must find bindings for the free variables `x`, `y`, and `z` that satisfy `flag =:= x ++ white:y ++ red:z`. Since `flag` is `[white,red,blue,white]` binding `x` to `[]`, `y` to `[]`, and `z` to `[blue,white]` will work. We used this information to find the path thought the computation structure for applying the first rule.

There were two ways we could have found this trace path in the computation structure. We could have stepped through the trace in the trace browser one step at the time. Then when a non-deterministic step was made we would have been prompted to pick a branch to follow. By selecting the appropriate branches we would have followed the trace corresponding to the path in the computation structure that we wanted. Because there is more contextual information, this method works well when the correct choices at each non-deterministic step are difficult to determine. The second

way to find a trace path in the computation structure is to follow branches through the tree. Each node in the tree has one branch for each possible replacement. By knowing what the replacements should be, the correct branch at each node can be selected. Thus, this method works well when the correct choices at each non-deterministic step are known. We chose the second option since we knew which choices we wanted to examine. The first branch we followed was the one for swapping `red` and `white`. The next branch in the computation structure was for binding the variable `x`. One of the branches was for binding `x` to the empty list and the other branch for the non-empty list. The same was true of `y`. So we chose the empty list for both. We saw that there was no choice to be made for `z` since our choices for `x` and `y` forced `z` to be bound to `[blue,white]`. The next choice we had to make was for `solve(:(...))`.

At this point we needed to see what the term for the trace looked like to see if `red` and `white` were actually swapped like we thought they should have been. We were expecting that the term would be `solve [red,white,blue,white]`. To check this, we right clicked on the node in the computation structure for `solve(:(...))` and selected "*Move trace to this step*." This updated the trace browser to show the trace along the path we have chosen so far and to display the step for this choice as the current step. Figure 4 shows the trace at this point. The upper left corner is the trace step for picking a non-deterministic choice for `solve(:(...))`, the upper right corner is the source code with the code for the choice in the trace browser highlighted, and the lower panel shows the computation structure with the current path through the trace highlighted. Since we were interested in whether or not `red` and `white` were actually swapped we moved the trace back one step. This step showed that `red` and `white` had been swapped. The term for this step was `solve(red : [] ++ [white,blue,white])`.

Now `blue` and `white` must be swapped to get a solution. So we continued along the path we had followed so far choosing the path in the computation structure for swapping `blue` and `white`. We noticed immediately that this path leads to a failure as can be seen in figure 4. This caused us to think that the bug for this program was somewhere between choosing to swap `blue` and `white` and the failure. So we stepped through the trace one step at a time in the trace browser starting with the step for choosing to swap `blue` and `white`. After looking at five trace steps we noticed that for the condition to evaluate to a success, `[red,...]` must be equal to `[blue,...]`. Obviously, this can never happen since `red` can not be equal to `blue`. With this information we then went back in the trace to see why `blue` must be equal to `red`. We went back to the previous choice step to examine how the condition was created. Here we noticed that the condition is `[] ++ (red : [] ++ [white,blue,white]) =:= [blue,y,white,z]`. At this point we realized that there is no way for `red` to match anything on the right hand side since there is no free variable for it. So we added a free variable to this rule giving us the code presented in Section 2.

We could have also found this bug by looking at the path in the computation structure that corresponds to applying the third rule of `solve` and then the first rule of `solve`. If we had chosen this path then we would have found the bug much faster since the path for applying the third rule of `solve` immediately leads to a failure as can be seen in figure 3.

6 Related Work

Functional logic languages borrow ideas from both functional and logic languages. Functional logic language debuggers are no different. They borrow ideas from debugging functional languages and from debugging logic languages. Four different debugging techniques have been applied to debugging functional logic languages. The first one is tracing. Tracing is a debugging technique used for debugging functional programs [14,15,38,42]. Tracers show each step of a computation to the pro-

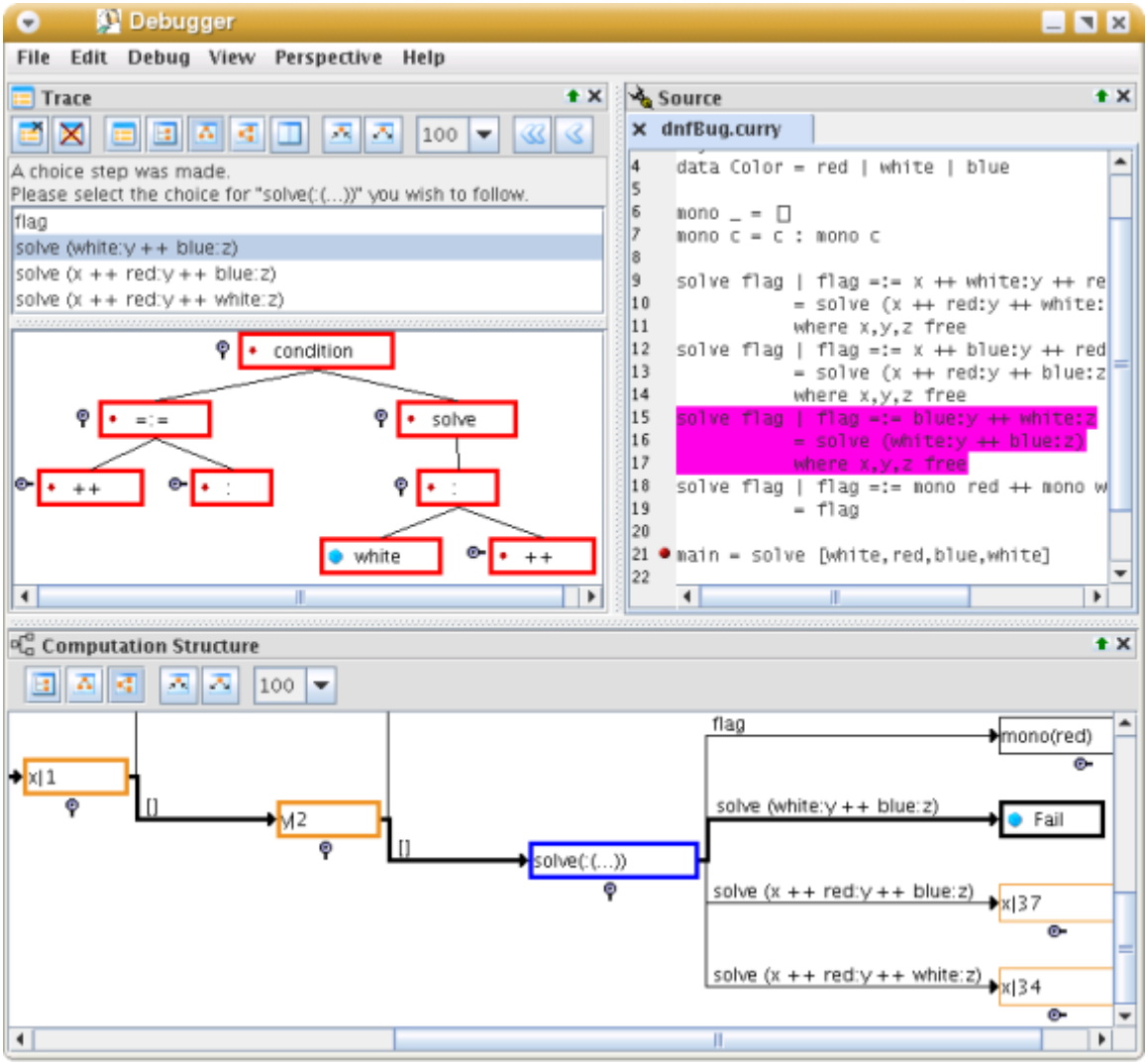

Fig. 4. Trace of Buggy Dutch National Flag Program

grammer. Typically these steps are reductions, though they do not have to be. For example, tracing has been used to trace the redex trail of Haskell programs [38,14]. Likewise, CIDER [22] is a functional logic debugger for Curry based on tracing. Trace steps in CIDER are narrowing or rewriting steps. A specialized form of tracing called box-oriented debugging is the second type of functional logic debugging. Box-oriented debugging was first developed by Byrd [11] for debugging Prolog programs. Box-oriented debuggers trace goals in logic programs. Box-oriented debugging has been extended to functional logic languages by Hanus and Josephs [21] and by Arenas-Sánchez and Gil-Luezas [9]. The third type of functional logic debugger is observation debugging. This idea was first developed by Gill [18] for Haskell. It was latter incorporated into the Haskell tracer called Hat [40]. Observational debugging works by letting the programmer see the intermediate data structures that are passed between functions. Recently, Braßel, et al. extended this idea to functional logic languages by handling non-deterministic search, logical variables, concurrency, and constraints [10]. The final type of functional logic debugger is algorithmic. This idea was initially proposed by Shapiro [34] for debugging Prolog programs. The idea of algorithmic debuggers has been used in functional [31,32,37], logic [25,34,39], and functional logic debuggers [3,4,12,13,30]. Algorithmic

debuggers work by using the declarative semantics of the program. An oracle, typically the user, is asked questions about the intended meaning of their program in an automated way until the debugger can point out where the bug is located. Some examples of algorithmic debuggers for functional logic languages are Buggy [3], an accessory of the Münster compiler [12], and the debugger in the Toy system [13].

At first glance it may seems surprising that almost all functional logic language debuggers are algorithmic. Why haven't more debugging ideas from the functional and logic communities be explored in functional logic languages? We believe the reason for this is that algorithmic debuggers have been proven to work in both functional and logic languages so it is only natural that they would also work in functional logic languages. Most of the other debugging schemes for functional and logic languages have only been shown to work in their respective family of languages. Thus directly using one of those schemes for debugging functional logic languages will only debug "half" of the language. For example, CIDER [22] contains a tracer of rewriting and narrowing steps for debugging. This tracer works fine for tracing deterministic programs. However, it becomes difficult to use in non-deterministic programs. It shows the trace of non-determinism as a deterministic backtracking step which can be difficult to follow [11]. The tracer in CIDER is not as effective on non-deterministic programs as it is on deterministic programs.

For a functional logic debugger to be useful it has to be able to deal with both deterministic and non-deterministic features that real programs use [12]. We have applied this principle to tracing rewriting and narrowing steps in functional logic programs. We created a functional logic tracer that can be used to trace both deterministic and non-deterministic programs. To do this we borrowed the traditional tracing of reduction steps idea from functional programming [42] and combined it with the structure of the search space [33]. We believe that our approach is the first attempt to exploit this combination to trace the steps in a computation.

TeaBag is a debugger for Curry. There are three other debuggers for Curry: Münster [12], COOSy [10], and CIDER [22]. Münster is a compiler for Curry that contains a declarative debugger of wrong answers. TeaBag and Münster take different approaches to debugging Curry. Münster uses the declarative semantics of the program for debugging it. TeaBag uses the runtime narrowing and rewriting steps. Münster systematically asks the user questions until it can deduce where the bug is located. TeaBag, on the other hand, lets the user investigate how their program is being executed to find the bug. Given these differences Münster and TeaBag should be viewed as complementary, rather than competing, debuggers. Like Münster, COOSy takes a different approach to debugging from TeaBag. COOSy is an observational debugger. Thus COOSy lets the user view the values of expressions. To handle the non-deterministic aspects of functional logic programs COOSy extended Gill's observational debugging idea [18] to handle non-deterministic search, logical variables, concurrency, and constraints. Like TeaBag, COOSy extended a functional language debugging idea to handle all aspects of functional logic languages. Alternate non-deterministic choices in COOSy are shown in a group and the bindings of logic variables are displayed. TeaBag is much more like CIDER in that both of them use tracing for their debugger. CIDER is an IDE for Curry that contains a debugger which uses tracing to debug Curry. However, CIDER does not provide context hiding, highlighting, or a trace structure suitable for debugging non-deterministic programs. Thus CIDER is more difficult to use than TeaBag for debugging large programs and non-deterministic programs. While the sole focus of TeaBag is debugging, CIDER focuses on program development of which debugging is just one aspect. Thus CIDER includes analysis, editing, and compilation tools which are not in TeaBag.

TeaBag has a unique place in the current landscape of debuggers for functional logic languages. It is the only tracer of narrowing steps we are aware of that truly handles both deterministic and

non-deterministic features found in real functional logic programs. For more detailed information, source code, and to download TeaBag refer to [24].

7 Conclusion

We have presented, TeaBag, a debugger for functional logic computations. TeaBag has been developed as an accessory of the *FLVM*, a virtual machine intended for the execution of Curry programs. A distinctive characteristic of this machine is its operational completeness. This means that the strategy for the execution of non-deterministic steps is concurrency, rather than backtracking. This strategy poses novel demands on a debugger.

Our debugger has both typical features of functional and logic debuggers, specifically features found in tracers and/or runtime debuggers, and novel features for displaying and managing non-determinism. In addition to standard features such as context elimination, highlighting and breakpoints on functions and terms, the user can view the non-deterministic steps of a computation and display only traces that make certain user-selected steps. To our knowledge, this is the first debugger with this capability.

References

1. H. Aït-Kaci. An overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
3. M. Alpuente and F. Correa. Buggy user's manual. http://www.dsic.upv.es/users/elp/buggy/.
4. M. Alpuente, F. Correa, and M. Falaschi. A debugging scheme for functional logic programs. In *Proc. of the 10th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, 2001.
5. M. Alpuente, F. Correa, and M. Falaschi. A declarative debugging scheme for functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, 2001.
6. S. Antoy. Optimal non-deterministic functional logic computations. In *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298, pages 16–30, Southampton, UK, 9 1997. Springer LNCS.
7. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. Architecture of a virtual machine for functional logic computations, October 2003. Preliminary manuscript, available at `http://www.cs.pdx.edu/~antoy/homepage/publications.html`.
8. S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.
9. P. Arenas-Sánchez and A. Gil-Luezas. A debugging model for lazy functional logic languages. Technical Report DIA 94/6, 1994.
10. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proceedings of the 6th Internationl Symposium on Practical Aspects of Declarative Languages*, Dallas, Texas, USA, June 2004. To appear.
11. L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138, 1980.
12. R. Caballero and W. Lux. Declarative debugging for encapsulated search. In Marco Comini and Moreno Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.
13. R. Caballero and M. Rodrguez-Artalejo. A declarative debugging system for lazy functional logic programs. In Michael Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier, 2002.
14. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 176–193, Aachen, Germany, September 2000.
15. O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In Ricardo Pena and Thomas Arts, editors, *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, LNCS 2670, pages 165–181, March 2003. Madrid, Spain, 16–18 September 2002.
16. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001.
18. A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham.*, 2000.
19. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
20. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2003.
21. M. Hanus and B. Josephs. A debugging model for functional logic programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 28–43. Springer LNCS 714, 1993.
22. M. Hanus and J. Koj. An integrated development environment for declarative multi-paradigm programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pages 1–14, Paphos (Cyprus), 2001. Also available from the Computing Research Repository (CoRR) at `http://arXiv.org/abs/cs.PL/0111039`.
23. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
24. S. Johnson. TeaBag: A debugger for Curry. Master's thesis, Portland State University, expected September 2004. Available at `http://redstar.cs.pdx.edu/~stephenj/teabag/`.
25. G. Kkai, L. Harmath, and T. Gyimthy. Algorithmic debugging and testing of Prolog programs. In *Proceedings of ICLP '97 The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments*, July 1997.
26. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
27. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
28. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer Verlag, 1999.
29. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
30. L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In Graham Forsyth and Moonis Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers*, volume 2, pages 91–99, Melbourne, June 1995. DSTO General Document 51.
31. H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, November 2001.
32. B. Pope. Buddha: A declarative debugger for Haskell. Technical report, University of Melbourne, 1998.
33. C. Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
34. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
35. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
36. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
37. J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional langauges. In Mireille Ducassé, editor, *Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995. IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France.
38. J. Sparud and C. Runciman. Tracing lazy functional computations using Redex Trails. In *PLILP*, pages 291–308, 1997.
39. A. Thompson and L. Naish. A guide to the NU-Prolog Debugging Environment. Technical Report 96/38, Department of Computer Science, University of Melbourne, Melbourne, Australia, August 1997.
40. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: A new Hat. In *Proceedings of the Haskell Workshop 2001*, Firenze, Italy, 2001. Final version to appear in ENTCS.
41. D.H.D. Warren. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.
42. R. Watson. *Tracing Lazy Evaluation by Program Transformations*. PhD thesis, School of Multimedia and Information Technology, Southern Cross University, 1997.