TEABAG: A DEBUGGER FOR CURRY

by

STEPHEN LEE JOHNSON

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2004

ABSTRACT

An abstract of the thesis of Stephen Lee Johnson for the Master of Science in Computer Science presented July 2, 2004.

Title: TeaBag: A Debugger for Curry

This thesis describes TeaBag, which is a debugger for functional logic computations. TeaBag 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. This thesis describes the features of TeaBag, 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. This thesis also describes TeaBag's architecture and its interaction with the associated virtual machine. Finally, some debugging sessions of defective programs are presented to demonstrate TeaBag's ability to locate bugs.

A distinctive feature of TeaBag is how it presents non-deterministic trace steps of an expression evaluation trace to the user. In the past expression evaluation traces were linearized via backtracking. However, the presence of backtracking makes linear traces difficult to follow. TeaBag does not present backtracking to the user. Rather TeaBag presents the trace in two parts. One part is the search space which has a tree structure and the other part is a linear sequence of steps for *one* path through the search space.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Sergio Antoy. He has been a source of inspiration and encouragement. I also greatly appreciate his helpful feedback and ideas on TeaBag and this thesis.

I would also like to thank Dr. James Hein and Dr. Xiaoyu Song for being on my thesis committee.

Next, I want to thank Dr. Andrew Tolmach for his helpful insights on TeaBag and for all his great and beneficial classes related to programming languages.

Also, I would like to thank Jimeng Liu for helping me learn and modify the virtual machine.

Finally, I would like to thank Marius Nita, Jason Wilcox, and Pravin Damle for all of their helpful discussions on Curry and functional logic programming.

Thank You!

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

Contents

List of Figures

Chapter 1

Introduction

Since the dawn of programming, programmers have strove to write bug free code. Yet, despite all their best attempts at writing bug free code, bugs continue to come to life, even in code thought to be perfect. These bugs arise in code, generally not because of programmer incompetence or laziness, but because of the complexity of software. As software systems get larger they get more and more complex. This complexity makes writing bug free code difficult if not impossible. Thus, bugs live in software. Exterminating these critters requires three steps. The first one is noticing that the bug exists. This involves testing the program and seeing the effect, e.g. the infamous "blue screen of death", the bug has on the software. The second step is finding where this bug lives. Once, the home of the bug is located the bug can be killed by "fixing" the code. This thesis deals with finding where bugs live for functional logic programs.

One approach used to mitigate the "bug problem" in software development is using functional languages such as SML [42] and Haskell [55]. One of the adages of functional programmers is "well-typed programs do not go wrong". Unfortunately, in this statement wrong does not mean bug free. Rather, not going wrong means that a program is never in a state that is not well-defined by the language. It says nothing about whether or not the program does what it was expected to do. However, this does not mean all is lost. If the “blue screen of death”, or more realistically, dereferencing a null pointer, is not defined by the language, then a well-typed program will not do those things. This helps to reduce the number of bugs in a program. Unfortunately, it does not reduce the number to zero.

Another step towards mitigating the “bug problem” in software is providing developers with tools that help them find, locate, and fix bugs. The tools that help programmers locate bugs are called debuggers. These have been deemed so critical, that Wadler says [75], “To be usable, a language system must be accompanied by a debugger and profiler.” Debuggers help the programmer deal with the complexity of software by assisting them in finding the location of bugs. This assistance may be algorithmic, where the debugger helps direct the programmer to the location of the bug, or the debugger may let the programmer visualize intermediate states of their program.

This thesis is about TeaBag (The Errors And Bugs Are Gone!) which is a debugger for a functional logic language called Curry. TeaBag mixes runtime debugging and tracing to provide the user with powerful tools to examine how their program executes. One of the key features of TeaBag is its presentation of non-deterministic computations in the tracer. Previous tracers linearized the the steps via backtracking. TeaBag presents the trace in two parts. One part is the search space which has a tree structure and the other part is a linear sequence of steps for *one* path through the search space. A brief description of TeaBag can be found in [15].

Functional logic languages aim at integrating features from functional and logic languages together into one programming paradigm. From functional languages they take features such as algebraic data types, first class functions, polymorphic typing, monadic I/O, and lazy evaluation. From logic languages they take features such as non-determinism, search, and logic variables. Like all other languages, functional logic languages also need debuggers.

The goals of this thesis are three-fold. First and foremost, this thesis describes TeaBag. TeaBag is the implementation that all of our research in debugging revolved around. This thesis describes the features and architecture of TeaBag. Second, this thesis discusses the strategy we used for debugging functional logic programs. A debugging strategy is debugging concepts that can be incorporated into other debuggers. Related to the strategy, this thesis will discuss how to trace and present non-deterministic steps. Finally, this thesis will describe the interface between TeaBag and the virtual machine that it interacts with. This interface can be used by another virtual machine to interact with TeaBag.

1.1 Contributions of the Thesis

This thesis provides the following contributions.

- A strategy suitable for tracing the narrowing steps of functional logic programs is described. Useful traces of functional logic programs must present both the functional and logic aspects of the language to the user in a meaningful way. The strategy described in this thesis does precisely this for narrowing step tracers. Specifically, we break the trace presentation into two parts. One part shows the user the search space. Another part shows the user the steps along one of the paths in the search space.

- This thesis also discusses the advantages of mixing runtime debugging and tracing for debugging functional logic programs. Typically, debuggers for functional, logic, and functional logic programs are either runtime debuggers or tracers. This thesis investigates blurring the line between runtime debugging and tracing. That is, TeaBag has runtime debugging features that interact with the tracer.

1.2 Overview of the Thesis

Chapter 2: Background The background gives the context our work occurred in. This chapter starts out by giving an overview of functional logic languages. It also discusses a particular functional logic language virtual machine called the FLVM that TeaBag was designed to work. Finally, this chapter provides an overview of existing logic, lazy functional, and functional logic debuggers.

Chapter 3: Non-Deterministic Tracing The main research contributions of TeaBag are related to non-deterministic trace generation and presentation. This chapter will start by defining a *trace* and a *trace view*. Then it will review the existing traces and trace views for functional logic languages. Finally, this chapter will present the trace and trace view used in TeaBag.

Chapter 4: Features This chapter discusses the features of TeaBag. There are three main categories the features fall into; runtime, tracing, and general features.

The runtime features can be used while the program is running to debug it. The tracing features provide a way to record and view the narrowing steps involved in the computation of a term. The general features, e.g. highlighting, are features that apply to both the runtime debugging and tracing.

Chapter 5: Architecture The architecture of TeaBag is described in this chapter. Specifically, this chapter focuses on two aspects of the architecture. Firstly, it talks about the main subsystems and packages of TeaBag and how they communicate. Then it describes in detail the socket interface used to communicate between TeaBag and the virtual machine.

Chapter 6: Examples This chapter gives examples of using TeaBag to debug Curry programs. Examples, of a wrong answer, a missing answer, and a non-terminating error are given. Also, this chapter provides an example of using TeaBag with a non-trivial Curry program. Finally, an example of using TeaBag to debug a program with a non-trivial search space is presented.

Chapter 7: Conclusion This chapters offers some concluding remarks. It also discusses the related and future work.

Chapter 2

Background

2.1 Functional Logic Languages

2.1.1 What are Functional Logic Languages

Functional logic languages integrate features from functional and logic languages into one programming paradigm. From functional languages they take features such as algebraic data types, first class functions, polymorphic typing, monadic I/O, and lazy evaluation. From logic languages they take features such as non-determinism, search, and logic variables.

Functional languages, such as Haskell [55] and SML [42], are made up of functions. They focus on what a function computes and not how it computes it. This frees the programmer from concentrating on the details of the computation.

Logic languages, like Prolog [53, 70], determine the truth of a proposition. A logic program is made up of predicates. The user asks the logic program if a certain proposition is satisfiable given a particular set of predicates. The result is either *yes* or *no*. Logic languages like Prolog allow the use of logic variables. These variables are *free* to be bound to a value. The runtime system tries to find a binding for the *free* variables that makes the proposition true. Also, logic languages introduce non-determinism in the sense that a given logic variable may have several bindings that satisfy a proposition.

Functional logic languages, such as Curry [39] and Toy [41], integrate the features of functional and logic languages into one language. Thus, functional logic languages are made up of functions. They allow the programmer to focus on what a function computes and not how the function computes the value. They also introduce logic variables and non-determinism. Combining these two programming paradigms together takes the good features of both languages and brings them into one programming paradigm. Logic languages are nice to use when search is involved. However, they are cumbersome to use for defining "normal" functions since a logic language only gives the truth of a statement. Combining functional and logic languages together allows the programmer to define "normal" functions and make use of the powerful search mechanisms found in logic languages. Functional logic languages let the user have the power of builtin search with all of the convenient programming features of functional languages.

Functional logic languages introduce two features into functional computations. They introduce non-determinism and logic variables.

Non-Determinism Non-determinism allows a function to have multiple values for a given set of arguments. This can occur either from explicitly making multiple definitions for a function or because there are multiple possible instantiations for a logic variable. In typical programming languages this is not allowed. Typically, functions are not allowed to have multiple definitions. At first glance this seem very reasonable. Obviously one would not want a function called `square` to be defined as either the square or double of its argument. When programming, one would clearly want `square` to always square its argument and not have the possibility of doing something else. Even more fundamentally, one may think that it is easier to reason about functions that return only one value for a given set of arguments. While it is true that there are many examples like `square` where non-determinism is not desired there are also many examples where non-determinism is very convenient. For example, consider defining a function called `childOf` that takes one argument, a parent, and returns a child of that parent. Now say Jim is defined to have two children, Alice and Bob. When the function `childOf Jim` is called it may return Alice or it may return Bob. There is no way to know which one it will return. However, if there is some constraint about the kind of child that it can return then it will return the right one. For example, the function `maleChildOf` will return one of the male children of its argument. Having non-determinism built into the language in this way is convenient for the programmer. Without non-determinism the programmer would write the same program where they basically encode this behavior into their program. Doing this requires more code. The programmer must explicitly maintain sets of the possible results for each function and explicitly search those sets. It also makes the code more difficult to read and reason about. Also, adding or removing constraints would be cumbersome. Appropriately using the non-deterministic features of a functional-logic language can actually improve, rather than degrade, the programmers ability to reason about their program.

Logic Variables Non-determinism is not only incorporated into functional logic languages by allowing multiple definitions for the same function, it is also added in via logic variables. Logic variables give the programmer a way to let the runtime system search for an instantiation that will satisfy a set of constraints. For example, the function `nephew` could be defined in Curry as follows. (See section 2.1.3 for a description of Curry).

```
nephew p | s =:= siblingOf p   &
           n =:= maleChildOf s
         = n where s,n free
nephew p | o =:= spouseOf p    &
           s =:= siblingOf o   &
           n =:= maleChildOf s &
         = n where o,s,n free
```

This definition makes use of both kinds of non-determinism. The first definition of `nephew` returns one of the nephews from the brothers and sisters of `p` and the second definition of `nephew` returns one of the nephews from the brothers and sisters of the spouse of `p`. When a programmer calls `nephew Jim` it will return one of the nephews of Jim. This nephew could be from either Jim or Jim's spouses side of the family. Notice, that there are *free* variables. Specifically, in the first definition of `nephew` the variables `s` and `n` are free. Thus, at runtime the system will try to find a binding for these variables that satisfies the constraints. For example, say we have called `nephew Jim` and Jim has a brother named Joe and a sister named Jill. Also, lets says that Joe has a daughter and that Jill has both a son and daughter. Since, Joe does not have a son he can not be the instantiation of `s` since `s` must have a male child. Thus, only Jill can be instantiated for `s`.

So even though calling `siblingOf Jim` could return either Joe or Jill it will only return Jill when it is called in this context since that is the only instantiation that will satisfy the constraints. In this example, there was only one possible answer for `nephew Jim`. Now let's say that Joe has a new baby boy. So now both Joe and Jill are valid instantiations for `s`. Thus, `nephew Jim` may return either the son of Joe or the son of Jill.

2.1.2 Implementation of Functional Logic Languages

Typically functional logic programs are modeled by term rewriting systems (TRS) which are evaluated using narrowing and residuation (see [34] for a survey). It is also possible to use the λ-calculus to model functional logic programs [74]. Since the λ-calculus approach is relatively new and not in common use (at least at the time of writing this thesis) we will focus on TRSs.

Term Rewriting Systems Term rewriting systems are at the heart of most computational models for functional logic languages. Consider the following set of basic algebra rules combined with the standard multiplication and addition tables for the integers 0 through 10.

$$\begin{aligned} a * (b * c) &= (a * b) * c \\ a * (b + c) &= (a * b) + (a * c) \\ a * b &= b * a \\ (a) &= a \end{aligned}$$

Now consider evaluating the expression $(5+2)*(1+4)$. It could be evaluated as follows.

$$(5+2)*(1+4) \to^1 (7)*(1+4) \to^2 7*(1+4) \to^3 (7*1)+(7*4) \to^4$$
$$(7)+(7*4) \to^5 7+(7*4) \to^6 7+(28) \to^7 7+28 \to^8 35$$

This computation is carried out by first *rewriting* $5+2$ to 7 by looking up $5+2$ in the addition table. Then (7) is *rewritten* to 7 using the $(a) = a$ rule from above. Next, $7*(1+4)$ is *rewritten* to $(7*1)+(7*4)$ using the $a*(b+c) = (a*b)+(a*c)$ rule. The rest of this computation continues in a similar manner until the result 35 is obtained. Since there is no rule that can rewrite 35 we say that 35 is a *normal form.* Notice here that we typically think of each equation as being equal to each other. Of course this is true since each equation in the computation evaluates to 35. However, we can also think of this computation as a sequence of rewrite steps. That is, each step in the computation uses the syntax of the equation and a set of rules to determine the next step in the computation. So the second rewrite step uses the rule $(a) \to a$ to rewrite $(7)*(1+4)$ to $7*(1+4)$. Here we say that (7) is the redex. The redex, reducible expression, is the expression or subexpression that is rewritten by the rewrite rule. Also, notice that the computation is just a sequence of pure symbolic manipulations. That is, we intuitively have a notion of addition and multiplication but that is not used. Rather rewrite steps are used for each addition and multiplication. Thus, we easily could have defined $5+2 = 1$. Then the normal form of $(5+2)*(1+4)$ would be 5. Intuitively this does not seem right. Operationally, this is correct since a term rewriting system just manipulates symbols. A TRS works with a given set of rewrite rules and has no notion of builtin operations like $+$ or $*$. So if the addition table is only defined for the integers 0 to 10 then $11 + 2$ and $3.4 + 1$ do not have any rules that can reduce them.

Formally, a term rewriting system consists of two parts, a *signature* Σ and a set of *rules* R. A signature is a set of *symbols* where each symbol has an arity. The symbols are the defined operations for a given TRS. The arity of the symbol is the number of arguments that it can be applied to. In the above example, $+$ is a symbol with arity 2. Before describing the rules of a TRS the concept of a term must be understood. A *term* is made up of symbols and variables. *Terms* are only defined in relation to a signature Σ. So the set of terms, $\mathcal{T}$, constructed from a signature, Σ, and an infinite set of variables, $\mathcal{X}$, is defined as

$$\mathcal{T} = \{t | t = \begin{cases} x & x \in \mathcal{X} \\ f(t_1..t_n) & f \in \Sigma \ \ \& \ \ \texttt{arity}(f) = n \ \ \& \ \ t_1 \in \mathcal{T} \ \ .. \ \ \& \ \ t_n \in \mathcal{T} \end{cases} \}$$

So a *term* is either a variable or it is an operation of arity n applied to n terms. Constants are defined as operations of arity 0.

A set of rewrite rules R is a set of pairs $l \rightarrow r$ where l and r are terms, the variables appearing in r is a subset of variables in l, and l is not a variable. The intuition behind a rewrite rule is that terms "matching" the left hand side, l, are rewritten to the right hand side, r. To formalize this idea we need a few more definitions. Firstly, the *occurrence* or *position* of a term u in a term t defines where u is located in t. The *occurrence* is represented as a sequence of integers $\langle p_1...p_k \rangle$ that defines the occurrence of u in t by: if $k = 0$ then $u = t$ and otherwise if $t = f(t_1...t_k)$ then u is the occurrence of $\langle p_2...p_k \rangle$ in t_{p_1}. For example, the occurrence of the term $+(1,2)$ in $*(+(5,2),+(\mathit{+(1,2)},3))$ is $\langle 2,1\rangle$. The term at occurrence p in term t is denoted as $t|_p$. When terms are rewritten part of the term is replaced with a new term. We write $t[u]_p$ for replacing $t|_p$ with u in t.

A *substitution* maps variables to terms. A substitution is first defined on variables and then extended to terms. A substitution, σ, applied to the term t is defined as

$$\sigma(t) = \begin{cases} t' & if\ t \in \mathcal{X}\ \&\ (t \mapsto t') \in \sigma \\ f(\sigma(t_1)...\sigma(t_n)) & if\ t = f(t_1...t_n) \end{cases}$$

Now we can formalize our informal notion of rewriting. If we have a term t, a rule $l \rightarrow r$, a position p in t, and a substitution σ such that $t|_p = \sigma(l)$ then t can be rewritten to $t[\sigma(r)]_p$. This is written as $t \rightarrow_{p,l\rightarrow r,\sigma} u$. So a term t "matches" a rule $l \rightarrow r$ if there is some substitution for l that produces t. For example, the rule $a*b \rightarrow b*a$ matches the term $(1+2)*3$ with the substitution $[a \mapsto (1+2), b \mapsto 3]$. Now we can use this substitution on the right hand side of the rule to obtain the replacement, $3*(1+2)$.

Each expression in a term that can be reduced with a rule is called a *redex* (reducible expression). In the $t \rightarrow_{p,l\rightarrow r,\sigma} u$ relation the redex is $t|_p$. When an expression contains no redexes it is in *normal form*. It is possible for a term to contain multiple redexes. For example, $3*(1+2)$ contains the redex $1+2$ using the addition table and the redex $3*(1+2)$ using the distributive rule. A *strategy* chooses which redex to reduce.

Most implementations of functional logic languages put a further restriction on TRSs. They also require them to be *left linear*. A *left linear* TRS is one where variables occur no more than once in the left hand side of the rewrite rules. Also, most implementations of functional logic languages work with constructor term rewriting systems. A constructor TRS breaks the signature up into two sets. There is one set, $\mathcal{C}$, of data constructors and another set, $\mathcal{D}$, of defined operations. The left hand side l of a rule, $l \rightarrow r$, must be a term, $f(t_1...t_n)$ where $f \in \mathcal{D}$ and every symbol occurring in $t_1...t_n$ is either in the set of variables, $\mathcal{X}$, or in $\mathcal{C}$. A *redex pattern* for a reduction $t \rightarrow_{p,l\rightarrow r,\sigma} u$ is the set of constructors in l not occurring in σ. Intuitively, a constructor TRS has three kinds of symbols. It has variable symbols, operations symbols, and constructor symbols. A constructor symbols defines data like a list or tree. Thus, there are no occurrences of defined operations for the data itself. The operation symbols define the operations to be performed on data.

A more detailed treatment of term rewriting systems can be found in [18, 20].

Narrowing *Narrowing* [60, 34, 13] and *residuation* [3] are the glue between functional computations and logic variables. When a computation of an expression cannot continue due to the presence of a logic variable, narrowing *non-deterministically instantiates* that variable. Residuation *delays* the evaluation of that expression and starts working on another part of the program in hopes that the variable will become instantiated by some other expression.

A narrowing step consists of two parts. First it instantiates logic variables of a term and then applies a rewrite rule to one of the resulting subterms. The instantiation of logic variables comes from a substitution which can be the identity. When the identity substitution is used the term does not change and the narrowing step has the effect of just performing the rewrite step. Thus, narrowing is a generalization of rewriting in that rewriting is narrowing with the identity substitution.

Loosely speaking, a term t is narrowable if applying some substitution to it makes it "match" the left hand side of a rewrite rule. More formally, a term t is *narrowable* to a term s if there exists a position p in t such that $t|_p$ is not a variable, a variant $l \rightarrow r$ of a rewrite rule in $\mathcal{R}$ where there are no common variables in t and $l \rightarrow r$, and a unifier σ of $t|_p$ and l such that $s = \sigma(t[r]_p)$. We say that $t|_p$ is a *narrex* (narrowable expression). A variant of a rewrite rule means that the names of the variables in the rule can be changed. That is, $l^{'} \rightarrow r^{'}$ is a *variant* of $l \rightarrow r$ if there exists substitutions σ and $\sigma^{'}$ such that $l \rightarrow r = \sigma(l^{'} \rightarrow r^{'})$ and $l^{'} \rightarrow r^{'} = \sigma^{'}(l \rightarrow r)$. The following example will help clarify the definition of narrowable.

```
data Nat = Z | S Nat

add :: Nat -> Nat -> Nat
add Z n     = n
add (S m) n = S (add m n)
```

Now consider narrowing the term t where t is `add (add x (S Z)) (S Z)` and where `x` is a logic variable. If p is $\langle 1 \rangle$ then $t|_p$ is `add x (S Z)`. There are two rules that can unify with $t|_p$. The left hand side of the rule $\texttt{add Z n} \rightarrow \texttt{n}$ unifies with `add x (S Z)` with unifier $\{\texttt{x} \mapsto \texttt{Z}, \texttt{n} \mapsto \texttt{S Z}\}$ and the left hand side of the rule $\texttt{add (S m) n} \rightarrow \texttt{S (add m n)}$ unifies with `add x (S Z)` with unifier $\{\texttt{x} \mapsto (\texttt{S x}^{'}), \texttt{m} \mapsto \texttt{x}^{'}, \texttt{n} \mapsto \texttt{S Z}\}$ where $\texttt{x}^{'}$ is a new logic variable. So if $\sigma = \{\texttt{x} \mapsto (\texttt{S x}^{'}), \texttt{m} \mapsto \texttt{x}^{'}, \texttt{n} \mapsto$

S Z} and $l \rightarrow r$ is add (S m) n $\rightarrow$ S (add m n) then

$$\sigma(t[r]_p) = \sigma(\texttt{add (S (add m n)) (S Z)}) = \texttt{add (S (add x}^{'} \texttt{ (S Z)) (S Z)}$$

Thus, `add (add x (S Z)) (S Z)` is narrowable to `add (S (add x' (S Z)) (S Z)`. Also, `add x (S Z)` is a narrex. Notice that if σ was $\{\texttt{x} \mapsto \texttt{Z}, \texttt{n} \mapsto \texttt{S Z}\}$ and $l \rightarrow r$ was add Z n $\rightarrow$ n then `add (add x (S Z)) (S Z)` is narrowable to `add (S Z) (S Z)`. It is the job of a *strategy* to pick a position p, a unifier σ, and a rule $l \rightarrow r$ such that $\sigma(t|_p) = \sigma(l)$. In this example, a strategy would decide which unification and rule to apply.

Residuation When narrowing is used with an appropriate strategy [13, 11] it is complete. That is, the result of a computation will be computed if a result exists. However, the search space for narrowing can be large. *Residuation* [3] is simpler than narrowing and can be more efficient, but residuation is not complete. When a computation of an expression e is unable to continue due to an uninstantiated logic variable, v, residuation suspends the execution of that expression and starts working on another part of the program. If v becomes instantiated then the computation can continue to work on e. Obviously, v may never be instantiated. In this case the computation cannot compute e. A simple example (in Curry) of residuation follows. (See section 2.1.3 for a description of Curry.)

```
digit = 0
digit = 1
digit = 2
```

```
⋮

digit = 9

addMult | x*x =:= x+x & x =:= digit = x where x free
```

For the `addMult` rule to be able to fire the condition `x*x =:= x+x & x =:= digit` must be satisfied. This means that both sides of the `&` must be satisfied. If a particular implementation started with the left hand side it would not be able to evaluate it due to the presence of the logic variable `x`. Residuation would halt the evaluation of `x*x =:= x+x` and start evaluating the right hand side, `x =:= digit`. This evaluation will instantiate `x`. When this happens the left hand side can continue.

2.1.3 Curry

There are many functional logic languages. For example, Curry [39], Escher [40], Le Fun [2], Life [1], Mercury [63], NUE-Prolog [43], Oz [61], and Toy [41] are just a few. While the virtual machine (§2.1.4) TeaBag was designed to work with supports many functional logic languages, currently Curry is the only language with a complier targeting this virtual machine. Thus, Curry is also the language that TeaBag currently debugs. In this section we will describe Curry. Curry has Haskell like syntax [55]. The basic form of a function in Curry is

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

where t_1 to t_n are data terms. That is, t_1 to t_n do not contain any operation (function) symbols. f is the name of the function. A term, t, rooted with f will be rewritten to $\sigma(e)$ if there exists a substitution σ such that $t = \sigma(f\ t_1\ ...\ t_n)$ and the condition c is satisfied. c is satisfied if it evaluates to either `true` or `success`. c is allowed to contain logic variables declared in vs that must be instantiated to satisfy the condition. Variables declared in vs are treated as logic variables. Also, e and c may refer to variables in $t_1\ ...\ t_n$.

Functions in Curry may have multiple values. So it is legal in Curry to define a function `digit` that returns 1 and returns 2 and returns 3, etc. In Haskell the first matching rule is applied whereas in Curry all unifiable rules are non-deterministically applied. This means that if $t|_p$ is unifiable with multiple left hand sides of a function in a Curry program then each of the corresponding right hand sides are non-deterministically applied.

Similar to Haskell, Curry has a `where` clause that introduces nested rules. The right hand side of a nested rule may refer to variables in the left hand side of a nesting rule. In the following example the function `appd` is nested inside of the function `append`. This means that `appd` can refer to the variables `x` and `y` which appear in the left hand side of `append`.

```
append x y = appd x
   where appd []     = y
         appd (z:zs) = z:appd zs
```

Curry also contains algebraic data types. Algebraic data types allow the user to easily define data structures. An algebraic data type is defined as follows.

$$\begin{array}{rl} \texttt{data}\ D = & C_1\ T_{11}\ ...\ T_{1n}\ | \\ & C_2\ T_{21}\ ...\ T_{2o}\ | \\ & \vdots \\ & C_m\ T_{m1}\ ...\ T_{mp} \end{array}$$

D is a user defined name for the data type. $C_1 ... C_m$ are the *constructor* names for each of the possible representations of the data. T_{q1} ... T_{qr} are the types associated with the constructor C_q. For example, the following algebraic data type defines a binary tree of integers.

```
data Tree = Leaf |
            Branch Int Tree Tree
```

In the above example a `Tree` is either a `Leaf` with or it is a `Branch` with an integer value and two subtrees. It is also possible to parameterize the data type with a *type variable.* So a binary tree of type `a` can be defined as

```
data Tree a = Leaf |
              Branch a (Tree a) (Tree a)
```

Now a binary tree of integers can be defined with `Tree Int`.

Functions in Curry are higher order. This means that functions can be used just like other pieces of data and functions can be partially applied. So the function `map` can be defined to take a function and a list and apply that function to each element of that list to get a new list. In Curry map could be defined as follows.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

The type of `map` is `(a -> b) -> [a] -> [b]`. This type contains two type variables, `a` and `b`. This type indicates that the first parameter to `map` is a function that takes an `a` and returns a `b`. The second argument of `map` is a list where the elements are of type `a`. Then the result of `map` is a list with elements of type `b`.

Now consider the list `[1,2,3]`. We could add one to each element of this list with `map (+1) [1,2,3]`. Notice here that the function `+` normally takes two arguments. In this context `+` is partially applied to the argument 1. Applying `+` to the argument 1 returns a function that takes another integer and adds one to it. So now this function is passed as an argument to `map`.

Curry has two operators that test for equality. The first operator is ==. The == operator tests if the term on the left hand side is the same as the term on the right hand side. The second operator is =:=. This operator checks if the term on the left hand side is the same as the term on the right hand for some instantiation of logic variables in either the left or the right hand sides of =:=.

Narrowing and higher order functions can be used to compute the inverse of a function in Curry. If `f :: a -> b` is a function then its inverse can be defined as follows.

```
f' :: b -> a
f' x | f y =:= x = y where y free
```

Curry supports both narrowing and residuation. When a computation cannot evaluate an expression due to an uninstantiated logic variable the computation can either narrow or residuate. This choice is determined by whether or not the function is *flexible* or *rigid*. Flexible functions narrow and rigid functions residuate. By default I/O actions and arithmetic operation residuate and all other functions are flexible. The user can change the default behavior for a function with the `eval` keyword in Curry. They can define a function `f` to be rigid by indicating `f eval rigid` in their source code.

A complete description of Curry can be found in [39].

2.1.4 FLVM

Most implementations of functional logic languages translate a program to Prolog. For example PAKCS [35], Toy [41], and UPV-Curry [9] all translate a functional logic language to Prolog. The implementation of Curry that TeaBag was designed to work with is different. This implementation, known as *FLVM* [14] (and section 3 of [80]), is a virtual machine for functional logic languages. FLVM stands for functional logic virtual machine. As its name implies, it is a virtual machine designed to work with many functional logic languages. Currently, Curry is the only languages that has a compiler for this virtual machine.

Programs that can be expressed as an *overlapping inductively sequential term rewriting system* [11] can be executed by the FLVM. Overlapping TRSs allow multiple rules with unifiable left hand sides. If there exists a term t, a position p, a rule $l \rightarrow r$, and a substitution σ such that $t|_p = \sigma(l)$ then there may exist another rule $l' \rightarrow r'$ and another substitution σ' such that $t|_p = \sigma'(l')$. An inductively sequential term rewriting system is one where each function can be represented with a definitional tree [10, 11]. A definitional tree is a hierarchical representation of a function. Definitional trees are a standard way of implementing computations that narrow.

Since the FLVM works with programs that can be represented as an overlapping inductively sequential TRS it can work with all functional logic languages which are defined by constructor-based TRSs. It is shown in [12] that all functional logic programs modeled by a constructor-based TRS can be translated to an overlapping inductively sequential TRS. This includes languages like Curry and $\mathcal{TOY}$. Thus the FLVM is a general functional logic virtual machine since it is designed to work with many functional logic languages.

The core data structure in the FLVM is a *term*. A term is a recursive structure. Each term has a root symbol and zero or more subterms. The subterms of a term can be shared. Thus, this structure is a graph. Terms can be operations, constructors, free variables, or primitives. Operations are functions that can be applied to arguments such as `++`. A constructor is a named piece of data such as `cons` or `nil`. Free variables are variables that can be narrowed on. Primatives are basic pieces of data like 1, 2, or 'a'. The FLVM has separate support for primatives for performance reasons. A term can be replaced (rewritten) with another term. A term can also have a substitution applied to it.

Computations manage reducing a term to normal form in the FLVM. Each computation is responsible for reducing a single term. At non-deterministic points in program execution new computations are created and the current computation is abandoned. Thus, each computation reduces a term until either that term is in normal form or until reducing that term causes non-determinism.

The computations are managed by the *space*. The space contains a pool of computations. It selects a computation from this pool, lets the computation make some progress, and then selects another computation to perform some work. The space ensures *operational completeness*. That is, the space does not let one computation "take over" and not allow any other computations have a chance at reducing their terms.

Functional logic languages can be compiled to a set of instructions that the FLVM can interpret. Currently, these instructions have a textual representation that the FLVM works with. Once the instruction set has stabilized the FLVM will interprete a byte-code representation of this instruction set. The instructions for the FLVM support the creation of terms, rewriting terms, sharing, pattern matching, narrowing, choice, and residuation operations.

2.2 Existing Debuggers

Functional logic languages borrow ideas from both functional and logic languages. Functional logic language debuggers also borrow ideas from functional and logic languages. Thus, to understand existing functional logic language debuggers it is helpful to understand logic language debuggers and functional language debuggers. We will start by presenting key logic language debuggers with respect to functional logic language debuggers. Then we will do the same for functional language debuggers. With this background, we will then talk about existing functional logic language debuggers.

2.2.1 Logic Language Debuggers

The three main kinds of logic language debuggers are tracing, box-oriented, and algorithmic. The initial debuggers for logic languages where tracers which showed the programmer the steps taken by the logic language implementation to reach an answer. They showed the programmer exactly what the logic language implementation was doing. However, this was difficult to follow since logic languages use backtracking to try other choices for predicates and clauses. Byrd says, "backtracking is described as being the remaking of the decision at the chronologically most recent choice point." [23]. He also says [23],

> In the first place, novices find it very difficult to understand what is happening when a program of any size starts backtracking, Even after considerable experience with Prolog, students will claim to be baffled in certain cases. Secondly, when practically debugging large programs a sudden backtrack to a choice point any distance away is highly confusing ("whence am I now?"). In neither case does the knowledge that it is the most recent choice that is being redone, provide us with any solution to our difficulties.

This difficulty in following a trace lead Byrd to develop the box-oriented debugger [23] for the logic language Prolog [70, 53].

The box-oriented debugging approach lets the user see the control flow of their program without being baffled by backtracking. Box-oriented debugging puts all of the clauses for a procedure into a box. It then puts four ports onto the box; call, exit, redo, and fail. These ports are the entry and exits points for the box. So now a trace is observing the movement between ports on boxes. When backtracking occurs it will not be presented as one large step back to some distant point in the past. Rather, it will be the small incremental steps used to arrive at that point. The following example will help explain box-oriented debugging. Performing box-oriented debugging on the Prolog program in figure 2.1 will produce the trace of ports in figure 2.2. In figure 2.2 the numbers on the far left represent the box number for the procedure. In this example the steps performed to arrive at

```
p :- s,t.
s :- q.
s :- r.
t.
r.
```

Figure 2.1: Simple Prolog Program to Demonstrate Box-Oriented Debugging

$$\left.\begin{array}{llll}(1) & Call & : & p\\ (2) & Call & : & s\\ (3) & Call & : & q\end{array}\right\} forward$$

$$\left.\begin{array}{llll}(3) & Fail & : & q\\ (2) & Redo & : & s\end{array}\right\} backtracing$$

$$\left.\begin{array}{llll}(4) & Call & : & r\\ (4) & Exit & : & r\\ (2) & Exit & : & s\\ (5) & Call & : & t\\ (5) & Exit & : & t\\ (1) & Exit & : & p\end{array}\right\} forward$$

Figure 2.2: Box Oriented Debugging Example

calling `r` from the backtracking are explicitly listed. Previously, in tracers these steps were omitted.

The next big development in debugging logic programs was by Ehud Shapiro [59]. Shapiro developed the algorithmic program debugging technique. 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. This is a very different approach to debugging from previous debuggers. This debugging technique does not show the user the details of how a result is obtained. Rather, the user answers questions about their program.

Naish et. al. [72] took algorithmic debugging and combined it with box-oriented debugging. They created a single debugging environment that allowed the user to use both of these debugging tools in combination.

2.2.2 Functional Language Debuggers

While there are different ways to categorize functional language debuggers, in the context of functional logic languages it is helpful to categorize them based on if the language they debug is eager or lazy. Since, most functional logic languages are lazy, the debuggers for lazy functional languages are more applicable. These are the debuggers considered here.

While Shapiro developed algorithmic debugging for Prolog, a logic language, he describes the general principles that a language must have for his debugging method to work [59]. Functional languages fit into this category. Much work has been done using algorithmic debuggers in functional languages. Nilsson and Fritzson point out that there are three problems with algorithmic debugging; the questions are big and complex, storing the trace is difficult unless the program is small, and too many questions are asked to the user [49]. They address these issues in their declarative debugger, Freja [49]. They use Strictification, a technique that replaces expressions with values where possible, to make the questions smaller and less complex [49]. Nilsson extended this debugger to make large traces practical by performing piecemeal tracing [47, 48]. Piecemeal tracing is on-demand partial tracing. Thus they only trace part of the program. If the user needs to see the trace from another part of the program, the program is rerun and that part of the trace is generated. To deal with the large number of questions, Nilsson incorporated the idea of trusted functions into his declarative debugger [48].

Sparud, with the help of Nilsson, took a different approach to reducing the number of questions the user gets asked in algorithmic debugging [65, 64, 69, 66, 50]. He created an evaluation dependence tree while the program is running. The user can then browse this tree and decide where to start the algorithmic debugging at. As he points out, strictly speaking this is not algorithmic debugging, rather it is declarative debugging. Declarative debuggers use the declarative semantics of the program to debug it. The algorithmic debugging technique presented by Shapiro is also declarative. Thus, declarative debugging is a larger class of debuggers. However, most declarative debuggers are also algorithmic. Browsing the evaluation dependence tree works well when the user has an idea of where the bug might be located. They can navigate the evaluation dependence tree to find the area where they think the bug is located. Then they can then start algorithmic debugging in that area.

Naish, Barbour, and Pope [46, 57] also created a declarative debugger for

Haskell. Their debugger was written almost entirely in Haskell itself. This makes the debugger more portable than when it is written in another language.

The next major type of functional language debugger is observational debugging, which was initially developed by Andy Gill [33] for Haskell [55]. Observational debugging works by letting the programmer add calls to the `observe` function in their code at places were they think it will give them insights to the cause of the bug. This allows the programmer to see what the expressions evaluate to. While the `observe` function superficially looks like `trace`, it does not behave like the `trace` statement. The trace statement is an identity function that prints its argument as a side effect. `observe` is also an identity function. However, `observe` writes to a file and not to standard out. Also, if the expression being observed is only partially evaluated then only the partial evaluation is written to the file. Thus, adding `observe` to a program does not change its evaluation. Expressions that were only partially evaluated prior to `observe` being added are still only partially evaluated. Gill created a viewer, called HOOD, to look at the contents of the file generated by `observe`. This viewer groups observations together based on a string identifier given to the call to `observe`. Thus, the observations are not listed in the lazy evaluation order. The lazy evaluation order of a program can be confusing [52, 47]. Grouping the observations together based on a tag frees the programmer from finding the observations in a list that is generated from lazy evaluation.

Claus Reinke took Gill's observational debugging idea and animated it [58]. HOOD allows the programmer to see *what* data is observed and *where* this data is located. It does not let you see *when* it is observed. Reinke developed GHOOD which allows the programmer to see *when* the data is observed.

The third type of functional debugger is tracing. Tracing shows steps of the computation of an expression to the user. The differences between tracers is in the type of step displayed, the way the step is displayed, and how the steps are recorded. Watson gives an overview of tracers in his Ph.D. dissertation [77]. The fundamental type of tracer traces reduction steps. Reduction step tracers show each step in reducing an expression to another expression. Many of these tracers show the β-reductions performed on an expression. Watson defines a trace where the reductions steps are based on a formal semantic model of lazy evaluation [77, 79]. He also created a browser for this trace that allowed the user to browse the trace and see highlighted expressions and source code [77, 78].

Penny created a trace called FIT similar to Watson's [54]. FIT addresses the problem of tracing a lazy language by presenting textual images of the return stack and heap.

Tolmach created a time traveling tracer [73] for SML [42]. He described how to view previous steps in a trace without storing all of the steps and without re-executing the entire program upto that step.

Another type of trace is a redex trail, which is a trace from a value or failure to the initial expression. So it is in the reverse order of reductions. Each expression is linked with its immediate redex. Sparud and Runciman initially developed this idea [68, 67]. They also developed an interactive trace browser for a redex trail trace. Since the trace may be large they let the user choose which parts of the trace to look at rather than showing the entire trace at once.

Chitil, Runciman, and Wallace noticed that Freja (declarative), Hat (redex trail), and Hood (observation) each had their own weaknesses and strengths [27]. Thus the ability to use all of these tools while debugging is beneficial. The Hat debugger was extended to include observational and declarative debugging [22, 76, 28]. When a program is run it generates a trace in a file. Hat then offers many viewers for that trace. It allows the user to view the trace declaratively as Freja would have shown the trace. Another viewer displays a redex trail for the trace. There is also a view for observations. Here the observational debugging is a little different from HOOD. In HOOD the programmer must add statements to their source code to generate observations. In Hat this is automatically done for the programmer. Hat also contains a viewer that lets the user see a virtual stack of function calls of when a program fails or is abruptly terminated. This stack is not the actual runtime stack, it is the stack of function calls that would occur if eager evaluation was used.

2.2.3 Functional Logic Language Debuggers

Functional logic language debuggers borrow ideas from both functional and logic language debuggers and extend them to debug functional logic languages. For a functional logic language debugger to be useful it must be able to deal with both the deterministic and non-deterministic features found in real programs [24]. So if a particular debugger was designed for debugging functional languages then it needs to be extended to handle the logic side of the language and vice-versa.

Declarative debugging is a debugging technique that has been shown to work in both functional and logic languages. Thus, it is only natural that declarative debugging would work in functional logic languages. This does not mean that creating a declarative debugger for a functional logic language is trivial. Most of the work for debugging functional logic languages has been focused on declarative debuggers. The techniques for using a declarative debugger in functional and logic languages must be integrated together to deal with the demanding area of functional logic languages. For example, the debugger should be able to deal with encapsulated search [24]. Naish and Barbour created the first functional logic declarative debugger for the functional logic language NUE-Prolog [45]. Declarative debuggers have been created for Toy [25, 26] and Curry [24]. Alpuente et al. have also created a declarative debugger, Buggy, which is a general framework for declarative debugging of functional logic programs [7, 6, 4, 5].

Hanus and Josephs [36] and Arenas-Sánchez and Gil-Luezas [16, 17] added new boxes and ports to Byrd's box oriented debugger to debug functional logic programs. Byrd's model had to be extended in the realm of functional logic languages to handle the functional side of the language.

Hanus and Koj created an integrated development environment, called CIDER, for Curry [37, 38]. CIDER contains a program editor, tools for analyzing Curry programs, a graphical debugger, and tools for drawing dependency graphs. The focus of CIDER is on program development of which debugging is just one aspect. The debugger in CIDER traces rewrite steps taken by a Curry program. This tracer is analogous to the a tracer of β-reductions in functional languages. They both trace the reduction steps. To handle non-deterministic features CIDER displays backtracking steps as the next step in the trace (see section 3.2 for a more detailed discussion).

Recently, Braßel, et al. extended Gill's idea of observational debugging to functional logic languages by handling non-deterministic search, logical variables, concurrency, and constraints [21]. Alternate non-deterministic choices in COOSy are shown in a group and the bindings of logic variables are displayed.

2.2.4 Summary

At first glance it may seem 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 [38] contains a tracer of 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 [23]. The tracer in CIDER is not as effective on non-deterministic programs as it is on deterministic programs.

Chapter 3

Non-Deterministic Tracing

The main research contributions of TeaBag are related to non-deterministic trace generation and presentation. This chapter will start by defining a *trace* and a *trace view*. Then it will review the existing traces and trace views for functional logic languages. Finally, this chapter will present the trace and trace view used in TeaBag.

3.1 What Is a Trace

Most people have an intuitive idea that a trace is thought of as a sequence of steps performed while executing a program. However, this definition does not adequately describe all traces. To be able to accurately talk about a trace a better definition is needed. We will start by giving some of the trace definitions that we have come across. By no means is this an exhaustive list of tracing definitions. Rather, this is just the definitions of a trace that we have found in the literature related to tracing lazy functional programs, tracing logic programs, and tracing functional logic programs. We will then point out where each of these definitions come short of adequately describing a trace and what parts of the definition are good. Finally, we will give a definition of a trace that adequately describes a trace by building off of the good points from the existing definitions and avoiding the pitfalls the existing definitions fell into.

Watson defines a trace as [77]:

> A trace of a computation is a *representation* of the *history* of the *computational steps* which are carried out to complete the computation.

Here a number of the terms used in the definition need an explanation. Firstly, what is a *computation step*? A *computation step* is operationally linked to an evaluation model for a given language. An evaluation model can either be a physical or artifical entity. That is, the evaluation model does not have to physically exist. For example the evaluation model may be the actual reduction steps taken by an interpreter of the language. On the other hand the evaluation model could be artificial in that it is the steps of the operational semantics of the language. A *computation step* is the smallest sub-part of a computation for a given model of evaluation that is recorded. So if the evaluation model is the β-reductions of a lambda calculus interpreter then a computation step could be an individual β-reduction.

The definition of *history* is very general so that the definition of a trace can apply to many types of tracers. As Watson says [77], "The key concept here is that the *history* is a collection of fundamental steps whose structure shows the *relationship* between the steps." Trace steps are *sequential* if the relationship between steps comes from the order of evaluation. Typically, sequential steps have a temporal ordering. In the lambda calculus evaluation model example, the history would be a sequence of β-reduction steps. While we usually think of the history as a sequence of steps it does not have to be. For example, a trace of the declarative semantics of a program does not necessarily rely on the sequence of steps. Rather it just shows the relationships between sub-expressions [77]. Thus, the history for a declarative semantics trace would not be sequential.

Finally, the *representation* is how the trace is displayed to the user. It is possible to have a history of computation steps that is viewed multiple ways. The debugger, Hat [22, 27, 71, 76], is a good example of this. Hat generates traces for Haskell programs. There are many Hat-viewers that display to the user different views of the same *history* of *computation steps*. Thus, even though each Hat-viewer works with the same trace, they each have their own definition of a trace since they each have different *representations*.

While Watson's definition of a trace comes close to adequately defining a trace it comes short in three regards. Firstly, this definition is not defined for non-terminating computations. That is, Watson's definition says, "... which are carried out to *complete* the computation." Thus, this definition only applies if the computation is able to complete. Obviously, this is not desirable since one would like to be able to use a trace to locate non-termination bugs. Many tracers deal with non-termination by allowing the user to "kill" the computation via a mechanism such as control-c. The tracer Hat is a good example of this. In this case the computation completed when it was killed. Thus, Watson's definition would apply. However, we do not want to limit the definition of tracing to be able to only trace non-terminating programs when there is a way to "kill" them. The tracer in TeaBag is a good example of this. TeaBag gives the user two ways to look at the trace of non-terminating programs. Firstly, the user can kill the program, much like they would with Hat, and then view the trace. However, TeaBag also allows the user to set a breakpoint and look at the trace when the breakpoint is hit at runtime. Thus the computation is not yet completed and it has not been killed, but the user is examining the trace by "pausing" the trace in the middle of generating it. The user can then choose to resume trace generation after they have looked at the partially generated trace.

The second problem with Watson's definition is that the definitions implicitly implies that the trace contains *all* of the steps that are "carried out to complete the computation." While most tracers do gather *all* of the steps performed to complete the computation, some tracers do not. For example, HOOD [33] and COOSy [21] are observational tracers. Both of them only record trace steps (a.k.a. observations) for the data structures and functions that the user has instructed them to observe. Thus, they do not record *all* observations for a computation. Another example is trusted functions. Many tracers do not record any steps performed while computing a trusted function. Thus, these tracers do not record all steps performed while executing a program. TeaBag also does not record all of the trace steps. TeaBag only records narrowing steps performed on a given subterm. Thus, TeaBag does not record *all* of the narrowing steps performed while computing the top level term.

The final problem with Watson's definition of a trace is that he mixes the concept of a trace with its presentation. At first glance this does not seem like a problem. It even seem likes it is necessary, since what good is a trace if it is not presented to the user? To see why it is not a good idea to mix these two concepts consider the following examples. The tracer Hat records the trace to the disk while the program runs. This process of generating the trace does not deal with presenting the trace to the user in any way. It does not even require that the user look at the trace. The user could just generate the trace and then never look at it. When this trace is generated Hat does not know how it will be presented to the user. Once the trace has been generated the user can view the trace with one of Hat's many trace viewers. Each of these trace viewers gives the user a *different* presentation for the *same* trace. Thus, Hat has multiple representations for one collection of trace steps. Also, it is possible for a third-party to define a new viewer for Hat that works with its existing trace. This viewer for the trace would be defined *after* the trace for Hat has been defined. There is yet another, and more important, reason why it is not desirable to have the definition of the trace mixed with the trace presentation. While no such tools that we are aware of exist at this time, it is not unreasonable to image a tool that could analyze a trace to automatically locate certain types of bugs or give the user some type of information about the trace. For example, this tool could perform a termination analysis on the trace steps to try and automatically locate the source of non-termination bugs. In this situation a trace would be generated and then the tool would analyze the trace. However, the trace is never viewed. Yet the trace is still a very useful entity. Thus, the presentation of the trace needs to be separated from the definition of the trace itself.

While Watson's trace does have some problems it also has some good points. Firstly, his *abstract* notion of a computation step is good. He does not limit the steps in the trace to merely be physical steps taken by a language implementation. Thus, his notion of the computation steps apply to redex trail traces and declarative traces. Secondly, he makes no requirements about the relationship among the steps. Specifically, he does not limit them to sequential steps.

Penney says [54],

> Traditionally, tracing the execution of a program means *displaying* an *outline* of the *sequence* of *evaluation steps* taking an initial program state to the final result.

Once again this definition has the same pitfalls as Watson's definition. First of all, it requires that the program terminates to get a *final* result. Secondly, it encompasses *all* evaluation steps taken to get a final result. Thirdly, it mixes the concept of the trace with the *display* of the trace. This definition has one extra problem that Watson's definition does not have. It requires that the evaluation steps be sequential. Thus, this definition would not apply to declarative traces. Penney even admits this is a problem with the definition. However, he did not give any other definition for a trace.

Ducassé also gives a traditional definition for a trace [30]. She says,

> "Traditional" tracers ... usually present *histories* of *execution events* where each event represents a *low-level step* in the execution process.

This definition also suffers from mixing the presentation with the trace definition. But this definition does not require that the execution process terminates or that all of the steps be in the history. However, this definition requires that the steps be "low-level". Not all tracers show "low-level" steps like β-reductions or rewrite steps. Certainly declarative tracers would not be considered "low-level".

Ducassé also defines a trace as [31],

> Tracers *provide* programmers with *detailed information* about *steps* of program execution.

This definition is the closest definition to adequately defining a trace that we were able to find. It does not require that the program execution terminates. It also says nothing about which steps of the program execution must be included. Thus, it does not require that all steps be provided to the programmer. Also, this definition does not include the trace presentation. It only says that the information about the steps must be provided. So the information can be provided in a file that is never viewed. However, the one problem with this definition is that a trace provides *detailed information* about steps. We certainly do not want to limit tracers to only providing detailed information. It is quite possible to define a trace that gives an overview. For example, a trace of a functional logic language may just trace an overview of the search space. Typically, one would want detailed information since a non-detailed view can be constructed from the detailed information. However, we do not want to limit our definition of tracers to only working with detailed information.

Given these positives and negatives of tracing definitions we will now attempt to define a trace in such a way that adequately describes all possible traces.

> A trace is a *collection* of *some* of the *steps* performed while *evaluating* a program where the structure of the collection represents the *relationships* between the steps.

Firstly, this definition does not impose any particular *structure* on the collection of steps. All it says is that there must be some kind of structure. That is, the steps must be related in some fashion. If the steps are not related in any way then they do not make up a trace. Rather the steps would be discrete pieces of information about the evaluation that are not connected in any way to the other discrete pieces of information. A basic requirement of a trace is that the steps are connected in some fashion.

Secondly, this definition only requires that *some* of the steps be in the trace. It does not require *all* of the steps to be in the trace. Obviously, this also includes traces that happen to have all of the steps.

Also, this definition does not require that the program terminates. All it requires is that the program was evaluated even if it was only partially evaluated. Thus, this definition works with tracers that can be paused in the middle of trace generation.

The only requirement on the steps that this definition imposes is that the steps be *performed* while evaluating the program. It says nothing about what the steps look like. Thus the steps could be a physical entity like a lambda expression or term. At the same time the steps could be abstract. However, this definition does not completely give free reign to the steps. It requires that the steps be something that is performed while evaluating the program. That is, a trace of a program cannot contain steps performed while evaluating some other program or random steps. While this almost seems too obvious to state, it is worth putting into the definition so that a trace is more precisely defined.

Finally, this definition says nothing about presenting the trace to the user. As already mentioned, it is beneficial to keep the definition of the trace distinct from viewing the trace.

This brings me to my next definition. Just a trace by itself is not too useful. Data needs to be extracted from the trace. Typically, this is done via a trace viewer.

A trace view is a *visual representation* of a trace.

Thus, a *trace view* provides a way of presenting the trace to the user. This may be a textual representation or it may also be a graphical representation. It is also possible for the trace view to only show some of the steps at once and then let the user navigate through those steps. Notice, that this definition does not limit the visual representation to showing the individual steps of the trace. Typically, this is what one would expect. Though it is possible to image a trace view that shows an overview of the trace without showing any of the steps of the trace. The only requirement is that the visual representation be a representation of the trace. That is, there must be some link between the representation and the trace. Thus, the representation cannot show the steps for some other trace. Also, there can be *multiple* trace views for *one* trace. Thus, one trace may have many ways of being displayed to the user.

3.2 Existing Functional Logic Tracers

Tracers for functional logic languages differ from tracers for functional languages in that they must deal with non-determinism. Tracers for functional logic languages differ from tracers for logic languages in that they must deal with function computation. So tracing functional logic languages is not merely a matter of directly using a functional or logic tracer. Rather, the tracer must be extended to encompass all aspects of the functional logic language. This section focus on how functional logic tracers deal with tracing the non-deterministic aspects of functional logic programs. Three types of tracers have been developed for functional logic languages; expression evaluation, box oriented and observational.

The first type of functional logic tracer is an expression evaluation tracer. CIDER [38, 37] is an example of an expression evaluation tracer. Expression evaluation tracers show the trace of the computation as a linear sequence of narrowing steps. Non-deterministic steps are shown as backtracking steps. Backtracking works by following one path at a non-deterministic step. If that path does not lead to any solutions then the evaluation "backtracks" and tries another path. If none of the paths led to a solution then the evaluation fails.

However, as noted by Byrd, backtracking can be difficult to follow [23]. He says,

> In the first place, novices find it very difficult to understand what is happening when a program of any size starts backtracking, Even after considerable experience with Prolog, students will claim to be baffled in certain cases. Secondly, when practically debugging large programs a sudden backtrack to a choice point any distance away is highly confusing ("whence am I now?"). In neither case does the knowledge that it is the most recent choice that is being redone, provide us with any solution to our difficulties.

The reason backtracking is difficult to follow is because the evaluation appears to jump around. The next step in the trace may actually be a backtracking step that is selecting another path to try. Consider the following example:

```
-- natural numbers defined by s-terms (Z=zero, S=successor):
data Nat = Z | S Nat

-- less-or-equal predicated on natural numbers:
leq :: Nat -> Nat -> Bool
leq Z     _     = True
leq (S _) Z     = False
leq (S x) (S y) = leq x y

goal | leq x (S(S Z)) =:= True = x where x free
```

The trace of narrowing steps for `goal` using backtracking is shown in figure 3.1. In this trace logic variables are displayed as x_i. Also, note that in this representation the conditional guard for `goal` has been changed to an `if then else`. This is one possible way that a compiler may compile conditional guards. If the guard is not satisfied then the computation of `goal` fails. Thus, the else branch of this term is `Fail`.

Step 1 in figure 3.1 is a rewrite step that rewrites `goal` to `if leq(`x_1`,S(S(Z))) =:= True then` x_1 `else Fail`. Step 2 instantiates x_1 to `Z`. Steps 3 through 5 are rewrite steps. Then step 6 instantiates x_1 from step 1 to `(S` x_1`)`. The rest of the steps continue in a similar fashion.

Now lets say we wanted to find out how the result `S(S Z)` is obtained from this trace. This is difficult to do since there are two backtracking steps involved in reaching this answer. Specifically, steps 6 and 12 are backtracking steps. These steps are difficult to follow since they really come from steps farther back in the trace. Step 6 is obtained by substituting `(S` x_2`)` for x_1 in step 1. Step 12 is

$$
\begin{array}{lll}
\texttt{goal} & \rightarrow & \texttt{if leq } x_1 \texttt{ (S(S Z))} =:= \texttt{True then } x_1 \texttt{ else Fail} \quad \text{(Step 1)} \\
 & \rightarrow & \texttt{if leq Z (S(S Z))} =:= \texttt{True then Z else Fail} \quad \text{(Step 2)} \\
 & \rightarrow & \texttt{if True} =:= \texttt{True then Z else Fail} \quad \text{(Step 3)} \\
 & \rightarrow & \texttt{if True then Z else Fail} \quad \text{(Step 4)} \\
 & \rightarrow & \texttt{Z} \quad \text{(Step 5)} \\
 & \rightarrow & \texttt{if leq (S } x_2\texttt{) (S(S Z))} =:= \texttt{True then (S } x_2\texttt{) else Fail} \quad \text{(Step 6)} \\
 & \rightarrow & \texttt{if leq } x_2 \texttt{ (S Z)} =:= \texttt{True then (S } x_2\texttt{) else Fail} \quad \text{(Step 7)} \\
 & \rightarrow & \texttt{if leq Z (S Z)} =:= \texttt{True then (S Z) else Fail} \quad \text{(Step 8)} \\
 & \rightarrow & \texttt{if True} =:= \texttt{True then (S Z) else Fail} \quad \text{(Step 9)} \\
 & \rightarrow & \texttt{if True then (S Z) else Fail} \quad \text{(Step 10)} \\
 & \rightarrow & \texttt{S Z} \quad \text{(Step 11)} \\
 & \rightarrow & \texttt{if leq (S } x_3\texttt{) (S Z)} =:= \texttt{True then (S(S } x_3\texttt{)) else Fail} \quad \text{(Step 12)} \\
 & \rightarrow & \texttt{if leq } x_3 \texttt{ Z} =:= \texttt{True then (S(S } x_3\texttt{)) else Fail} \quad \text{(Step 13)} \\
 & \rightarrow & \texttt{if leq Z Z} =:= \texttt{True then (S(S Z)) else Fail} \quad \text{(Step 14)} \\
 & \rightarrow & \texttt{if True} =:= \texttt{True then (S(S Z)) else Fail} \quad \text{(Step 15)} \\
 & \rightarrow & \texttt{if True then (S(S Z)) else Fail} \quad \text{(Step 16)} \\
 & \rightarrow & \texttt{S(S Z)} \quad \text{(Step 17)} \\
 & \rightarrow & \texttt{if leq (S } x_4\texttt{) Z} =:= \texttt{True then (S(S(S } x_4\texttt{))) else Fail} \quad \text{(Step 18)} \\
 & \rightarrow & \texttt{if False} =:= \texttt{True then (S(S(S } x_4\texttt{))) else Fail} \quad \text{(Step 19)} \\
 & \rightarrow & \texttt{if False then (S(S(S } x_4\texttt{))) else Fail} \quad \text{(Step 20)} \\
 & \rightarrow & \texttt{Fail} \quad \text{(Step 21)}
\end{array}
$$

Figure 3.1: Trace of `goal` with backtracking

obtained by substituting `(S` x_3`)` for x_2 in step 7. So to understand where the backtracking steps came from we must look back in the trace. In general this can be a unbounded number of steps back in the trace. Not only must one look back in the trace to find out where the backtracking step came from, but one must also keep track of what substitutions have been applied at that step. In this example this is fairly easy since there are only two possibilities for each non-deterministic step; `Z` and `(S` x_i`)`. This means that the second choice is always the substitution for the backtracking step. One can easily imagine a situation where there are multiple non-deterministic steps each using a different substitution that must be tracked by the user. Thus, using a trace with backtracking is difficult to do.

Viewing a trace as a linear sequence of narrowing steps is difficult when backtracking is involved. When there are no backtracking steps in the trace then the trace is much more intuitive and easier to follow. Evaluations that do not involve backtracking steps are ones with no non-deterministic steps. These evaluations are purely functional in the sense that none of the logic features of a functional logic language are involved.

Byrd developed box oriented debugging for Prolog to make reading a trace with backtracking easier. Box oriented debugging shows the individual steps taken while backtracking. Thus, the trace does not show backtracking steps as big jumps to some previous point in the trace to try another alternative. Rather, the trace shows the individual steps taken to determine the next backtracking step. So Byrd did not remove backtracking from the trace. Rather, he made backtracking easier to understand.

Box oriented debugging was extended to functional logic languages by adding new boxes and ports [36, 16]. Fundamentally, the trace of non-deterministic steps in functional logic box oriented debuggers is no different from the trace of non-deterministic steps in the Prolog box oriented debugger. They both still trace non-deterministic steps with backtracking.

The final type of tracer for functional logic languages is observational. Braßel et. al. created an observational debugger called COOSy [21]. 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 [33] with non-deterministic search, logical variables, concurrency, and constraints. Alternate non-deterministic choices in COOSy are shown in a group and the bindings of logic variables are displayed. COOSy deals with non-determinism by grouping non-deterministic alternatives together. So if a user is observing a particular non-deterministic function then each time that function is called all of the alternative results for that function will be grouped together. Since COOSy does not trace evaluation steps it does not have to deal with backtracking.

3.3 Non-Deterministic Tracing in TeaBag

We wanted to create an expression evaluation tracer for TeaBag. However, we also did not want the trace to suffer from backtracking. One way to mitigate this problem is to use box oriented debugging. However, we did not want the user to be aware of backtracking at all. There were two reasons for this. The first one is that any amount of backtracking can be confusing. The second one is because TeaBag was designed to work with a virtual machine that does not have backtracking. When looking a trace that involves backtracking we noticed that the trace is much easier to understand if the steps for obtaining one of the results are extracted out of the trace. For example, if we extract the steps from the trace in figure 3.1 performed to get the result `S(S Z)` we get the trace in figure 3.2.

In figure 3.2 it is much easier to see how `S(S Z)` is obtained than in figure 3.1. However, in figure 3.2 there is no way to find out how `S Z` or `Z` was obtained. To deal with this we decided to break up one big non-deterministic trace of a computation into many smaller deterministic traces. We choose to have one trace

$$\begin{array}{lll}
\texttt{goal} & \rightarrow\ \texttt{if leq } x_1 \texttt{ (S(S Z)) =:= True then } x_1 \texttt{ else Fail} & \text{(Step 1)} \\
 & \rightarrow\ \texttt{if leq (S } x_2\texttt{) (S(S Z)) =:= True then (S } x_2\texttt{) else Fail} & \text{(Step 6)} \\
 & \rightarrow\ \texttt{if leq } x_2 \texttt{ (S Z) =:= True then (S } x_2\texttt{) else Fail} & \text{(Step 7)} \\
 & \rightarrow\ \texttt{if leq (S } x_3\texttt{) (S Z) =:= True then (S(S } x_3\texttt{)) else Fail} & \text{(Step 12)} \\
 & \rightarrow\ \texttt{if leq } x_3 \texttt{ Z =:= True then (S(S } x_3\texttt{)) else Fail} & \text{(Step 13)} \\
 & \rightarrow\ \texttt{if leq Z Z =:= True then (S(S Z)) else Fail} & \text{(Step 14)} \\
 & \rightarrow\ \texttt{if True =:= True then (S(S Z)) else Fail} & \text{(Step 15)} \\
 & \rightarrow\ \texttt{if True then (S(S Z)) else Fail} & \text{(Step 16)} \\
 & \rightarrow\ \texttt{(S(S Z))} & \text{(Step 17)}
\end{array}$$

Figure 3.2: Steps taken to get `S(S Z)` in trace of `goal`.

for each path through the search space. In this example, there are four paths in the search space. There is three paths for obtaining the results `Z`, `S Z`, and `S(S Z)`. There is also one path for obtaining the final `Fail`. Thus, TeaBag contains a trace for each of these paths. Notice here, that the trace along each of the paths is completely deterministic. None of the traces contain a backtracking step.

Each trace of a path in the search space contains a number of shared steps with other paths. Two paths share steps up to the first non-deterministic steps where they differ. Thus, to make tracing more efficient we choose to use one tree structure to contain all of the information for these traces. This allows the traces to share common steps. So now a trace is obtained from all of the steps along one path in this tree. The tree for this collection of traces in TeaBag has the same structure as the search space. That is, the tree fans out at the same places where the search space fans out.

So tracing a computation in TeaBag really generates many traces. There is one trace for each path in the search space. These traces are efficiently collected by using a tree structure which allows the traces to share common steps.

3.4 Non-Deterministic Trace Viewer in TeaBag

TeaBag not only generates traces, it also displays the traces to the user. We had two options for the trace presentation in TeaBag. The first option was to display the tree of the traces to the user. The second option was to break the trace presentation up into two parts where the first part shows the user an overview of the search space and the second part shows the user the trace for one of the paths in the search space.

One way the trace could have been presented was to display the tree used to collect the traces as shown in figure 3.3. In figure 3.3 determining how `S(S Z)` is obtained is easier than in figure 3.1 where backtracking is presented. The steps for obtaining this result are the ones in the tree that are on the path from `goal` to `S(S Z)`. In the trace in figure 3.3 this path is in bold. However, we choose to not use this representation. The primary reason for this is because this trace could get big very fast. Trying to read all of the steps displayed in the tree would be difficult and overwhelming. Also, we felt that just presenting the steps along one path in the search space is still a better way to read a trace. This allows the user to focus on the steps taken to reach on particular result.

So we decided to split the information presented in figure 3.3 into two parts. The first part contains an overview of the search space. The second part contains the steps for a given path in the search space. The user can select paths in the search space. This lets them view the trace steps along that path. By selecting

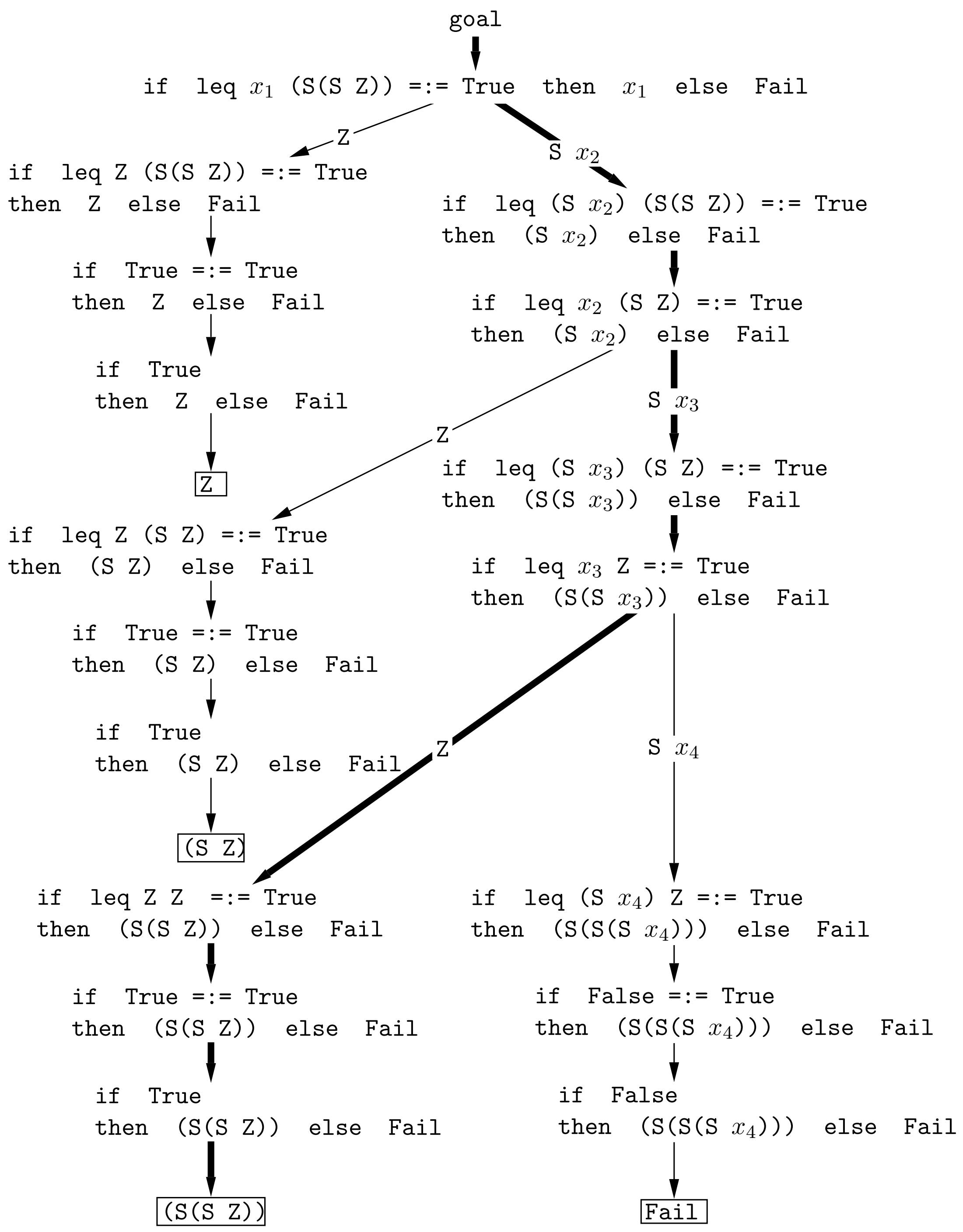

goal
if leq x_1 (S(S Z)) =:= True then x_1 else Fail
Z
S x_2
if leq Z (S(S Z)) =:= True then Z else Fail
if True =:= True then Z else Fail
if True then Z else Fail
Z
if leq (S x_2) (S(S Z)) =:= True then (S x_2) else Fail
if leq x_2 (S Z) =:= True then (S x_2) else Fail
Z
S x_3
if leq Z (S Z) =:= True then (S Z) else Fail
if True =:= True then (S Z) else Fail
if True then (S Z) else Fail
(S Z)
if leq (S x_3) (S Z) =:= True then (S(S x_3)) else Fail
if leq x_3 Z =:= True then (S(S x_3)) else Fail
Z
S x_4
if leq Z Z =:= True then (S(S Z)) else Fail
if True =:= True then (S(S Z)) else Fail
if True then (S(S Z)) else Fail
(S(S Z))
if leq (S x_4) Z =:= True then (S(S(S x_4))) else Fail
if False =:= True then (S(S(S x_4))) else Fail
if False then (S(S(S x_4))) else Fail
Fail

Figure 3.3: Trace of goal using a tree.

different paths in the search space the user can see the narrowing steps taken to obtain the final expression. Examining the steps of a trace first involves picking a path in the tree to follow and then looking at the steps on this path.

We believe that having the representation split into two parts is better than showing it all together as one tree as in figure 3.3. There are two reasons for this. The first one is because this limits the amount of information the user must absorb. When the trace steps are integrated in to the search space the user must be able to determine what path in the search space they want to look at while looking at the individual narrowing steps. By having the presentation split into two views the user can concentrate on the information that is important for selecting a path in the search space. Then, once they have selected this path, they can concentrate on the narrowing steps along this path. Secondly, separating the presentation into two parts allows the trace viewer to give more details for each of the narrowing steps. Since, fewer steps need to be shown at once when the presentation is broken up into two parts the trace viewer can show more detail for each step.

We described a new trace to adequately handle tracing expression evaluations in functional logic languages. This trace is different from previous functional logic tracers in that it generates a separate trace for each path in the search space. TeaBag presents this trace to the user in two parts. The first part shows an overview of the search space. The second part shows the trace steps along one path in the search space.

Chapter 4

TeaBag Features

TeaBag is a debugger for Curry. Thus it includes many debugging features found in other debuggers for functional, logic, and functional-logic languages. TeaBag also has features not found in other debuggers. The debugging model of TeaBag is a runtime debugger with optional tracing. To understand some of the features of TeaBag, 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. TeaBag is different from most debuggers in that includes both a runtime debugger and a tracer. The runtime debugger and tracer interact to provide a unique style of debugging.

For runtime debugging TeaBag includes breakpoints, rewrite step viewing, step control, context hiding, and choice control features. The tracer in TeaBag shows the computation structure and provides a trace browser. Both the runtime debugging environment and the tracer provide highlighting and eager evaluation.

4.1 Runtime Features

4.1.1 Breakpoints

TeaBag has the ability to set breakpoints. Breakpoints are a debugging feature found in most debuggers for imperative languages. It has been observed that breakpoints and debugging features found in debuggers for most imperative languages do not work well in functional languages [52] thus they do not work well in functional logic languages. When breakpoints are mentioned one may think of setting a breakpoint on a line of code. However, in general, breakpoints are not associated with lines of code. Breakpoints are runtime events that halt program execution and allow a programmer to look at a "snapshot" of their program. Thus when breakpoints are associated with lines of code the event is the line of code being executed. TeaBag supports three different kinds of breakpoints. Each of them are based on different types of runtime events. They are functional breakpoints, term breakpoints, and non-deterministic step breakpoints. These breakpoints are better suited to functional-logic programs than breakpoints on specific lines of code.

Like CIDER [38] TeaBag supports functional breakpoints. The programmer can set a breakpoint on any function defined in their code. The halting event for this breakpoint is rewriting a term rooted with the function that has a breakpoint set on it. During runtime when any term that is rooted with a function that has a breakpoint is rewritten the virtual machine halts. When this happens TeaBag displays the rewrite step to the user in the runtime term viewer (§4.1.2).

Another kind of breakpoint that TeaBag supports is term breakpoints. A term breakpoint is a breakpoint that applies only to a single term. When that term is rewritten the virtual machine will halt and TeaBag will display that rewrite step to the user. This is different from function breakpoints. A function breakpoint on a function f will cause TeaBag to display the rewrite steps for *all* terms rooted with f that are rewritten. On the other hand a term breakpoint on term t will only display the rewrite step when t is rewritten. Thus term breakpoints are more specific in that they apply to only one particular term where as function breakpoints can apply to many terms. Term breakpoints are basically the same as the sub-expression breakpoints supported by FIT [54].

Term breakpoints give the user more flexibility in the kinds of breakpoints they can set. For example a user may want to see how a function, f, behaves in certain situations. For example, say their code contains the following function.

```
g x = let y = SomeComplexComputationInvolvingX in f y
```

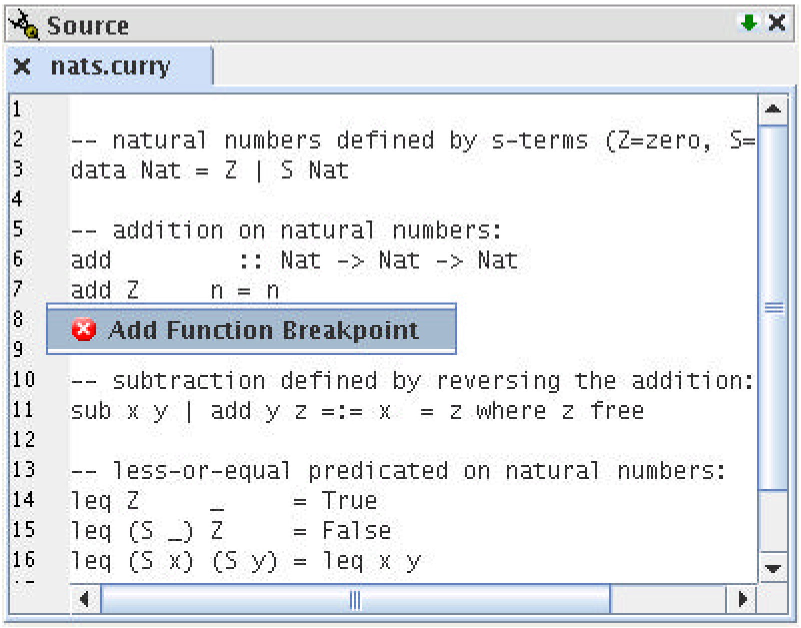
Now lets say that `g` is called infrequently and `f` is called a lot. If we want to see how `f` behaves when it is passed `y` as an argument we may have to step past a lot of other calls to `f` that we do not care about. Instead we can set a breakpoint on `g`. Then when TeaBag display the rewrite step for `g` we can set a breakpoint on the term rooted with `f`. Then when that term is rewritten TeaBag will display the step.

Term breakpoints are also useful when "hunting" for the source of a bug. Typically, a user will have an idea of where to start searching for a bug, e.g. the last function they wrote. They can set a breakpoint on that function. Then when a term rooted with that function is rewritten they will see the rewrite step. When they are looking at the rewrite step they can choose to set breakpoints on terms displayed in the rewrite step viewer. Many times they may decide that they want to investigate how certain terms are rewritten only once have they have see them in another term displayed in a rewrite step. They can continue to do this to hunt down the bug.

Function and term breakpoints are typically used together. Initially, a breakpoint is set on a function. Then when a term rooted with that function is rewritten the rewrite step is displayed to the user. The user can then set term breakpoints on specific subterms, typically parameters. When those subterms are rewritten the rewrite step is displayed to the user.

TeaBag also supports non-deterministic step breakpoints. Non-deterministic step breakpoints will halt the virtual machine on each non-deterministic step and display that step to the user. There are three kinds of non-deterministic steps breakpoints. The first one is for when the non-determinism comes from multiple instantiations of a logic variable in a narrowing step. The second one is for when the non-determinism comes from multiple choices for a function. The final kind of non-deterministic step breakpoint is for both of the first two kinds non-deterministic step.

TeaBag allows the user to interact with the breakpoints through the GUI. To set a function breakpoint the user “right-clicks” next to the function that they wish to set a breakpoint on. For example, figure 4.1 shows a user setting a breakpoint on the function `add`. All functions that have breakpoints set on them have a red dot next to the function. Breakpoints on functions can be removed by "right-clicking" next to the function and selecting to remove the breakpoint. Breakpoints can be set on terms by "right-clicking" on them and selecting to add a breakpoint as shown in figure 4.2.



```

-- natural numbers defined by s-terms (Z=zero, S=
data Nat = Z | S Nat

-- addition on natural numbers:
add          :: Nat -> Nat -> Nat
add Z     n = n



-- subtraction defined by reversing the addition:
sub x y | add y z =:= x  = z where z free

-- less-or-equal predicated on natural numbers:
leq Z     _     = True
leq (S _) Z     = False
leq (S x) (S y) = leq x y
```

Figure 4.1: Setting a breakpoint on a function

To manage the breakpoints TeaBag provides a breakpoint manager panel (figure 4.3). This panel shows all function and term breakpoints. The user can select to remove individual breakpoints, all breakpoints, all term breakpoints, or all function breakpoints.

4.1.2 Runtime Term Viewer

TeaBag displays runtime narrowing steps in the runtime term viewer. Deterministic steps are displayed in the runtime term viewer with the redex on the left

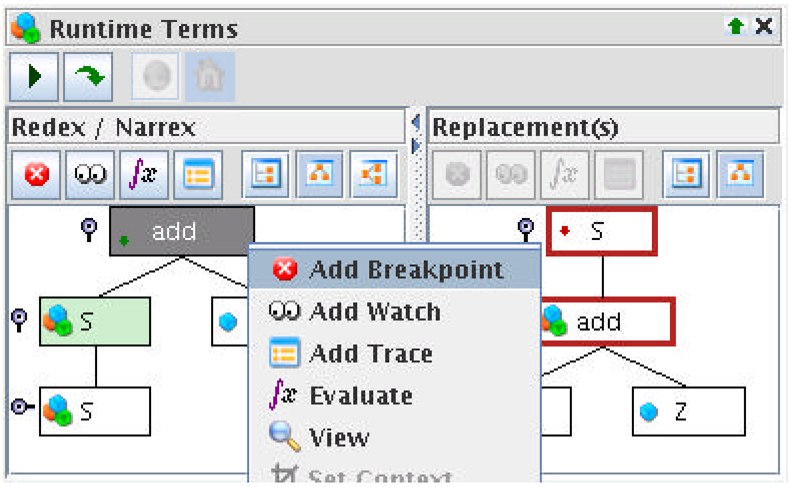


Figure 4.2: Setting a breakpoint on a term

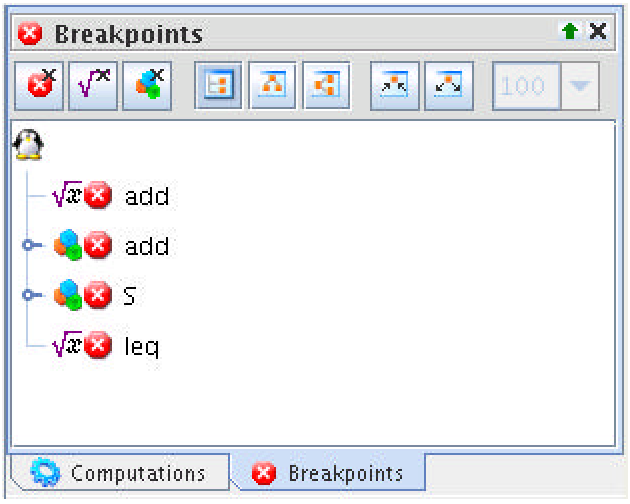


Figure 4.3: Breakpoint Manager

and the replacement on the right. For example, figure 4.4 shows the runtime term viewer for the rewrite step `add (S (S Z)) Z` $\rightarrow$ `S (add (S Z) Z)`.

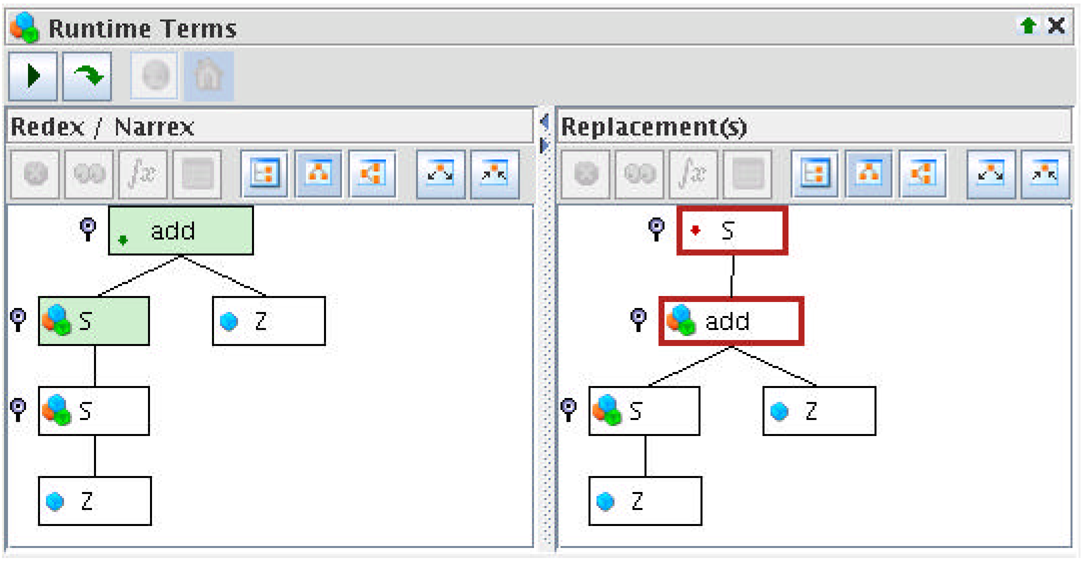
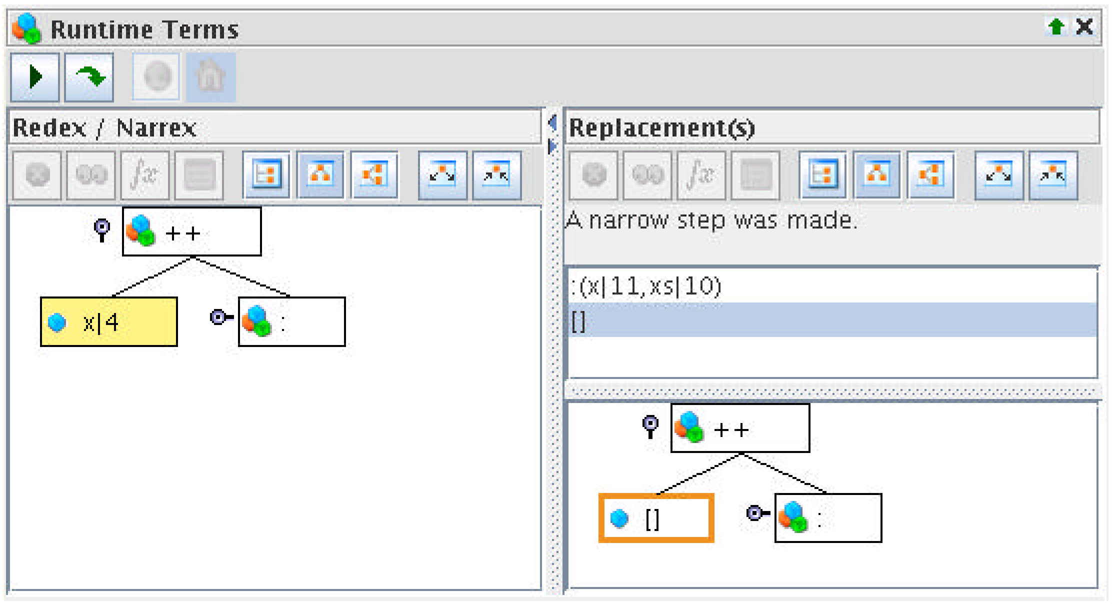


Figure 4.4: Deterministic step in the runtime term viewer

The runtime term viewer also displays non-deterministic steps. These are displayed with the narrex on the left. The right hand side has two parts to it. If the non-determinism was from multiple possible instantiations for a logic variable then the top part displays those instantiations in a list. If the non-determinism was from multiple choices for a function then the right hand from the source code for each of the options is list in the list. When the user selects alternatives from this list the replacement term for that option is displayed in the lower portion. For example, figure 4.5 shows a non-deterministic step for binding the variable `x|4` to either `:(x|11,x|12)` or `[]`. In this figure `[]` is selected so the term from substituting `[]` for `x|4` is displayed in the lower portion.



A narrow step was made.

:(x|11,xs|10)
[]

Figure 4.5: Non-deterministic step in the runtime term viewer

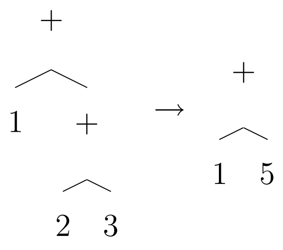
4.1.3 Step Control

There are two ways the user can control which rewrite step they see next. The user can choose to run until the next breakpoint is encountered or the they can see the next rewrite step performed by the virtual machine. Often the user will want to skip over "uninteresting" rewrite steps and just see the ones with breakpoints set on them. Selecting to run until the next breakpoint in encountered will do this. The user may then want to see some of the rewrite steps around the one with the breakpoint. They can do this by single stepping. This shows the next rewrite that the virtual machine takes to the user.

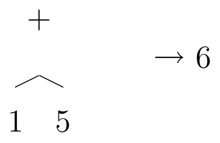
While single stepping can be a useful tool it can also be confusing on an architecture where non-deterministic choices are executed fairly rather than via backtracking. The reason for this is that the next step may be a non-deterministic step that is unrelated to the current step. This makes it very difficult to determine what exactly is going on. When single stepping the user is expecting that the next step is the next step of reducing the result of the current rewrite step. For example consider evaluating `g 1 2 3` with the following program:

```
g x y z = x + y + z
g x y z = x - y - z
```

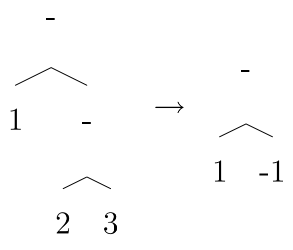
Now say that the user is viewing the following rewrite step:



When they do a single step they expect that the next step they will see is:



However, they may actually see:



They could see the above rewrite step if the two choices for `g` are executed fairly by alternating rewrite steps between the choices. However, this is not the expected behavior for single stepping. Not only is this an unexpected behavior but it is also very confusing. It is difficult to determine which choice is being worked on and where in the sequence of rewrite steps for that term the displayed rewrite step is. This issue can be addressed with choice control (§4.1.5). Using choice control the user can pause all computations except the one that just made the rewrite step. Then they will only see rewrite steps on this term. This issue is also addressed with tracing. In a situation where the user wants to single step through the rewrite steps for `g 1 2 3` they can perform a trace of `g 1 2 3`. When they view this trace they will be able to single step through the trace and the trace does not suffer from this confusing issue.

4.1.4 Runtime Context Hiding

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.

Clearly, it is desirable to work with small terms. Since a debugger cannot change the terms during runtime it must be able to deal with large terms. One way to deal with large terms is with context hiding. Context hiding only shows subterms to the user. The problem is knowing which subterms to display. TeaBag has three options for context hiding of terms during runtime. The default option is to display the subterm that is the root of the redex of the rewrite step. Since the redex is the term that is reduced by the rewrite step it is most likely the term the user is interested in. By just displaying the redex the portion of the term above the redex is hidden. Also, TeaBag will expand terms only as much as is needed to display redex pattern, narrex pattern, and created positions, i.e. to eliminate the portion of the term below the parts of the term that changed. The user has the option to further expand the term to see more.

The second context hiding option TeaBag has is the global context. This context displays the root of the overall term. This does not hide any of the term above the redex. TeaBag will still only expand the term as much as is needed to display the redex pattern, narrex pattern, and created positions. This option lets the user see the entire term if they want.

The third option TeaBag provides for context hiding is a user defined root of the context. This lets the user pick a term that will be the root of the context. Many times a user will not want to see the entire term and just seeing the redex does not give enough context information to really understand what is going on. In this situation the user defined context can be used. This context is helpful when using the single step option. When single stepping the user often wants to be able to see a common context for each of the rewrite steps.

For example Consider evaluating `change 50 [Quarter,Dime]` with the following program:

```
data Coin = Quarter | Dime | Nickel | Penny
type Bag = [Coin]

val :: Bag -> Int
val [] = 0
val (Quarter:xs) = 25 + (val xs)
val (Dime:xs)    = 10 + (val xs)
val (Nickel:xs)  = 5  + (val xs)
val (Penny:xs)   = 1  + (val xs)

genBag :: Int -> Bag
genBag a | a >= 25 = Quarter : genBag (a-25)
         | a >= 10 = Dime    : genBag (a-10)
         | a >= 5  = Nickel  : genBag (a-5)
         | a >= 1  = Penny   : genBag (a-1)
         | otherwise = []

change :: Int -> Bag -> Bag
change price bag = genBag (price - (val bag))
```

To see how `val` is evaluated the user would set a breakpoint on `val` and then single step. If all the user is interested in is how `val` is evaluated they do not want to have to use the global context. However, using just the redex context does not give enough context information. Sometime the user will want this context information and other times they will not need it. To illustrate these differences consider the difference between the following three sequences of rewrite steps. The redex positions are displayed in **bold** and the created positions are displayed in *italics*. The first sequence of rewrite steps in figure 4.6 is the rewrite steps displayed using the redex context. The rewrite step sequence in figures 4.7 and 4.8 illustrates the first two steps using the global context. Finally, the sequence in figure 4.9 uses a user defined context on the root of the subterm rooted with `val`.

Each of the different types of contexts for the rewrite steps have advantages and disadvantages. In figure 4.6 the redex context is able to show the parts of the term that changed automatically. Compared to seeing the entire term in 4.7 and 4.8 this is nice. However, there is no contextual information for where this term is located. Contextual information is most useful when single stepping through some rewrite steps. However, having too much contextual information makes the terms difficult

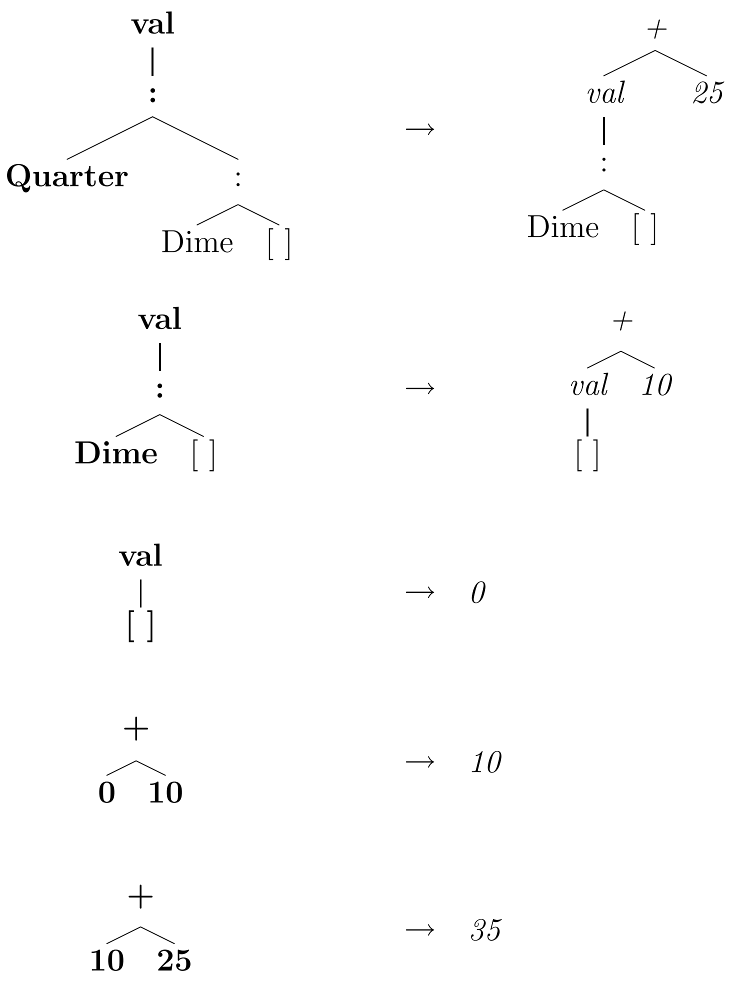


Figure 4.6: Rewrite steps using the redex context.

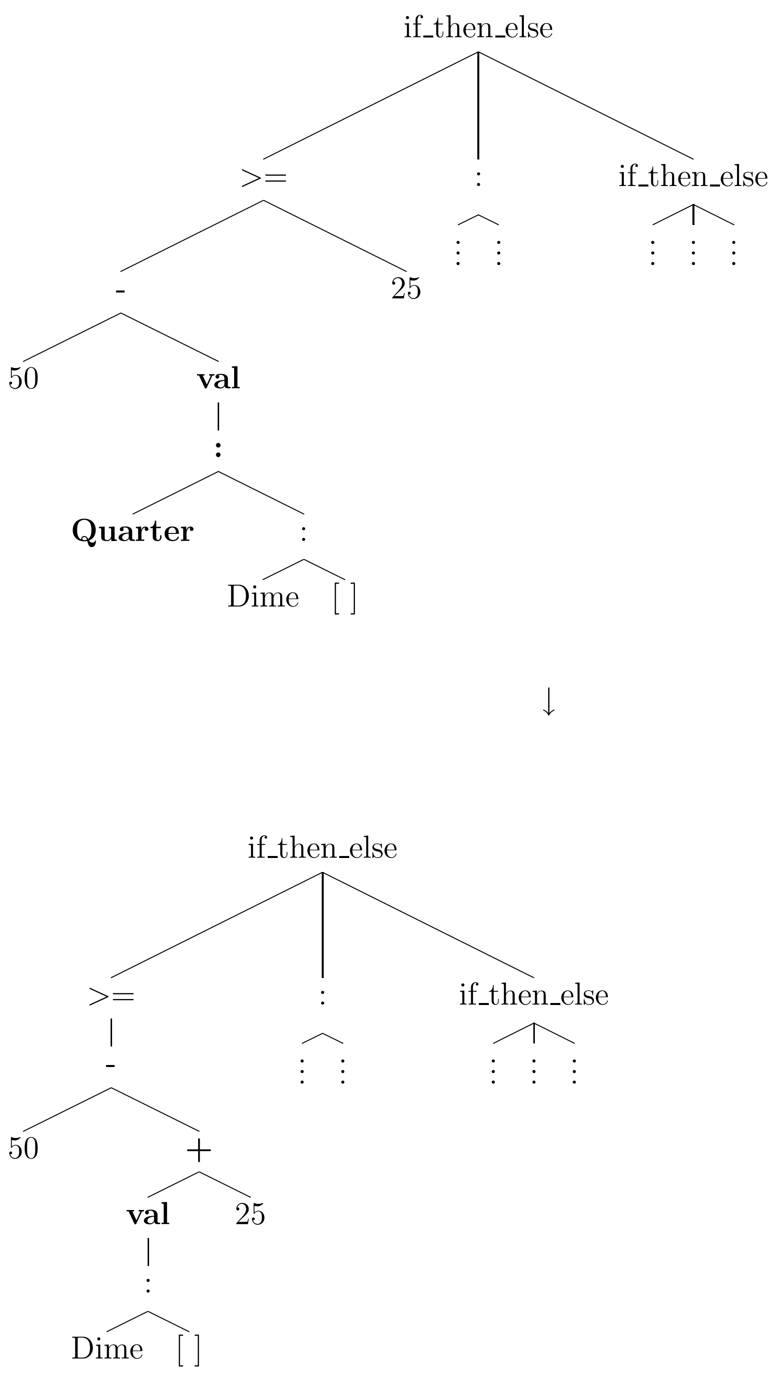


Figure 4.7: First Rewrite step using the global context.

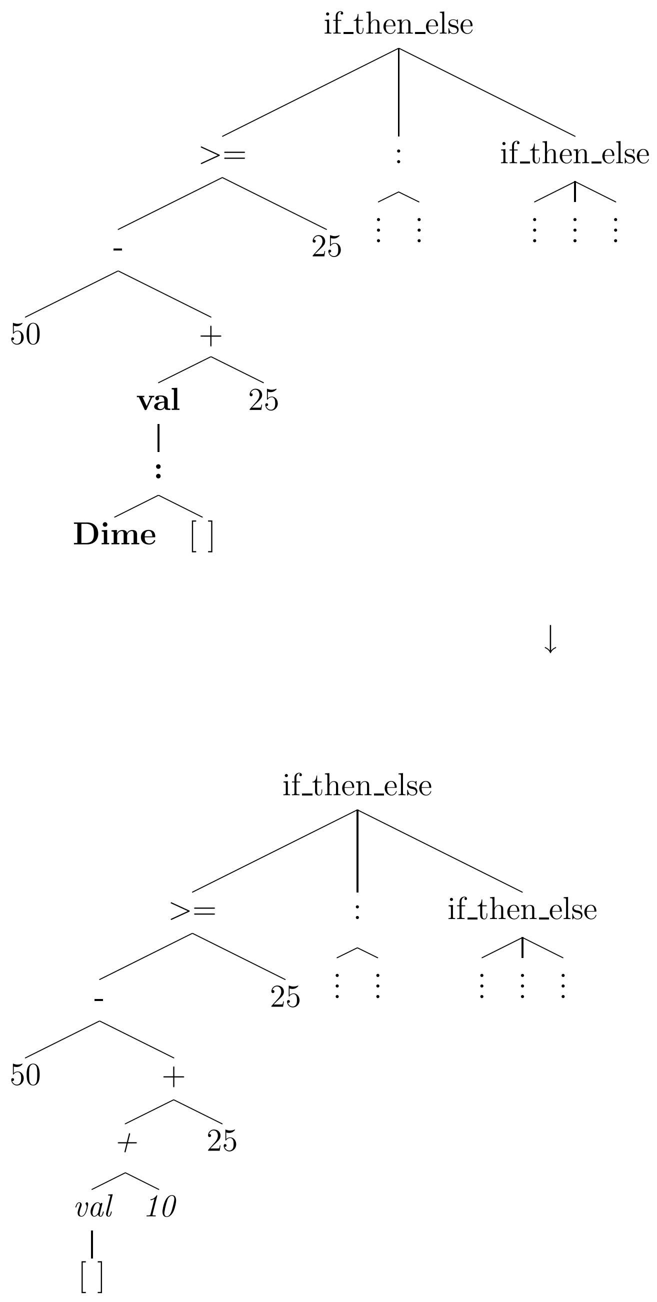


Figure 4.8: Second Rewrite step using the global context.

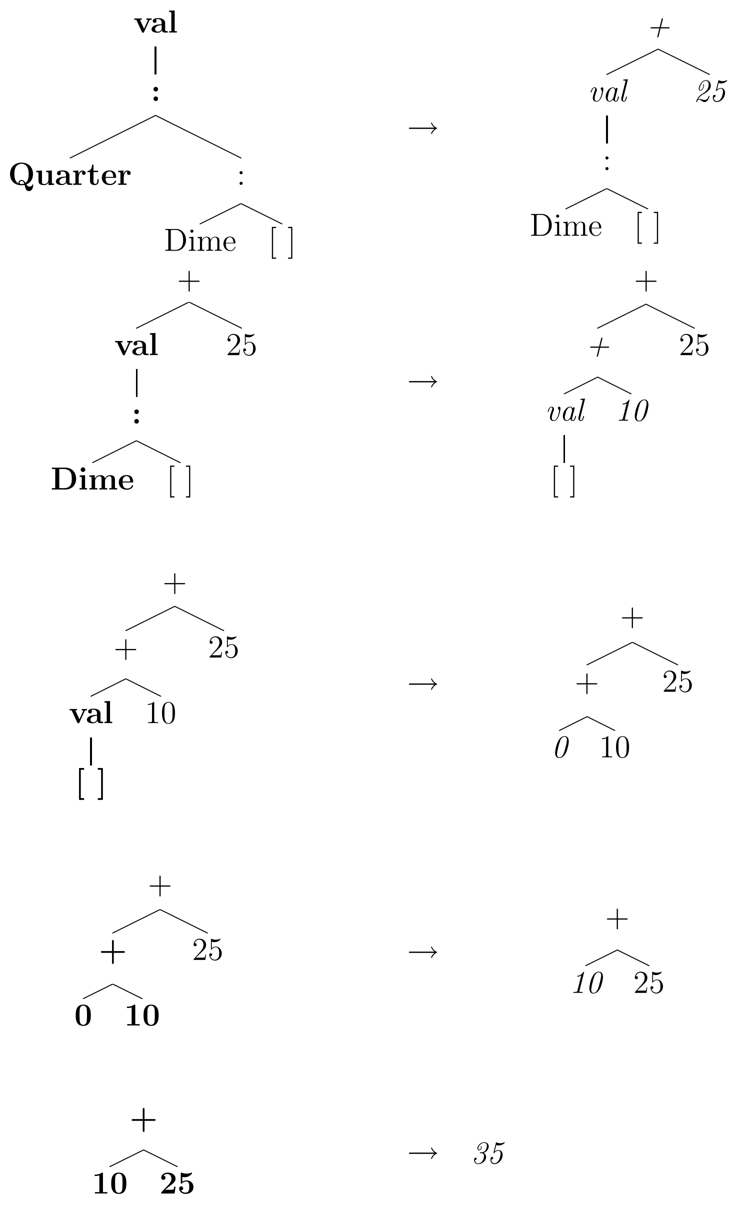


Figure 4.9: Rewrite steps using the user defined context.

to read. This highlights the key difference between the terms displayed using the redex context in figure 4.6 and the user defined context in figure 4.9. Using the user defined context makes reading a *sequence* of steps easier and using the redex context makes reading a *single* step easier. So when the user is only interested in viewing unrelated single steps the redex context is preferable. However, when they wish to view a sequence of steps the user defined context is preferable. This leads to the main disadvantage of the user defined context, the user must select the context. Sometimes this may mean viewing the global context to find the term for the context. So the user defined context is not automatic like the redex and global contexts.

The goal of runtime context hiding in TeaBag is to provide a way for the user to obtain information about terms without being overwhelmed by the sheer amount of information contained within the terms. TeaBag provided three different contexts for addressing this problem; the redex, global, and user defined contexts. Each of these contexts have advantages and disadvantages. Being able to use all three of these contexts in one debugging session gives the user many options for how they wish to view the terms displayed to them.

4.1.5 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.

Since many computations can be created by the FLVM TeaBag provides a hierarchical view of the computations so that the user can see the relationships among the computations. TeaBag also highlights the computations based on their state. If a step is displayed in the runtime term viewer then the computation that performed that step is highlighted in blue. All active computations are highlighted in green. The computations that no longer exist are grayed out. An example of the computation management window is in figure 4.10. This computation management window is for the Dutch national flag problem example presented in section 6.3. Also, this figure shows a user killing a computation by "right-clicking" on it.

4.2 Tracing Features

Runtime debugging limits which narrowing step the user can look at. That is, the user has no way to look at previous narrowing steps or look at narrowing steps in another computation and then return back to the narrowing step they were looking at. Tracing provides a solution to this. Tracing records the narrowing steps while a computation is running. When the computation has finished or when the user has terminated the computation the user can view the trace using the trace browser

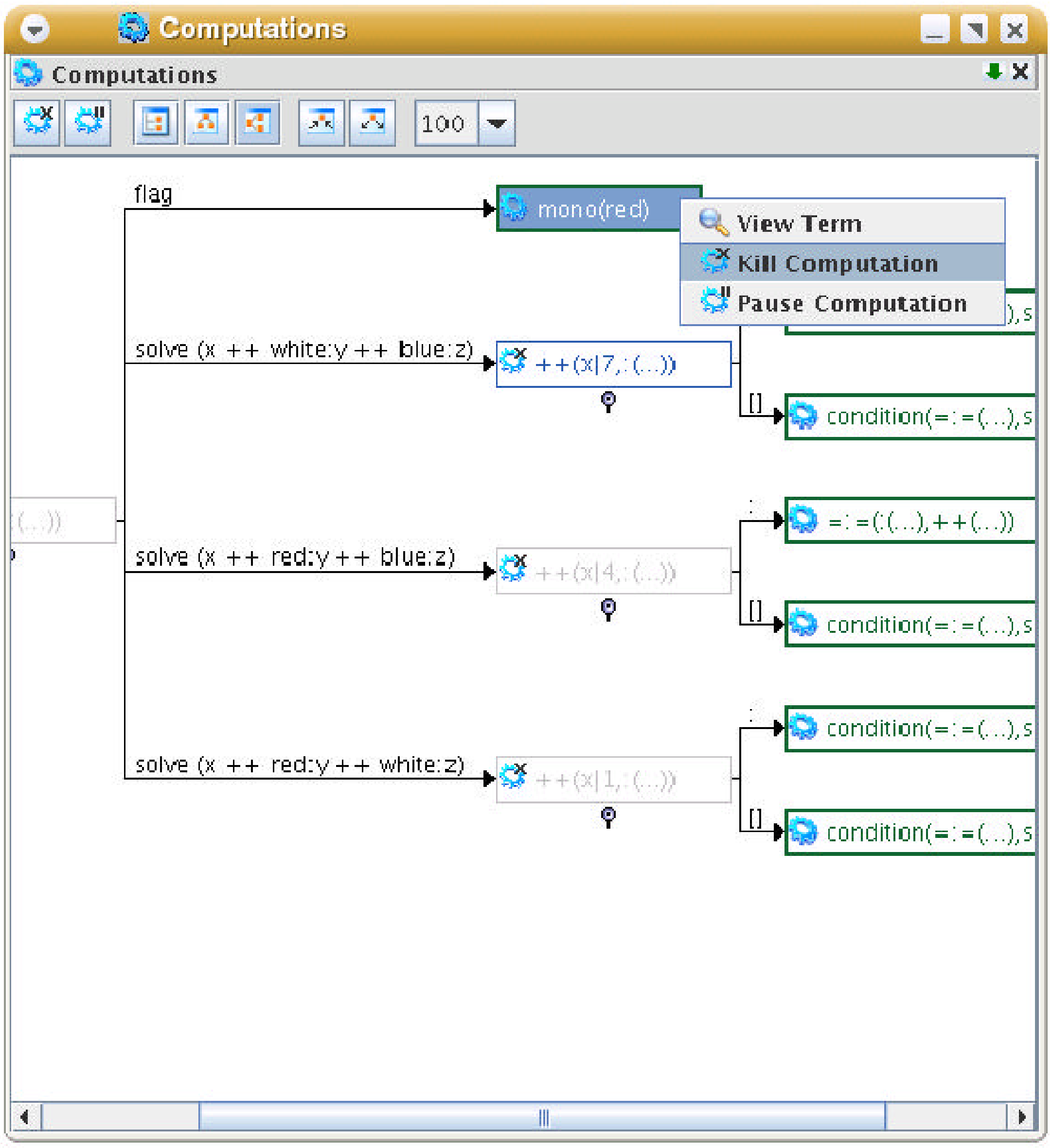


Figure 4.10: Computation Management Window

and computation structure. When viewing the trace the user can choose to look at any step in the trace. Thus they can look at the previous step or at a step from another computation and then return back to the step they were looking at.

4.2.1 Trace Generation

In TeaBag the user must specify which term they want to trace the rewrite steps of. Thus not every computation is traced. This is different from most tracers. Most tracers record all of the steps made during runtime. In TeaBag the user specifies a subset of all the steps to trace. They do this by specifying a subterm they want to trace the computation of. This has two advantages. Firstly, it provides trace context hiding (§4.2.2). Secondly, it lets TeaBag debug many large programs. Currently, large trace generation is an active area of research. The trace generator in TeaBag does not have any of the features necessary for large trace generation. However, by only generating traces of subterms TeaBag can debug many large programs. Generating traces of subterms that do not have too many rewrite steps can be handled by TeaBag even if the computation for the overall term has thousands or millions of rewrite steps.

The user instructs TeaBag to generate a trace for a term by right clicking on a term displayed during runtime debugging (§4.1) and selecting add trace. This lets the virtual machine know to generate a trace for the computation of that subterm. During a typical debugging session the user would find the subterm they want to trace by setting breakpoints using the runtime debugger. Then they would add a trace to that subterm. Next, they would let the virtual machine generate the trace by letting it run (§4.1.3). Finally, they would view the trace using the trace browser (§4.2.4).

4.2.2 Trace Context Hiding

The number of steps in a trace can be very large. Many times they are thousands or millions of steps long. Clearly, it would be difficult to look at each of these steps to find a bug. Many times the user will only want to look a subset of all these steps, in particular, at all the steps that affect a particular subterm of the term being evaluated. For example, they may want to look at the steps 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 narrowing steps performed on that term or one of its subterms. This feature limits the number of 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.

Since traces are only generated for subterms of a computation, the root of that subterm is the root of the term for each trace step. So this provides the same kind of context hiding on terms as does the user defined context in runtime context hiding.

Another way that the number of steps presented in a trace is reduced is with eager evaluation (§4.3.2). If the programmer eagerly evaluates a term and chooses to replace that term in the runtime debugger, and that term is part of a term being traced, then the trace just shows one step for the replacement. This is a technique that the programmer can use to limit the number of steps in a trace. For example, an argument to a function may take 100 steps to rewrite. However, if the user is only interested in how the function behaves and not how the argument is evaluated, they can add a trace to a term rooted with that function. They can then evaluate the argument to the function and choose to replace the evaluated term. This will make the 100 rewrite steps necessary for the argument be just one rewrite step in the trace.

The user can also use choice control (§4.1.5) to reduce the number of steps in the trace. The user can kill or pause computations that compute some of the steps for the trace. When they do this the trace will not contain those steps. This also makes selecting a path through the computation structure (§4.2.5) easier since killing computation reduces the size of the search space.

One of the unique features of TeaBag is its interaction between the runtime debugger and tracer. TeaBag allows the user to have some control over trace generation via runtime debugging features. This can help to make traces smaller. Also, this “blurring” of the line between runtime debugging and tracing provides the user with more ways to use the trace for debugging.

4.2.3 Trace Steps

In TeaBag there are two different kinds of trace steps, one for deterministic steps and one for non-deterministic steps. Trace steps for deterministic steps have two terms. One term is for the term before being rewritten, the redex, and the other one is the result of the rewrite step. Thus deterministic steps are presented as terms before and after rewriting. For example, in figure 4.11. The term on the left, `add (S Z) Z`, is the redex and it is rewritten to the term on the right, `S (add Z Z)`.

The narrex of a non-deterministic step may rewrite to many terms. Non-

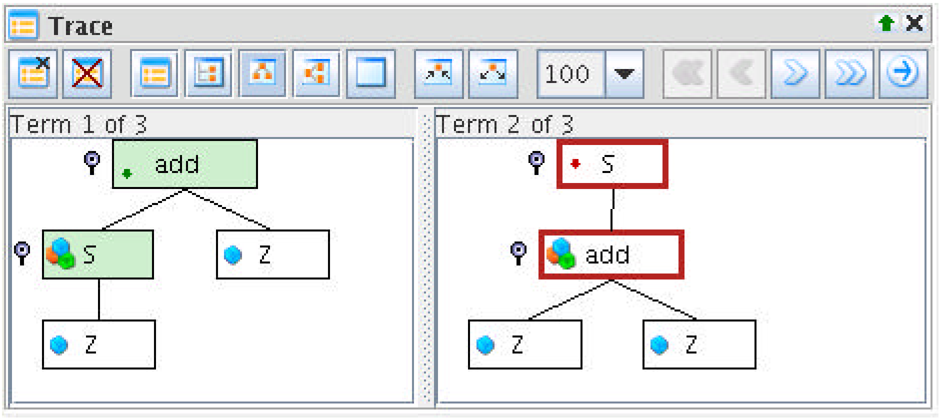


Figure 4.11: Deterministic trace step

deterministic steps are presented as a narex term with many possible terms it could be rewritten to. For example, the term on the left in figure 4.12 has two possible replacements. The replacements are shown by selecting a binding for the free variable in the list located in the upper right corner of the trace browser. The first replacement is shown in figure 4.12 and the second is show in figure 4.13. As a side note, in TeaBag logic variables are displayed as `name|number` where the name is the source code name of the logic variable and the number is a unique number assigned to each instance of a logic variable by the virtual machine. Displaying the unique number for logic variables assists the user in distinguishing between two logic variables with the same name that are not the same instance of the variable.

4.2.4 Trace Browser

TeaBag has a trace browser that lets the user view the steps of the trace. During runtime the trace is saved to a temporary file. To make viewing the trace contained

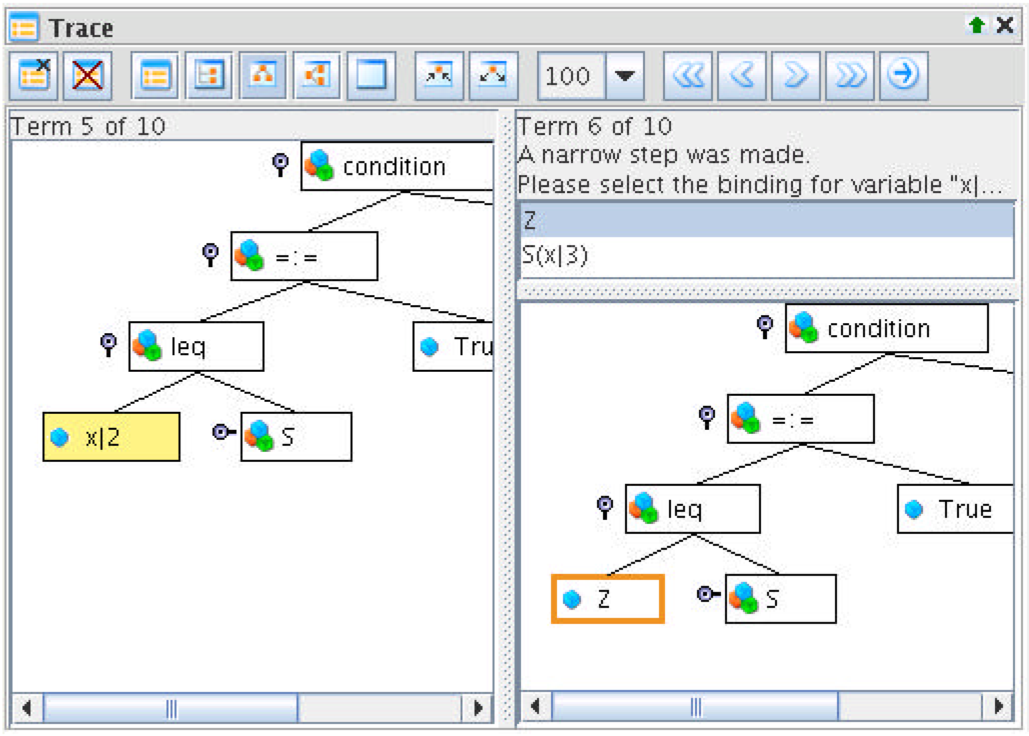
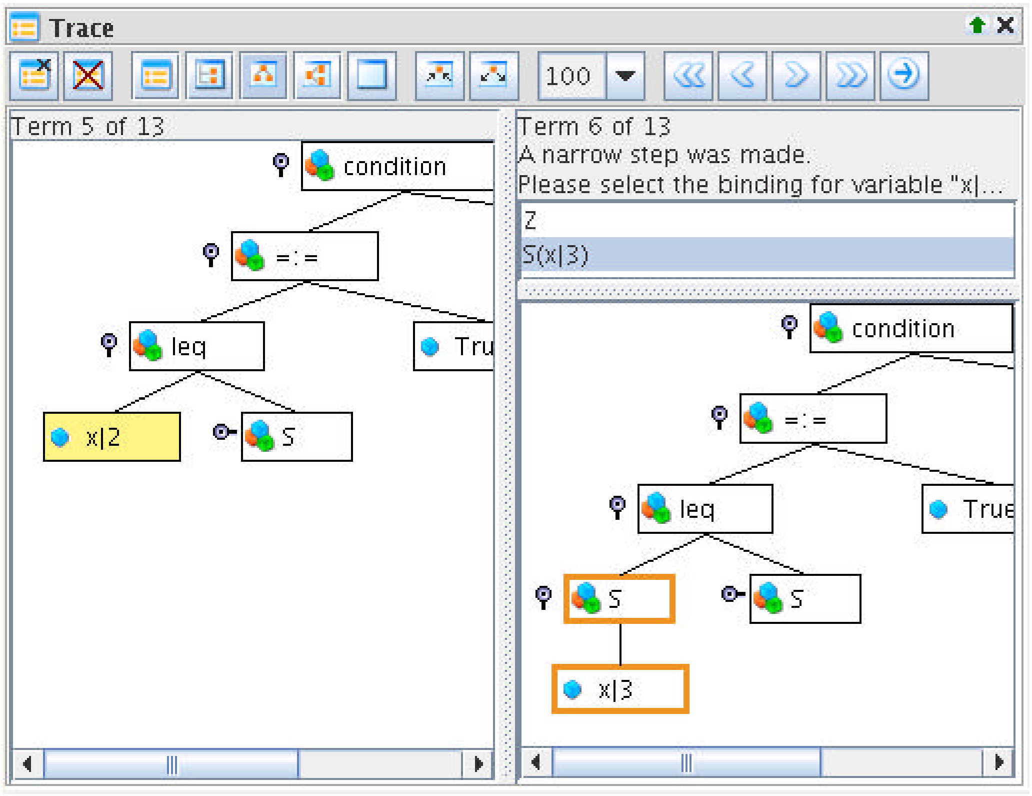


Figure 4.12: Non-deterministic trace step example for binding `x|2` to `Z`



A narrow step was made.
Please select the binding for variable "x|...
Z
S(x|3)

Figure 4.13: Non-deterministic trace step example for binding `x|2` to `S(x|3)`

in this file easy, TeaBag provides a trace browser. The trace browser shows terms in the trace, lets the user pick how to view the trace, and provides buttons to navigate through the steps of the trace.

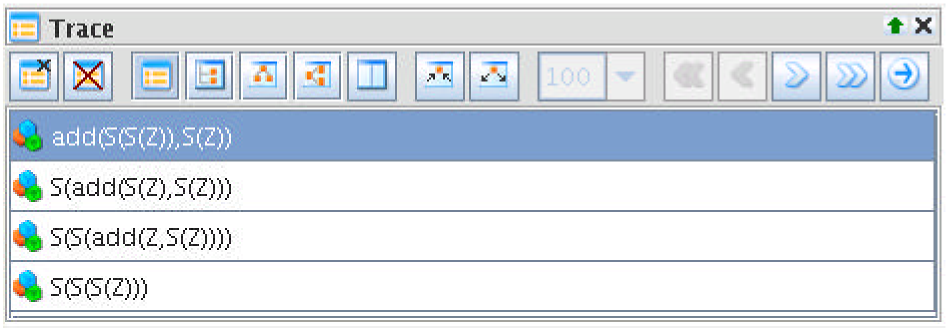
The trace browser in TeaBag provides two different ways to view the trace. The first way is by displaying all of the terms in the trace at once. Each of the terms are displayed on a single line. This view of the trace shows each of the rewrite steps along the selected path in the computation structure (§4.2.5) at once. This view is good for getting a “birds eye” view of the trace. The user can use this view to determine where in the trace they want to start looking in more detail. For example, figure 4.15 shows all of the rewrite steps for evaluating `add (S(S Z)) (S Z)` using the program in figure 4.14.

```
-- natural numbers defined by s-terms (Z=zero, S=successor):
data Nat = Z | S Nat

-- addition on natural numbers:
add :: Nat -> Nat -> Nat
add Z     n = n
add (S m) n = S(add m n)
```

Figure 4.14: Addition on Natural Numbers

The second way to view the trace in TeaBag is one rewrite step at a time. This view gives the user more information about each rewrite step. The terms are displayed as trees. This lets the user easily choose which parts of the term to view. Also, the redex pattern, created positions, and variable bindings are highlighted (see section 4.3.1 for more information on highlighting). Figures 4.11, 4.12, and 4.13 are examples of the trace browser.



add(S(S(Z)),S(Z))
S(add(S(Z),S(Z)))
S(S(add(Z,S(Z))))
S(S(S(Z)))

Figure 4.15: Trace Table Example

The trace browser has five controls for navigating the trace. It lets the user move to the next or previous trace step. Also, the trace browser gives the user the option to jump to the first or last step in the trace. Finally, the trace browser lets the user jump to any step number in the trace.

4.2.5 Computation Structure

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* [11] 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 can be a head-normal form, a failure, or a term that needs further evaluation. 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 4.2.4, shows all of the 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 narrowing step fits into the overall computation. The computation structure lets the user know which non-deterministic steps were made to get to any 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. Penny [54] mentions that having a sequence number for linear functional tracing is important because it gives the user a reference to where the trace step occurs. The user can use this information to place bounds on sections of the trace thought to contain the bug. The user can use the computation structure to place bounds on where the trace might contain the bug by saying that the bug might be located between two steps along a particular branch of the computation structure.

The computation structure for evaluating `goal` from the example presented in section 3.2 is shown in figure 4.16. The computation structure in figure 4.16 shows each of the results of evaluating `goal`. To get the result `Z`, the logic variable `x|1` had to be bound to `Z`. If however, `x|1` was bound to `S(x|2)` then the result, S(Z), is obtained from binding `x|2` to `Z`. So this computation structure shows all of the non-deterministic steps and their relationships for evaluating `goal`.

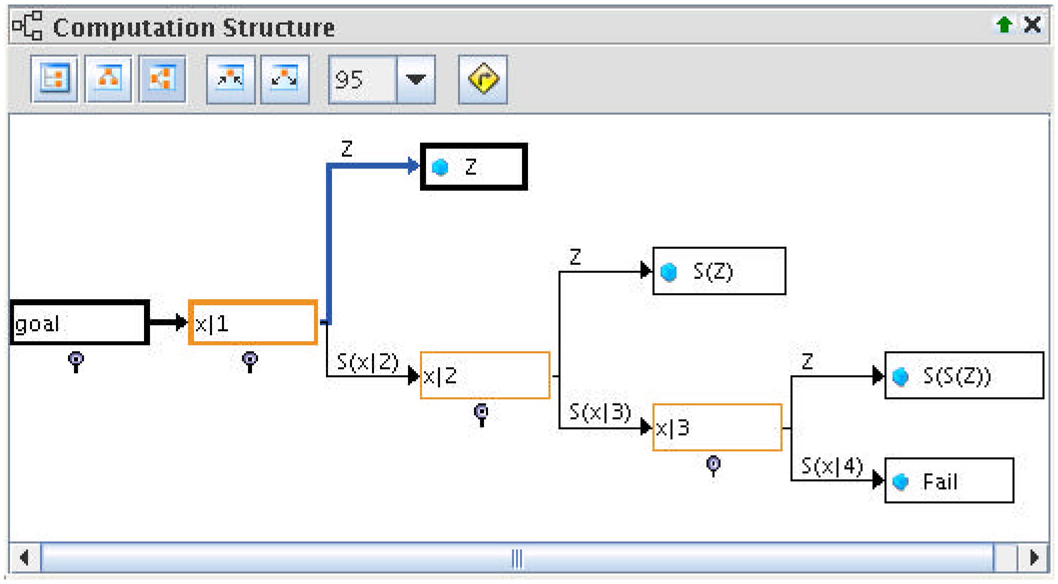


Figure 4.16: Computation structure for trace of `goal`

4.2.6 Trace Browser and Computation Structure Interaction

The trace browser and computation structure interact in TeaBag. The trace browser shows the rewrite steps on the selected path through the computation structure as a linear sequence of rewrite steps. The user can select paths through the computation structure to view. This can be done either in the computation structure itself or in the trace browser. In the computation structure the user can select any node and instruct the trace browser to jump to the trace step for that node. This will update the sequence of steps in the trace browser to a path that includes the selected node. It will also cause the trace browser to jump to the step for the selected node. In the trace browser the path through the computation structure can be chosen by choosing which branch to follow at each non-deterministic step. The computation structure will highlight the path selected.

The two different approaches to select a path through the computation structure are appropriate to different situations. Selecting nodes in the computation structure lets the user jump to any non-deterministic step in a trace. This method of selecting a path in the computation structure is good when the user can determine which choice to make at non-deterministic steps without much contextual information. Selecting a path to follow in the trace browser requires the user to explicitly pick a branch to follow for every node. Selecting a path in this manner is useful when the user needs contextual information to know what branch to select. In the trace browser the user can see what the previous step to a non-deterministic step is. Also, they can see the entire term for the step. This contextual information can assist in determining which branch to follow. The drawback of this approach is that the user must move more slowly through the trace. However, these two methods of path selection can be used together. Thus the user could jump to a point in the trace and then use the trace browser to determine which branch to follow for the next non-deterministic step. So using these two ways of selecting a path in the computation structure together provides the user with many options for selecting the path they wish to look at.

4.3 General Features

Some of the features of TeaBag are not specific to either runtime debugging or tracing. These features are highlighting, eager evaluation, customizable GUI, and virtual machine control.

4.3.1 Highlighting

Like many modern debuggers TeaBag provides highlighting to quickly draw the users attention to important information and to relate the source code to the information presented. TeaBag highlights six pieces of information; redex pattern, created positions, variable to be bound, variable binding, source code for narrow step, and selected path in computation structure.

Term Highlighting

The first four kinds of highlighting are for terms. Highlighting terms is meant to help the user quickly understand the important parts of the term in the rewrite step. This helps the user focus on what changed during a rewrite step. The first kind of term highlighting is redex pattern highlighting. A term is rewritten by replacing a redex. A redex is identified by a redex pattern. When rewrite steps are presented in the debugger the redex pattern is highlighted by coloring the nodes in the redex pattern green. For example, the redex pattern of `add (S Z) Z` as defined in figure 4.14 is `add (S _) _` where `_` stands for any term. So the nodes for `add` and `S` are highlighted in green as shown in figure 4.17.

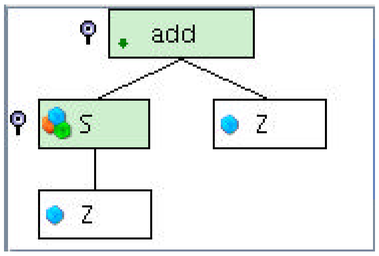


Figure 4.17: Redex pattern highlighting example

When a term is rewritten, the redex is replaced by creating a new term. Only some parts of this term are actually created. New occurrences are created for non-variable symbols on the right hand side of the rewrite rule. Together, these constructors make the created positions for one rewrite step. For example, when `add (S Z) Z)` is rewritten to `S add Z Z` using the rule `add (S m) n = S (add m n)` the created positions are `S` and `add` since they are not variables in the right hand side of the rule. In TeaBag the created positions of rewrite steps are highlighted by outlining the node in red. An example of created position highlighting is in figure 4.18.

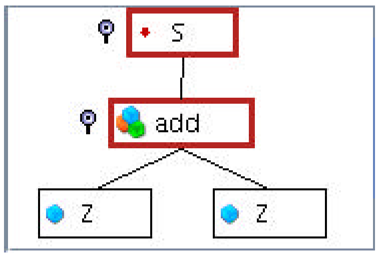


Figure 4.18: Created position highlighting example

Sometimes a rewrite step involves binding a logic variable to a term. When a variable is bound the node for the variable is highlighted in orange. The binding for the variable is outlined in orange. For example, in the figure 4.19 the variable `z|12` in the term on the left is bound to `blue:[]` in the term on the right.

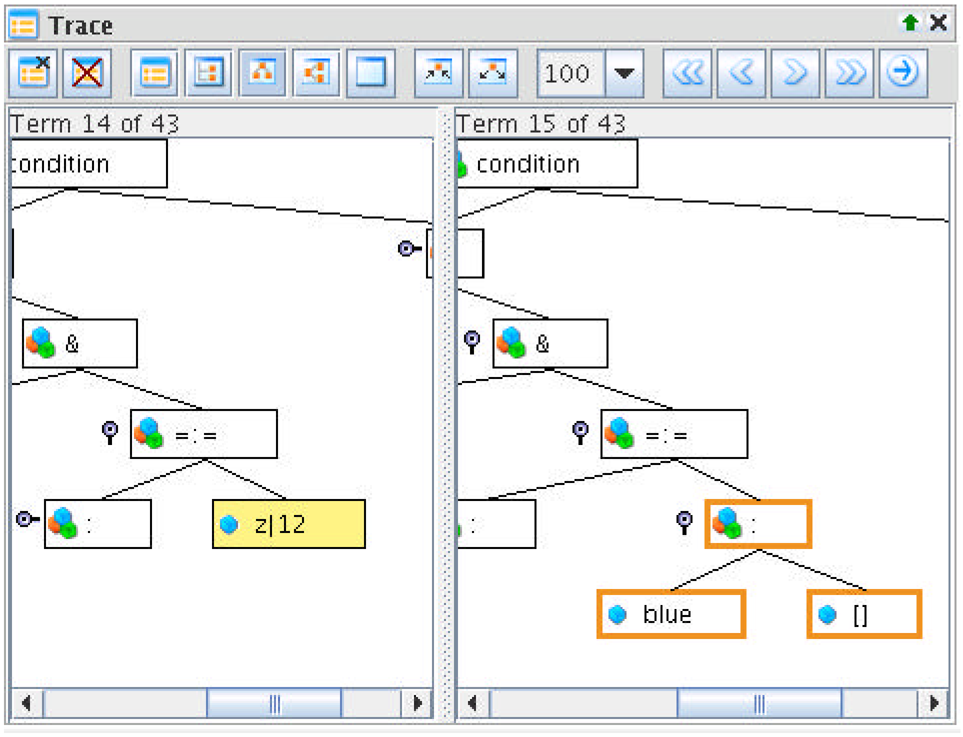


Figure 4.19: Variable binding highlighting example

Source Code Highlighting

When source code for a rewrite step is available TeaBag will highlight the source code. So when `add (S Z) Z` is rewritten to `S (add S Z)` the rule

```
add (S m) n = S (add m n)
```

will be highlighted in the source code. Source code highlights occurs in both the runtime debugger and tracer. Having source code highlighting is important so that the user can relate the rewrite step back to source code. Ultimately a bug will be fixed by changing the source code. So relating the information presented in the debugger back to the source code is important.

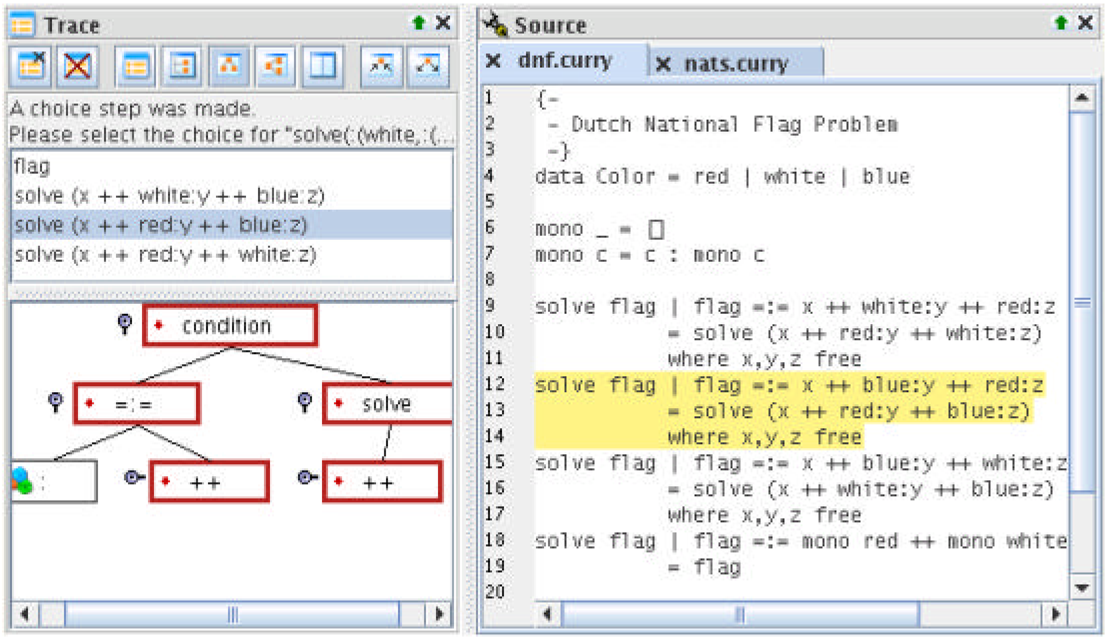
The source code, if available, is also highlighted when the user is choosing which branch to follow for a non-deterministic choice step in the trace browser. This helps the user relate their choice to the source code. For example, consider the code presented in figure 4.20. This code solves the *Dutch National Flag* problem in the spirit of [29], i.e., by swapping pebbles out of place. When the term `solve [white,red,blue,white]` is rewritten there are four possibilities for its replacement. At this step in the trace browser the user will have to select which replacement to follow. To assist the user in this selection the source code for each of the replacements is highlighted when the user selects different replacements. Figure 4.21 is an example of this. In this example the user has selected the replacement corresponding the second rule of `solve`. This rule is highlighted in the source code.

```
data Color = red | white | blue

mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ red:z
      = solve (x ++ red:y ++ white:z)
      where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red:z
        = solve (x ++ red:y ++ blue:z)
        where x,y,z free
solve flag | flag =:= x ++ blue:y ++ white:z
       = solve (x ++ white:y ++ blue:z)
          where x,y,z free
solve flag | flag =:= mono red ++ mono white ++ mono blue
          = flag
```

Figure 4.20: Dutch National Flag Problem



```
A choice step was made.
Please select the choice for "solve(:(white,:(...
flag
solve (x ++ white:y ++ blue:z)
solve (x ++ red:y ++ blue:z)
solve (x ++ red:y ++ white:z)
```

```
{-
 - Dutch National Flag Problem
 -}
data Color = red | white | blue

mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ red:z
           = solve (x ++ red:y ++ white:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red:z
           = solve (x ++ red:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ white:z
           = solve (x ++ white:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= mono red ++ mono white
           = flag
```

Figure 4.21: Choice step highlighting example

Computation Structure Highlighting

To help the user relate the information in the trace browser to the computation structure there are two kinds of highlighting in the computation structure. The first one is the path of trace steps being viewed in the trace browser is highlighted in the computation structure. This shows the user what path of rewrite steps they are looking at. Also, the location of the step being looked at in the trace browser is highlighted in blue. Thus the user not only knows the path in the computation structure that they are looking at, but they also know where on that path they are.

For example, in figure 4.16 the selected path is for computing the result `Z`. The current step is on the branch between the non-deterministic step for binding `Z` to `x|1` and computing the result `Z` (In a black and white print out the blue highlighting for the current step is difficult to distinguish from the black highlighting).

4.3.2 Eager Evaluation

TeaBag includes two features for eager evaluation. It has on demand eager evaluation and optional evaluation of if statements. In TeaBag eager evaluation means evaluating to head normal form. On demand eager evaluation lets the programmer choose when to perform eager evaluation. The programmer can choose to evaluate any term displayed in TeaBag. This gives the programmer the ability to choose when to perform eager evaluation. When a term is eagerly evaluated the results of the evaluation are displayed in a window. The term will be evaluated until head normal form is reached or until a non-deterministic step is made. If the term being evaluated would cause the computation to split due to non-determinism the eager evaluation stops and lets the user know that the eager evaluation halted early. It displays the term that caused the non-determinism to the user. Figure 4.22 has an example of performing eager evaluation on `add (S Z) (S Z)` using the program presented in section 3.2.

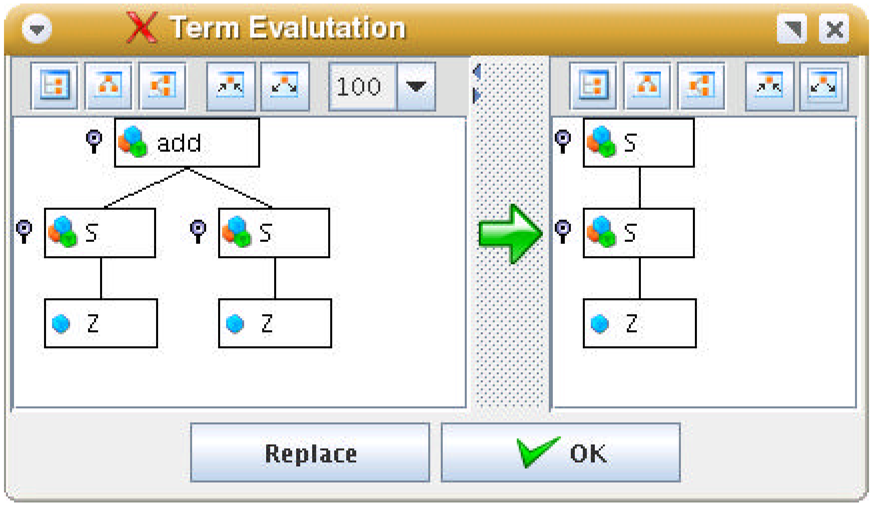


Figure 4.22: On demand eager evaluation of `add (S Z) (S Z)`

When a term is evaluated it is not replaced. This lets the user see what the term evaluates to without changing the lazy behavior of the program. If a user wishes to see what an argument to a function evaluates to and they want to preserve the lazy evaluation of the program they can. There are also times when the user may want to replace the evaluated term. If the term was evaluated from the runtime debugger then the user has the option to replace the evaluated term. This gives the programmer the option to replace any of the terms in the runtime debugger with their head normal form. The reason a user can not replace terms in a trace is because the trace is viewed after the execution of the program. So the trace just shows what happened when the program ran where as the runtime debugger shows what is happening while the program is running.

One area we thought that users may want to always use eager evaluation was in evaluation of conditions. Since conditions are compiled to `if then else` terms we gave the user the option to always eagerly evaluate `if then else` terms. In this case the term is not evaluated to head normal form. Rather, the if condition is evaluated to head normal form and then the single rewrite step for `if then else` is performed. If the replacement is another `if then else` term then the process is repeated. Thus, the term is evaluated until it is no longer rooted with an `if then else` term. This lets the user see which if branch in a nested `if then else` term is taken without executing the code for that branch. We felt that this was very helpful for conditions. Having this feature lets the user see which condition in a function is taken. This also lets the debugger highlight which condition in a function was used for the rewrite step. Without eagerly evaluating the `if then else` term the debugger can not know which condition will evaluate to true when a break point is hit since the evaluation has not occurred yet. It is still possible that the user wants their program to be evaluated completely lazily. So we made this feature an option that the user can turn on or off.

Our general goal with eager evaluation was to preserve the lazy evaluation of the program as much as possible while still giving the user the opportunity to see the eager evaluation of terms. Obviously these are two contradicting goals. We feel that the on demand eager evaluation features in TeaBag does a good job at accomplishing these goals. It lets the user see the evaluation of a term without actually changing the lazy runtime behavior of the program. However, the user still has the option to change the lazy runtime behavior of the program if they wish. We feel that this eager evaluation scheme lets the user have the eager evaluation information they may want while still having full lazy evaluation of the original term.

4.3.3 Customizable GUI

A common feature found in many modern applications with a GUI (Graphical User Interface) is the ability to customize the interface. Different users have different ways they want to look at data. One data presentation that suits one user's needs may not suit another's. Thus, TeaBag provides a customizable GUI. TeaBag has four features for customizing the GUI: *panel layout*, *perspectives*, *tree view*, and *tree zoom*.

TeaBag's GUI is made up of panels. Each panel contains one particular type of information. For example, there is a panel for source code, trace browser, etc. Each of the panels in TeaBag can be docked or undocked. An undocked panel is in its own window. This lets the user place the panel anywhere that the host operating system lets a window be placed. A docked panel is one that is contained within the main TeaBag window. Being able to undock a panel is nice when that panel contains a lot of information since the undocked panel can fill the entire screen. Docking a panel is convenient for seeing multiple panels at the same time.

When a panel is docked the user can decide where to place the panel in TeaBag by dragging the panel to the desired location. There are nine regions in TeaBag that the panels can be located. The nine regions are defined by a three by three grid. The size of each grid location is also customizable by the user. Between each grid location is a dragable bar. The user can drag the bar to make the region bigger or smaller. It is possible for more than one panel to occupy the same grid location. When this happens there are two options for how the panels are arranged. The first option is that both of the panels occupy the same space. It this case only one of the panels is visible at a time. When multiple panels occupy the same space a tab for each panel is placed below the panels. This lets the user choose which panel to view in that space. The second option for having multiple panels in the same grid location is that they are placed next to each other. In this case the panels are arranged vertically when they are in the far left and right columns. When the panels are in the center column they are arranged horizontally. The user selects which of these options they want by where they drag the panel. If they drop the panel over the title bar of another panel then the two panels will occupy the same space. By dropping the panel to the side, top, or bottom of another panel then TeaBag will decide if the panel should be put into a new grid location or if it should occupy the same grid location as the other panel and be place next to it. As the user drags a panel around, the cursor changes it icon to indicate where the panel will be placed.

TeaBag also has the notion of perspectives. A perspective is a collection of panels where each panel has a location and size. The location of each of the panels can be either a location in the three by three grid in the main TeaBag window or it can be the location of the containing window for an undocked panel. The size of each of the panels is either the size of the grid it is in or the size of the containing window for an undocked panel. Each perspective gives the user a particular view of the data. TeaBag comes with two default perspectives. They are *debug* and *trace.*

The debug perspective has all of the panels used for runtime debugging arranged with the configuration the user has chosen. Likewise the trace perspective has all of the panels used for tracing arranged with the configuration the user has chosen. If the user wants yet another view of the data they can create their own perspective. For example, a user may want to have a view that shows some of the runtime debugging information and some of the tracing information at the same time.

Much of the information in TeaBag is presented as trees. TeaBag gives the user three options for how to view each tree. The user can choose to view the trees using the normal tree view which is a typical GUI widget tree as shown in figure 4.23. The user can also choose to view the tree using the natural tree which is a top down tree as shown in figure 4.24. Finally, the user can choose to view the tree using the horizontal tree as shown in figure 4.25. Each of the tree views can be individually selected for each tree. Having three different views for trees lets the user pick the best view for a given situation. For example, the normal tree view is a more compact view of the tree. So it lets the user see more of the tree in a smaller space. This view is also good when there is a large branching factor in the tree since this view will present the data as lists. However, with the normal view is not as easy to visually group pieces of information together. The top down tree is good for this. The top down tree makes it easier to identify groups of data and their relationships. The horizontal tree works well for presenting sequential data. Thus, the user can choose which tree view they want for each of the trees. By default TeaBag selects the view that is most appropriate most of the time.

Finally, TeaBag lets the user select a zoom level for the top down and horizontal trees. Since these tree representations are not as compact as the normal tree,

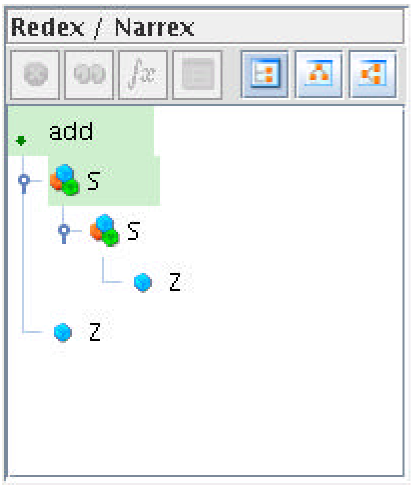


Figure 4.23: Normal tree representation of `add (S (S Z)) (S Z)`

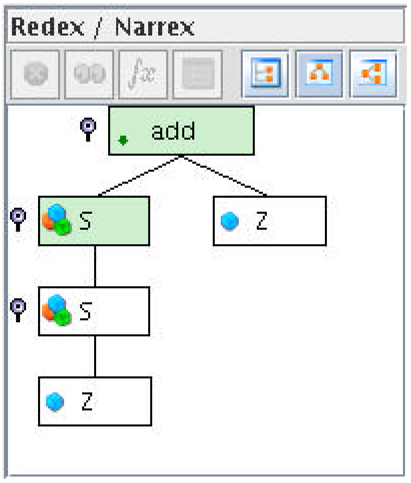


Figure 4.24: Top down tree representation of `add (S (S Z)) (S Z)`

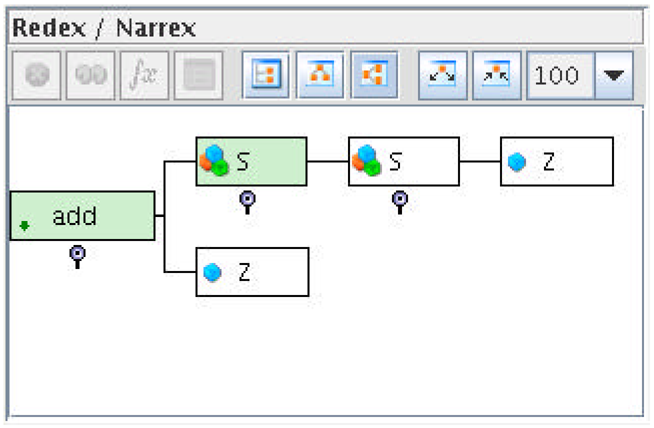


Figure 4.25: Horizontal tree representation of `add (S (S Z)) (S Z)`

TeaBag lets the user zoom out and see more of the tree at once. Of course this is at a cost of readability. The text in the tree may become difficult or impossible to read as the user zooms out. However, this option lets the user get an overview of the structure of the tree. Once the user understands the overall structure they can zoom back in and read the details.

4.3.4 Virtual Machine Control

The Curry virtual machine that TeaBag interacts with is run in a separate process from TeaBag (see chapter 5 for an architecture description). This allows TeaBag to independently start and stop the virtual machine. This feature is very convenient for the user. This provides two debugging benefits. The first one is for tracing non-terminating computations. If the user terminates the virtual machine in the middle of collecting a trace then all of the trace steps up to the point of terminating the virtual machine will be available in the trace browser and computation structure. Also, to repeat a particular bug the user will typically want to start a computation over. To make sure there is no lingering state information in the virtual machine, the user can easily restart it and run the computation again.

Chapter 5

Architecture

Ian Sommerville states [62]

> Large systems are always decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called *architectural design* ...

This section describes the main sub-systems and packages of TeaBag. It also defines the means of communication among these systems. There are two main sub-systems in the TeaBag system. They are TeaBag itself and the FLVM. TeaBag is further decomposed into packages. The architecture of TeaBag also includes identifying these packages and specifying the package communication framework.

Up to this point references to TeaBag have included the entire debugging system which includes TeaBag and the FLVM. However, in reality these are two different systems. TeaBag is just an application that interfaces to another application the FLVM. Thus, from this point forward references to TeaBag refer to the application TeaBag which does not include the FLVM.

5.1 Sub-System Architecture

While TeaBag is a debugger for Curry it was developed to work with a particular implementation of Curry called FLVM [14]. However, we did not want to limit TeaBag to working with just FLVM. Thus one of the architecture goals was to decouple FLVM from TeaBag in such a way that another implementation of Curry could be used. To do this TeaBag and FLVM are run in separate processes and communicate over sockets. This means that any Curry implementation that implements the socket interface will work with TeaBag.

There are three sockets used to communicate between TeaBag and FLVM. The first socket is known as the command socket. Normal user commands are sent over this socket from TeaBag to FLVM and consol output is sent from FLVM to TeaBag over this socket. This allows the virtual machine to redirect the standard output and input streams to this socket and treat it like it would a consol. The second socket is known as the data socket. FLVM events are sent over this socket from FLVM to TeaBag. Debugging commands are sent from TeaBag to FLVM over this socket. Some of the commands that TeaBag sends to the virtual machine can be sent in the middle of a computation. Thus the command socket can not be used since the virtual machine treats it like consol input. These commands are sent over the data socket. Finally, the last socket is called the exception socket. All error messages in the virtual machine are sent over this socket as plain text. Since there is no special error message format the virtual machine can redirect standard error to this socket. Thus all error messages sent to standard error will now be sent to TeaBag as error messages. Allowing the virtual machine to redirect standard input, output, and error streams to sockets make the task of sending information back and forth easier.

The process for FLVM is started by TeaBag. When TeaBag starts the process for FLVM it passes the port numbers for each of the three sockets to FLVM. FLVM then opens a socket connection back to TeaBag on the specified ports. Just before starting the process for FLVM TeaBag starts a thread that waits for each of the sockets to be connected. This thread times out if the socket connections are not opened in the alloted time. TeaBag expects that first the command socket will connect, then the data socket, and finally the exception socket in that order.

5.1.1 Sub-System Communication

Most of the socket communication between TeaBag and FLVM is event driven rather than sequence based. This means that there is not a conversation between TeaBag and FLVM over the socket. Rather self-contained events are sent back and forth between the two systems.

In TeaBag there are only two classes that know about the existence of a socket, `PacketParser` and `Main`. The rest of TeaBag views the communication with the virtual machine as event based. The class structure of the events for virtual machine communication is in figure 5.1. (UML notation used in this thesis follows OMG Unified Modeling Language Specification version 1.5 [51]). The communication over the socket follows the specification laid out in section 5.4. Data is sent over the socket in packets. There is a single instance of the `DataParser` class running in a thread in TeaBag. This thread runs in an infinite loop. The pseudo code for this loop is in figure 5.2. The `DataParser` first parses a packet from the

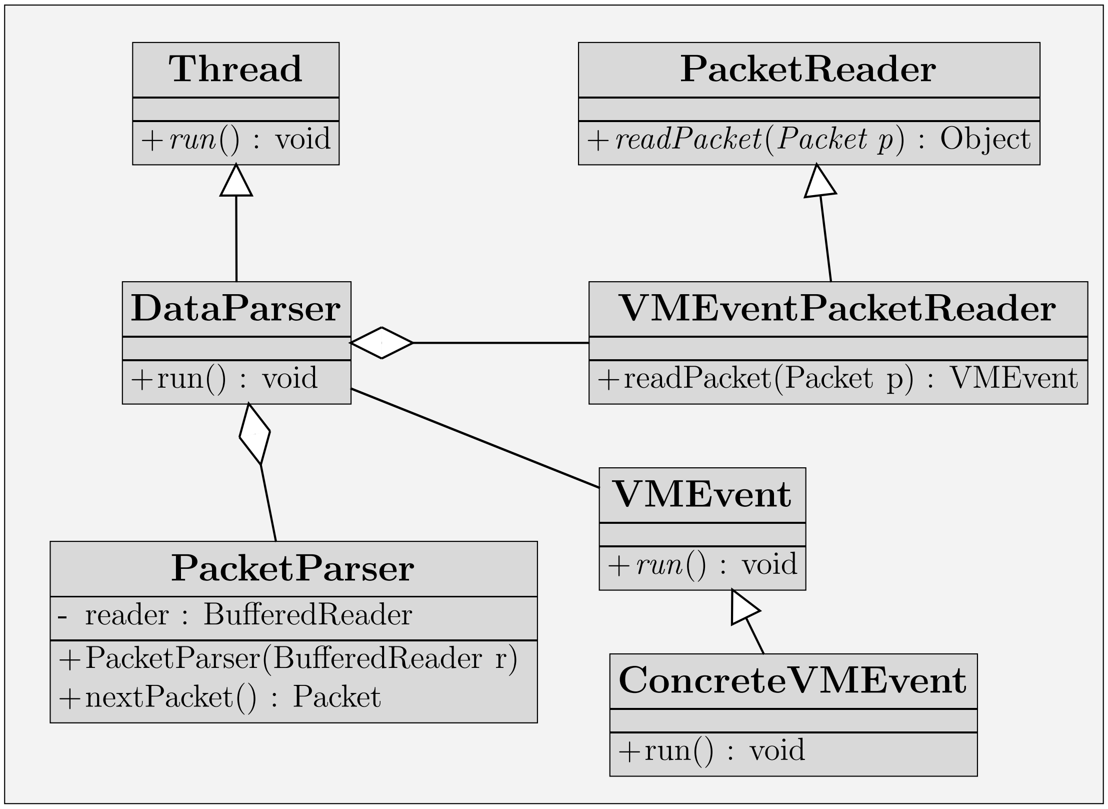


Figure 5.1: Class structure of virtual machine events

```
PacketParser        parser = new PacketParser(dataSocket);
VMEventPacketReader reader = new VMEventPacketReader();

while (true){
    Packet  p = parser.nextPacket();
    VMEvent e = reader.parsePacket(p);
    dispatchEvent(e);
}
```

Figure 5.2: Pseudo code for `DataParser` thread.

data socket via the a `PacketParser` object. If no data is available on the data socket then the `PacketParser` object will block. Getting a full packet from the data socket indicates that the virtual machine has fired an event. Once a packet is obtained from the `PacketParser` it is passed to a `VMEventPacketReader` which creates a concrete `VMEvent` for the event specified in the packet. This event is then dispatched into the Swing event loop by pushing the event into the Swing event queue. Dispatching the VMEvent to the Swing event loop simplifies thread synchronization between virtual machine events and GUI events.

When the Swing event loop gets a concrete `VMEvent` out of its event queue it calls the `run` method. This will run the code for the particular event. There is one concrete `VMEvent` for each type of virtual machine event.

5.2 Package Architecture

TeaBag contains over 150 classes. These classes are grouped together into packages. Grouping classes together into packages has two benefits. Firstly, it groups similar classes together. This makes it easier to find classes related to a particular topic. For example, the breakpoint package contains all of the classes related to the breakpoint data structures. The second advantage is that it promotes decoupling. While it is perfectly legal, and sometimes desirable, to have classes in different packages refer to each other, it is not desirable. Breaking classes up into packages helps the programmer realize where the boundaries of concrete references should be. So classes that refer to each other across packages should use interfaces to provide greater decoupling. In TeaBag many of the cross package references use interfaces. This is not true in all cases in TeaBag. For example, many objects in the GUI package directly reference objects in the data structures packages. However, the objects in the data structure packages only have references to objects in the GUI package through interfaces.

5.2.1 Inter-Package Communication

Information in TeaBag needs to be communicated among objects within one package as well as between objects in different packages. Within TeaBag there are two ways that information is communicated from object A to object B. The first way is by giving object A a reference to object B and then having object A call some method on object B to communicate the information. This is the simplest form of communication. However, this imposes high coupling between objects A and B. This is especially not desirable if objects A and B are in different packages. For example, if object A is in the breakpoints package (§5.2.2) and object B is in the GUI package (§5.2.2) it is not desirable to have breakpoints know anything about a GUI. In fact a breakpoint is a concept that is completely independent of a GUI. Yet the GUI needs to know about breakpoints so that it can display them to the user.

The second way of communicating information from object A to object B in TeaBag solves this problem. This communication is with events. TeaBag primarily uses events for internal communication among top level packages. The class structure for an event is shown in figure 5.3. There are two classes and one interface involved in an event. The *Subject* class is the one that needs to communicate information to other objects. The objects receiving the information are

*ConcreteListener*s. The *ConcreteListener*s implement the *Listener* interface. The *Subject* just needs to keep a list of *Listener*s. It does not need to know what the concrete type of the *Listener* is. If an object wishes to be notified of events from the *Subject* it can register itself using the `addListener` method. When the *Subject* needs to notify listeners of an event it fires the event by calling the appropriate method on each of the *Listener*s. Since the *Subject* only knows what the interface for the *Listener* is, it does not need to know about the existence of any other packages. So now if object A is in the breakpoint package and object B is in the GUI package object A will not need to know about the existence of a GUI package to communicate information to object B. Rather object B will register itself with object A via the `addListener` method. Object A will only know that object B implements the *Listener* interface. It will not know the real type of object B. When object A needs to communicate information to object B, object A will fire the appropriate event. Of course for object B to register itself with object A it will need to know the real type of object A. This though is not a problem since object B needs to know this for some other purpose since it is interested in information from object A already. In this example, object B might need to display information about breakpoints. For object B to do this it must know what a breakpoint is. So it must know about the existence of an object A. However, since breakpoints are independent of a GUI object A should not have to know about an object B.

Using events for internal communication has two benefits. The first, as shown already, is that it decouples objects in different packages. The second benefit is that it makes adding new behaviors into the system easier. If an object needs to perform an action when a particular event occurs all it needs to do is implement

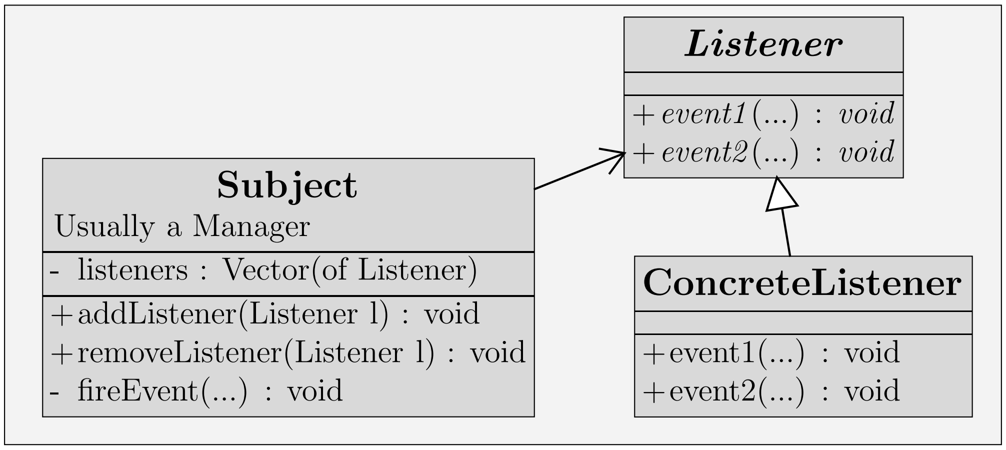


Figure 5.3: Class Structure of Events

the appropriate interface, register itself with the subject, and respond to the event. Without this kind of structure adding new behavior requires making changes to the Subject and coupling it to the new object. This is much more error prone than just registering the new object.

5.2.2 Packages Overview

The classes for TeaBag are broken up into many packages. A number of the packages are for different kinds of data structures that map to data structures on the virtual machine. The virtual machine has data structures for terms, computations, etc... TeaBag also has data structures for these. However, in TeaBag the data structures tend to be simpler than in the virtual machine. All TeaBag needs to do, is be able to display the structures and communicate information about the data structures to the virtual machine. The virtual machine performs real work on its data structures and communicates this information to TeaBag. Thus, the purpose of these data structures in TeaBag is for visualization and in the virtual machine it is for computing a result.

Many of the packages in TeaBag have a manager class. The manager classes are singletons [32] that allow other packages to get information from the package. Typically, the manager provides events that other objects can register for. The manager will then notify the listeners when the data structures have changed. For example, the `BreakpointManager` will notify its listeners when ever a breakpoint has been added or removed.

Breakpoints

The breakpoint package contains data structures for breakpoints. It contains a `BreakpointManager` singleton. Objects can register themselves with this manager to be informed when a breakpoint has been added or removed.

Computations

The computation package contains a data structure that maps to a specific computation in the virtual machine. Once again this package has a `ComputationManager` singleton. Objects can register themselves with this manager to be informed when computations have been created, destroyed, or changed.

Modules

The modules package contains a data structure that maps to a specific module in the virtual machine. This package also has a manager, `ModuleManager`. There are no events to register for with this manager.

Perspectives

A perspective in TeaBag is a collection of panels arranged in a particular order (§4.3.3). This package contains data structures for perspectives. It also has a manager that controls when perspectives are switched, loaded, and saved.

Settings

The settings package contains the global settings object for TeaBag. This package also has a manager that controls access to the settings object. Objects can register themselves with the manager to be notified when the settings are being restored, saved, or changed. This makes it easy for an object to save some settings. It just needs to register itself with the manager.

Trace

The trace package contains data structures for a trace. 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 browser. 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 [32] among its sub-terms via a listener. Then, when a sub-term 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 un-marshaling 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.

Commands

The command package provides a facade [32] for debug commands. This simplifies the process of performing debug commands like adding a breakpoint. This package will create the necessary data structures, register them appropriately, and inform the virtual machine as necessary.

Exception

The exception package contains two exception handlers. The first exception handler is used to display exception in TeaBag to the user. The second exception handler is used to display exceptions in the virtual machine to the user.

GUI

The GUI package contains all objects for displaying the GUI. Since the GUI for TeaBag has numerous classes the GUI package is further divided up into sub-packages.

The first of these sub-packages is for panels. Each debugging element within TeaBag is in its own panel. These panels derive from the `DebugPanel` class. The user can move the panels around on the screen. Each panel has a title bar. The title bar has the name of the element, the elements icon, a dock/undock button, and a close button. For example, figure 5.4 contains the title bar for breakpoints. The user can move the panel around on the screen by dragging its title bar. A panel can be moved next to other panels by dragging it to the edge of another panel. A panel can also be added to the same space as another panel by dropping it on the title bar for the other panel. When two or more panels occupy the same space the user can select which panel they want to view with the tabs at the bottom of the panels. The size of each of the panels can be adjusted by moving the slider bars placed around the panels. The user can chose to not view the panel by clicking on the close button in the panels title bar. The user can also undock the panel by clicking on the undock button in the title bar. Doing this causes the panel to be put in its own window. Panels can also be shown by selecting them in the view menu item. All panels that derive from `DebugPanel` inherit this panel behavior. Thus adding a new panel to TeaBag is just a matter of deriving from `DebugPanel`.

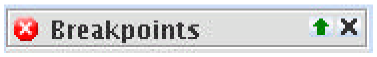


Figure 5.4: Title Bar Example

There is also a sub-package for drag-and-drop support. All classes for supporting drag-and-drop are in this package.

The next sub-package is for pop-up windows. This package contains all of the windows that are displayed when the user right clicks on an object in TeaBag. All of the pop-up windows derive from a common base class `PopupBase`. Deriving all of the pop-up windows from a common base classes allows TeaBag to treat them all uniformly. This makes it much easier to display different pop-ups for different types of tree nodes.

The fourth sub-package is stepview. This package provides viewers for displaying deterministic and non-deterministic steps. The viewers can be used as decorators [32] with the runtime term viewer and the trace browser.

The fifth sub-package is for table support. This package contains classes used for displaying information in tables.

The final sub-package is for tree support. Most data in TeaBag is displayed in Trees. All tree nodes are derived from `TreeNodeBase`. There is a custom tree renderer that works with tree nodes that derive from `TreeNodeBase`. This tree renderer is able to add icons, borders, and backgrounds to the tree nodes. These tree nodes can be displayed using either a `JTree`, which is provided with Swing, or with a `NaturalTree`. Both the `JTree` and `NaturalTree` can work with the same tree node data structure and tree renderer. Thus the same tree nodes can be displayed with either tree class.

Packet

The packet package contains a data structure for a packet. A packet is just a set of string fields separated by the '|' character. Thus the only kind of data that can be written to and read from a packet is a string. The `Packet` class provides convenient methods for reading and writing primitive types to the packet. However, not all data is primitive types. Thus, this package also contains packet readers and packet writers for reading and writing other data types to the packet. For example, the `PacketWriter` class has a method for writing a term to the packet.

Term

The term package contains a data structure for terms. Each term in TeaBag maps to a term in the virtual machine. It is possible that the term in TeaBag has a longer life than the term in the virtual machine. Thus, it is not required that the term exist in the virtual machine for it to exist in TeaBag. The only requirement is that the term existed at some point for some virtual machine. In TeaBag a term has an identification, root symbol, arguments which are terms, and a created time. The identification for the term identifies a term in then virtual machine if the created time for the term is after the last time the virtual machine was started.

5.3 Threads

TeaBag has four threads running. It has the Swing event thread and one thread for each of the sockets used to communicate with the virtual machine (§5.1). The Swing event thread is used to respond to GUI generated events. This thread is created and managed by Swing classes. To respond to data coming across the sockets there is one thread for each socket. Each of these threads perform a blocking read on their socket. This causes the thread to sleep if there is no data available on the socket. When data is available on the socket the thread wakes up and handles the information from the socket. One of the threads reads from the data socket. When this thread parses an entire event from the socket it dispatches it to the GUI event thread for processing. TeaBag also has a thread for reading from the command socket. This thread just reads characters from the command socket and then appends those characters to the end of the consol window. This way the user sees the output of the virtual machine. The final thread reads data from the exception socket. Error messages in the virtual machine are sent on this socket. So when this thread is able to read data on this socket it invokes an exception handler to display the error message.

5.4 Socket Interface Specification

The virtual machine and TeaBag communicate over sockets. This communication follows the protocol defined in this section. Thus any virtual machine that implements this socket interface could be used with TeaBag. The data sent over the sockets follow a specific format. There are three sockets used for communication (§5.1). The data format is specific to each socket and to each data flow direction on that socket. Thus there are six different socket interfaces. They are

1. Command Socket: From virtual machine to TeaBag
2. Command Socket: From TeaBag to virtual machine

3. Data Socket: From virtual machine to TeaBag
4. Data Socket: From TeaBag to virtual machine
5. Exception Socket: From virtual machine to TeaBag
6. Exception Socket: Form TeaBag to virtual machine

Before discussing each of the above interfaces some common concepts are specified. Many of these interfaces have parts of their specification in common. These concepts will be developed before giving the specification for each of the socket interfaces.

5.4.1 Packet format

Many of the socket specifications require that all or part of the data be sent in packets. A packet is just a record of fields. Each packet is a string of ASCII characters. The start of a packet is denoted by a '{' character. A packet is ended by a '}' character. Each packet contains fields. Each field is separated by a '|' character. Each field in the packet can be either a string representing a value or another packet. Thus, a packet is defined recursively. Also, each packet for a particular type or event does not have to be a fixed length. Most packets have a title in the first field that identifies what type of packet it is.

Packet formats will be presented with the following conventions:

- Optional fields are displayed in *italics*
- Text that must be typed in as shown is displayed in `true type`

- Text that must be replaced with an appropriate value is surrounded by < and >.
- Multiple options for a field are separated by a '/'
- Fields that are packets are denoted by {name:Type} where name is used to identify the field and Type says which type of packet it is. If the type of the packet is a list then the type is denoted as [Type]Packet. Thus [Int]Packet means a packet where all of the fields are of type Int.
- Each field is labeled with its field number in superscript at the start of the field.

For example, consider the following packet.

$$\{^1\texttt{text/bigText}|^2\text{<textId>}|^3\text{<text>}|^4\text{<}\textit{title}\text{>}\}$$

The above packet would be read as a packet that has either the string "text" or "bigText" in the first field. The second field has the textId attribute field in. The third field contains text attribute field in. Finally, if there is a fourth field then it has the title attribute.

5.4.2 Virtual Machine Identifiers

Since the virtual machine and TeaBag communicate over sockets they need a way to refer to specific data structures without using memory pointers or some language specific referencing. This is done with TeaBag identifiers. Each data structure in the virtual machine that is communicated to TeaBag must be given a unique identifier if it is possible that TeaBag may want to refer to it latter on. This applies to two different data structures in the virtual machine. The first one is a term. Each term sent to TeaBag must have a unique identifier. This identifier must be an integer. The second data structure that requires a unique identifier is computations. Each computation that is sent to TeaBag must have a unique identifier that is a string. These unique identifiers must be unique to one session of the virtual machine. So the virtual machine may reuse identifiers from when it previously ran. In TeaBag all computations are removed when the virtual machine is shutdown. Thus it cannot refer to a computation that was not created by the currently running virtual machine. When a new term is created in TeaBag the time of its creation is recorded. Each time the virtual machine is started the start time is also record. To determine if a term was created by the current virtual machine its creation time must come after the last time the virtual machine was started.

5.4.3 Common Packet Formats

Since packets can be nested, there is a common packet format for the data structures that are included in many of the data packets. The common data structures are modules, terms, computations, and non-deterministic information.

Packet Format for Modules

When a module is sent in a packet it uses the following packet structure.

$$\{^{1}\texttt{Module}|^{2}\text{<moduleName>}|^{3}\text{<isCompiled>}|^{4}\text{<sourceFile>}\}$$

1. *title*: The first field must contain `Module` to identify this as a module packet
2. *moduleName*: The name of the module like IntModule.
3. *isCompiled*: `true` if the module has been compiled and `false` otherwise.
4. *sourceFile*: The name of the source file for this module. If there is no source file this field is empty. If there is no path given then the current directory is used.

Packet Format for Terms

When a term is sent in a packet it uses the following packet structure.

$$\{^{1}\texttt{Term/empty}|^{2}\text{<termId>}|^{3}\text{<watchId>}|^{4}\text{<hasBreakpoint>}|^{5}\text{<rootSymbol>}$$
$$|^{6}\text{<representation>}|^{7}\{\text{module:ModulePacket}\}$$
$$|^{8}\{arg_1\textit{:TermPacket}\}|...|^{8+n}\{arg_n\textit{:TermPacket}\}\}$$

1. *title*: The first field must contain either `Term` to identify this as a term packet or `empty` to identify that there is no term e.g. a null pointer. If this field is `empty` then all of the other fields are ignored.
2. *termId*: The unique id of the term. This must be an integer.
3. *watchId*: The termId that is being watched. In the virtual machine terms are rewritten with new terms. When this happens the root of the resulting term will have a different identifier from the term it replaced. TeaBag may have requested that a watch (§5.4.7) be set on the term that was replaced. This watch information needs to be passed on to the replacement. This is

done with a watchId. The watchId is the original identifier of the term that a wach was set on. The watchId and the termId will be the same until the term is rewritten.

4. *hasBreakpoint*: `true` if this term has a breakpoint and `false` otherwise.

5. *rootSymbol*: The root symbol of the term. For example, the root symbol of `f (g 1) 2` is `f`.

6. *representation*: An alternate representation for displaying the term.

7. *arg*$_{1..n}$: The arguments of the term for this packet where n is the arity of the rootSymbol. If this term has no arguments then these fields are not included.

Packet Format for Computation

When a computation is sent in a packet it has the following structure.

$$\{^1\texttt{Computation}|^2\text{<compId>}|^3\text{<compState>}|^4\{\text{term:TermPacket}\}$$
$$|^5\text{<creatorId>}|^6\text{<clientId>}\}$$

1. *title*: The first field must contain `Computation` to identify this as a computation packet

2. *compId*: The unique identifier of the computation. This can be any string.

3. *compState*: The state of the computation. This must be an integer between 0 and 7 inclusive where the meanings of the integers are as follows.

   0: Active

1: Waiting

2: Abandoned

3: Success

4: Failed

5: Residuating

6: Flounder

7: Paused

4. *term*: The term this computation is working on

5. *creatorId*: The computation identifier of the computation responsible for creating the computation with an identifier matching compId.

6. *clientId*: The computation identifier of the computation that is currently the "parent" of this computation. The parent computation is the one that is logically this computation's parent in the search space.

Non-Deterministic Information

Both the breakpoint and non-deterministic trace step events (§5.4.5) have a number of common pieces of information. This common data is put into its own packet to make marshaling and un-marshaling the data easier.

{^{1}`NDInfo`|^2<lhs_var>|^3<lhsId>|^4<{lhsPos:[Int]Packet}>
|^5<{rhs_bindings:[TermPacket]Packet}>|^6<{bindingIds:[Int]Packet}>|^{7}stepType}

- *title*: The first field must be `NDInfo`.

- *lhs_var*: The left hand side of the source code for non-deterministic choice steps or the name of the variable being bound for narrowing steps.

- *lhsPos*: Source code positions for non-deterministic choice options. This field is a packet that has integers in every field. The length of this packet must be a multiple of two. It is OK if the packet is empty but it must be in this field so that this field can not be empty. The $(i * 2)^{th}$ entry in this packet is the character offset into the source code for the start of the code for the i^{th} entry in replacements. The $(i * 2 + 1)^{th}$ entry in this packet is the character offset into the source code for the end of the code for the i^{th} entry in the replacements.

- *rhs_bindings*: The right hand side of the source code for non-deterministic choice steps and the bindings of the variable being bound for narrowing steps. This field is a packet that contains only strings. The i^{th} entry in this packet corresponds to the i^{th} entry in the replacements field. The text for the right hand sides and bindings are displayed to the user in the same format they are received in.

- *bindingIds*: The term identifiers of the bindings of variables for narrowing steps. When the non-deterministic step is a non-deterministic choice step then the packet for this field is empty. The i^{th} entry in this packet is the term identifier in the i^{th} entry of the replacements that is the root of the binding for the variable being bound by the narrowing step.

Packet Format for Redex Pattern

The redex pattern is the constructors in a redex that were used to identify which rewrite rule to apply.

$$\{^1\text{<}\{\text{pos}_1\text{:[Int]Packet}\}\text{>}|...|\{^n\text{<}\{\text{pos}_n\text{:[Int]Packet}\}\text{>}\}$$

1. $pos_{1..n}$: Each of the terms in the redex pattern is described with its occurrence in the redex. This is the term's position in the redex and not in the top level term. If the only term in redex pattern is the root of the redex then this list is empty. Each occurrence is a list of integers. The following algorithm demonstrates how the term, `t`, at occurrence, `o`, is obtained from the `redex`

   ```
   Term t = redex
   for (i = 0; i < o.length; i++)
      t = t.getArgument(o[i])
   ```

5.4.4 Command Socket Interface

The command socket is meant to replace the standard in and out streams in the virtual machine. Thus, the command socket is designed to look like a console to the virtual machine.

Command Socket: From Virtual Machine to TeaBag

The data on the command socket going from the virtual machine to TeaBag is the output of the virtual machine. This is anything that is written to standard out. Thus, there is no special format for this data. All characters, with a few exceptions, will be displayed in the console as virtual machine output in TeaBag. Some unwanted command prompt characters, '>', are suppressed by TeaBag.

Command Socket: From TeaBag to Virtual Machine

The data on the command socket going from TeaBag to the virtual machine is input to the virtual machine. Thus, there are commands that a user would type into the virtual machine if the virtual machine was running by itself. All commands that the user types into the console in TeaBag are sent to the virtual machine over this socket just as the user typed them in. Also, TeaBag uses this socket to send some debug commands to the virtual machine. Since the debug commands for the virtual machine have the same format as the normal user commands they can be treated as user input. So this socket can be redirected to the standard in stream in the virtual machine.

Debug commands can be either synchronous or asynchronous. Debug commands that are synchronous are ones that can only be sent when the command prompt is available. That is, they can only be sent when the virtual machine is expecting the user to type in a command. Asynchronous commands can be sent to the virtual machine at any time. Thus they might be sent when the virtual machine is not expecting any input from the console. All of the commands sent over the command socket are synchronous commands. This allows the virtual machine to treat this socket as standard in. The list of debug commands is in section 5.4.7.

5.4.5 Data Socket Interface

Data Socket: From Virtual Machine to TeaBag

The virtual machine sends debug events to TeaBag over this socket. Each event is encoded into a packet. The following events are supported. The notation used for describing these packets is explained in section 5.4.1.

Breakpoint Event A breakpoint event is sent from the virtual machine whenever the virtual machine halts for a breakpoint set on a function or term. There are two kinds of breakpoint events. The first one is when the breakpoint occurs on an ordinary rewrite step. The second is when the when the breakpoint occurs on a non-deterministic step. This event has the following format:

$$\{^{1}\texttt{Break/NDBreak}|^{2}\{\text{redex:TermPacket}\}|^{3}\{\text{replacements:[TermPacket]Packet}\}$$
$$|^{4}\{\text{context:TermPacket}\}|^{5}\{\text{redexPattern:RedexPatternPacket}\}$$
$$|^{6}\text{<sourceFile>}|^{7}\{\text{ndInfo:NDInfoPacket}\}|^{8}\text{<compId>}\}$$

- *title*: This must be `Break` or `NDBreak`. `Break` indicates a deterministic rewrite step. `NDBreak` indicates a non-deterministic step.
- *redex*: The term replaced with the rewrite step.
- *replacements*: The possible replacement terms of the rewrite or narrowing step. If the the title is `Break` then then only the first replacemnt is used. If the title is `NDBreak` then all replacements are used.
- *context*: The context is the global term the replacement is a part of. The context is only sent when the debug command `:sendContext on` has been

issued. If this command has not been sent to the virtual machine then this field is blank. Since it is possible for terms to get large sending big terms could slow down the marshaling and unmarshaling of data between FLVM and TeaBag. Thus, this option can be turned off if the user is not planning on using it. If there are multiple replacements then this is the context of the first replacement.

- *redexPattern*: This field contains the occurences of the terms in the redex that make up the redex pattern.
- *sourceFile*: If the operation for the narrowing step has an associated source file then this field contains the path and file name for the source, otherwise this field is blank.
- *ndInfo*: The non-deterministic information data structure.
- *compId*: The computation identifier of the computation that is computing the redex.

Watch Event Traces are implemented in TeaBag by setting a watch on a term. Whenever that term or one of its subterms is rewritten a watch event is fired to the debugger letting it know which term was updated. TeaBag records all of the steps taken on a watched term to create a trace. A watch event has the following format:

$$\{^{1}\texttt{Watch}|^{2}\text{<sourceFile>}|^{3}\text{<startPos>}|^{4}\text{<endPos>}|^{5}\{\text{watchTerm:TermPacket}\}$$
$$|^{6}<\{\text{redexPattern:RedexPatternPacket}\}\ \}$$

- *title*: This must be `Watch`
- *sourceFile*: If the rewrite step that caused the watch event to be fired was from an operation that has source code available then this is the name of the source code file, otherwise this field is blank.
- *startPos*: The character offset in the source file for the start of the code that performed the rewrite step. If there is no file position then this field is 0.
- *endPos*:The character offset in the source file for the end of the code that performed the rewrite step. If there is no file position then this field is 0.
- *watchTerm*: The term that the watch was set on. This needs to be the term after the rewrite step has occurred. Also, the watchId field of the term packet for this term must be filled in even if the watchId is the same as the term identification.
- *redexPattern*: The occurrences of the terms that make up the redex pattern.

Evaluation Event When TeaBag asks the virtual machine to eagerly evaluate a term it will wait for an evaluation event. This event gives TeaBag the result of the eager evaluation. The format of this event is as follows:

$$\{^1\texttt{Eval}|^2\text{<terminationCause>}|^3\{\text{result:TermPacket}\}\}$$

- *title*: This field must be `Eval`.
- *terminationCause*: The cause for firing the evaluation event. The virtual machine will only eagerly evaluate terms up to non-deterministic steps. This field must be either 0, 1, or 2 where the integers have the following meaning.

0: The term evaluated to normal form

1: A non-deterministic narrow step was taken

2: A non-deterministic choice step was taken.

- *result*: The result of the evaluation. The virtual machine must hold onto this result until it gets the next debug command since that debug command may ask for the result to replace the term that evaluated to the result.

Result Event To let the user browse the result of a computation in a tree structure it must be sent as an event rather than over standard out. This is one area where just redirecting standard output to the command socket does not work. When the virtual machine has a result to display it fires this event. The result event has the following format:

$$\{^1\texttt{Result}|^2\{\text{result:TermPacket}\}|^3\text{<answer>}\}$$

- *title*: This field must be `Result`.
- *result*: The term that is the result of the computation.
- *answer*: The binding of terms to variables that was used to obtain this result. This can be any string. It will be displayed in whatever format it is received in.

Computation Register Event When a new computation is registered, a computation registration event is sent to TeaBag if the virtual machine has not received a `sendComputationInfo off` command. A computation is registered when it is designated to work on a term. Since numerous computations can be registered and unregistered the user can turn these event off to improve the performance of TeaBag if they do not care about computation information. The packet structure for a computation register event is as follows.

$$\{^{1}\texttt{CompReg}|^{2}\text{comp:ComputationPacket}\}$$

- *title*: This field must be `CompReg`.
- *comp*: The computation being registered.

Computation Unregister Event When a computation is unregistered a computation unregister event is sent to TeaBag if TeaBag has not sent the command `sendComputationInfo off`. A computation is unregistered when it is designed to not perform any more work. The format of this command is as follows.

$$\{^{1}\texttt{CompUnreg}|^{2}\text{comp:ComputationPacket}\}$$

- *title*: This field must be `CompUnreg`.
- *comp*: The computation being unregistered.

Computation Change Event This event notifies TeaBag that a computation has changed. Once again, this event is only sent if the virtual machine has not received the command `sendComputationInfo off`. There are two cases where this event gets fired. The first one is when the state of the computation changes. The states of a computation are listed in section 5.4.3. The second time this event is sent is when the virtual machine receives the command `:get computation`. See section 5.4.7 for a description of this command. The format of the packet for this event is as follows:

$$\{^1\texttt{CompChange}|^2\{\text{comp:ComputationPacket}\}\}$$

- *title*: This field must be `CompChange`.
- *comp*: The computation that changed.

Start Non-Deterministic Step Event When the virtual machine makes a non-deterministic step it needs to let TeaBag know what computations where created for that step. The virtual machine does this by sending TeaBag an event saying that it is about to start a non-deterministic step. This event must be sent before any computations are created for the step. Once again, this event is only sent if the virtual machine has not received the command `sendComputationInfo off`.

$$\{^1\texttt{StartNDStep}|^2\texttt{Narrow/Choice}\}$$

- *title*: This field must be `StartNDStep`.
- *type*: The type of non-deterministic step.

End Non-Deterministic Step Event When all of the computations for a non-deterministic step event have been created the virtual machine sends an end non-deterministic step event. This lets TeaBag know when to stop grouping computation register events together for one non-deterministic step. As in all other events for computations this event is only sent if the virtual machine has not received the command `sendComputationInfo off`. There are three packet formats for this command. They are as follows.

$\{^1$`EndNDStep`$|^2$`Narrow/Choice`$|^3\{$redex:TermPacket$\}\}$

$\{^1$`EndNDStep`$|^2$`Narrow`$|^3\{$redex:TermPacket$\}|^4$<variableName>

$|^5\{$binding$_1$:TermPacket$\}|...|^{5+n}\{$binding$_n$:TermPacket$\}\}$

$\{^1$`EndNDStep`$|^2$`Choice`$|^3\{$redex:TermPacket$\}|^4$<lhs>$|^5$<rhs$_1$>$|^6...|^{6+n}$<rhs$_n$>$\}$

- *title*: This field must be `EndNDStep`.
- *type*: The type of non-deterministic step. This must be either `Narrow` or `Choice`.
- *redex*: The term replaced with the rewrite step.
- *variableName*: The name of the source code variable that was instantiated.
- *binding*$_{1...n}$: The bindings for the variable. When a variable is instantiated many times there are multiple possible substitutions for it. These fields are all of those substitutions.
- *lhs*: The left hand side in the source code that matches the redex.
- *rhs*$_{1...n}$: The right hand sides in the source code that are the possible replacements for the left hand side.

The order of binding$_{1...n}$ and rhs$_{1...n}$ must be the same as the order the corrisponding computations were sent to TeaBag via the computation register event between the start non-determistic step event and the end non-detereministic step event.

Module Loaded Event Whenever the virtual machine loads a module a module loaded event is sent to TeaBag. The packet format of this event is as follows:

$$\{^1\texttt{modLoaded}|^2\{\text{module:ModulePacket}\}\}$$

- *title*: This field must be `modLoaded`.
- *module*: The module that was loaded.

Module Unloaded Event Whenever the virtual machine unloads a module, a module unloaded event is sent to TeaBag. The packet format of this event is as follows:

$$\{^1\texttt{modUnloaded}|^2\{\text{module:ModulePacket}\}\}$$

- *title*: This field must be `modUnloaded`.
- *module*: The module that was unloaded.

Non-Deterministic Trace Step Event When a term, or one of its subterms, that has a watch set on it is rewritten from a non-deterministic step a trace non-deterministic step event is fired. This event is also fired when a variable is bound. In this special case the narrex has just one replacement. The packet format of this event is as follows:

$$\{^1\texttt{traceNDStep}|^2\{\text{narrex:TermPacket}\}|^3\{\text{replacements:[TermPacket]Packet}\}$$
$$|^4\text{<ndTermId>}|^5\{\text{ndInfo:NDInfoPacket}\}\}$$

- *title*: The first field must be `traceNDStep`.
- *narrex*: The term with a watch set on it that caused a non-deterministic step to occur. Even if one of the subterms of the term with the watch caused the non-deterministic step to occur the term with the watch set on it is still put in this field.
- *ndTermId*: The term identifier of the term that caused the non-deterministic step. This term does not need to have a watch set on it.
- *ndInfo*: The non-deterministic information data structure.

Data Socket: From TeaBag to Virtual Machine

The asynchronous debug commands are sent over this socket. See section 5.4.7 for a description of the debug commands. The debug commands sent over this socket can come at any time. They do not have to only come when the virtual machine is expecting input from the user. So the virtual machine needs to have a thread dedicated to listening for data on this socket.

5.4.6 Exception Socket Interface

Exception Socket: From Virtual Machine to TeaBag

All error messages that the virtual machine wants displayed in TeaBag are sent over this socket as plain text. TeaBag will display this text to the user in a window. All text received on this socket from the end of the last error message displayed until the user closes the error window will be displaed in the error window. This allows the virtual machine to redirect standard error to this socket. Then all error messages written to standard error in the virtual machine will be displayed as an error message to the user in TeaBag. Also, in Java calling `printStackTrace()` on an exception prints the stack trace to standard error. This makes it easy to display the stack trace in the error message in TeaBag. It is possible that multiple virtual machine errors will be displayed together as one error in TeaBag since TeaBag just streams the data on this socket to a text box in the error window.

Exception Socket: From TeaBag to Virtual Machine

No data is sent over this part of the socket.

5.4.7 Debug Commands

Commands in Curry have the form:

:Name *arg1 ... argN*

All commands start with the ':' character followed by the name of the command. The commands can have arguments separated by spaces. The debug commands follow the same format.

When describing the debug commands the following notation will be used:

- Optional parts are displayed in *italics*
- Text that must be typed in as shown is displayed in `true type`
- Text that must be replaced with an appropriate value is surrounded by < and >.

- Multiple options for an argument are separated by a '/'
- Each command will be marked as [Synchronous] or [Asynchronous]. A command that is synchronous is one that is sent over the command socket and can be treated as a command typed in by the user. The asynchronous commands are sent over the data socket.

`:breakpoint function` <functionName> [Asynchronous]

> This command toggles the breakpoint on the operation that has a name matching functionName. If no operation with a name matching functionName exists then an error message is sent on the exception socket.

`:breakpoint term` <termId> [Asynchronous]

> This commands toggles the breakpoint on the term with an identification matching termId. If no such term exists then an error message is sent over the exception socket. The termId must be an integer.

`:step` [Synchronous]

> This command causes the virtual machine to perform the next rewrite step and then halt. This command can only be issued when the virtual machine has halted for a breakpoint. When the virtual machine halts after performing one rewrite step it should fire a breakpoint event to the debugger (§5.4.5).

`:run` [Synchronous]

This commands causes the virtual machine to run until a solution is found or a breakpoint is encountered. Once again this command can only be issued when the virtual machine has halted for a breakpoint.

`:eval` <term> [Synchronous]

The evaluate command tells the virtual machine to evaluate the term to head normal form or until a non-deterministic step is taken. Once a result is obtained the virtual machine fires an evaluation event (§5.4.5). Even if the evaluation stoped because of a non-deterministic step the evaluation event is still fired with the term that caused the non-deterministic step. The only times that an evaluation event is not sent is if there is an error during evalution or the `:cancel` command is received by the virtual machine.

The term given to this command to evaluate is a packet. The identifier of the term is not used so that the virtual machine does not have to keep all terms sent to TeaBag live in case the virtual machine is asked to evaluate a term. Also, not using the identification of the term allows the user to perform a trace on a non-terminating computation, kill the virtual machine, start a new virtual machine, and evaluate a term in the trace. So the term being evaluated does not have to have existed in the virtual machine that is being asked to evaluate it. If an error occured during evaluation then the error message is sent over the exception socket. The packet for the term passed to this command does not follow the term packet format presented in section 5.4.3. The format for this modified term packet is as follows:

$$\{^1\text{<rootSymbol>}|^2\{\text{arg}_1\text{:ModifiedTermPacket}\}|\ldots$$
$$|^{1+n}\{\text{arg}_n\text{:ModifiedTermPacket}\}\}$$

- *rootSymbol*: The root symbol of this term
- *arg*$_{1\ldots n}$: The arguments of the term. These are in this modified term packet format.

Since the `:eval` command is synchronous the virtual machine will not be asked to eagerly evaluate a term when it is in the middle of a computation.

`:cancelEval` [Asynchronous]

This command cancels the eager evaluation of a term. When the virtual machine receives this command it does not need to send an evaluation event with the result of the eager evaluation.

`:get computation` <compId> [Asynchronous]

This causes the virtual machine to fire a compuation changed event for the computation with an identifier matching compId to TeaBag.

`:replace` <redexId> <replacementId> [Synchronous]

This replaces the term with an identification matching redexId with the term that has replacementId for an identification. The replacement term is always the result of the `:eval` command.

`:removeWatch` <termId> [Asynchronous]

> This removes the watch from the term with an identification matching termId.

`:addWatch` <termId> *watchId* [Asynchronous]

> This adds a watch to the term with an identification matching termId. If watchId is not specified then termId is used for the initial watch identification. Otherwise watchId is used for the initial watch identification.

`:kill` <compId> [Asynchronous]

> This designates the computation with an identification matching compId to not perform any more work.

`:pauseComputation` <compId> [Asynchronous]

> This tells the virtual machine to not execute any narrowing steps for the computation with an identification matching compId. The computation may be activated again so the virtual machine cannot get rid of it completely like it can with the `:kill` command.

`:activateComputation` <compId> [Asynchronous]

> This tells the virtual machine to start executing narrowing steps for the computation with an identifier that matches compId.

`:eagerEvalIf` <`on/off`> [Asynchronous]

> This command turns the eager evaluation of ifs feature on and off. When eager evaluation of ifs is turned on rewriting an if term must evaluate its condition to head normal form and then replace the if term with either the then or else clause. This process must be repeated until the term is not rooted with an `if then else` term. This whole process should appear as a single step. So if this affects a trace then only one watch event should be fired. Also, the virtual machine should not stop for any breakpoints when evaluating the if conditions.

`:showNDPolicy` <`0/1/2/3/`> [Asynchronous]

> This sets the global non-deterministic breakpoint. When the virtual machine breaks for a non-deterministic step it does not send any events to the debugger. Rather, once the non-deterministic step has occurred it waits for TeaBag to give it a command just like it would after sending a break event. The parameter for this command is one of four possibilities.
>
> 0: Never break for non-determinisitic steps
>
> 1: Break for both non-deterministic choice and narrowing non-deterministic steps.
>
> 2: Break for all non-deterministic narrowing steps but not for non-deterministic choice steps.

3: Break for all non-deterministic choice steps but not for non-deterministic narrowing steps.

`:sendComputationInfo` <`on/off`> [Asynchronous]

This turns sending computation information on or off. By default the virtual machine should send computation information. This command can turn this behavior on and off. Sending computation information means firing computation registered, unregistered, and changed events (§5.4.5).

`:sendContext` <`on/off`> [Asynchronous]

This commands tells the virtual machine if it should send context information in the term packets. If this command is invoked with the parameter `on` then the virtual machine should include context information in the term packets. If the parameter is `off` then it should not include the context in the term packets.

Chapter 6

Examples

This section presents some examples of how TeaBag can be used to debug Curry programs. In functional logic languages there are three kinds of errors that can occur.

1. Wrong Answer: A wrong answer is when a program returns an unexpected, but legal, result.
2. Missing Answer: A missing answer is when the program is unable to return any result and it halts.
3. Non-Terminating Error: A non-terminating error is when the program does not terminate.

The first example, sorting (§6.1), will demonstrate how TeaBag can be used to detect all three of these kinds of errors with a small Curry program. The next example, a simple lambda calculus interpreter (§6.2), will demonstrate how TeaBag can be used on larger programs. Finally, the last example, Dutch national flag problem (§6.3), will show how TeaBag can be used in applications that have a non-trivial search space.

6.1 Sorting Example

The following example, taken from the Hat Tutorial [71], demonstrates three possible bugs. It demonstrates a program that terminates with a run-time error, a program that does not terminate, and a program that terminates with an incorrect output.

```
sort :: [Int] -> [Int]
sort(x:xs) = insert x (sort xs)
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) = if x <= y then x:ys
                  else y : insert x (y:ys)
```

When `sort[4,3,2,1,6]` is run the result is `No more solutions`. The result should have been the sorted list, `[1,2,3,4,6]`. This is an example of a program that halts with a run-time error. To find this bug we traced the evaluation of `sort[4,3,2,1,6]`. Figure 6.1 shows steps six and seven of the trace. From from step six to step seven `sort []` was rewritten to `Fail`. This indicates that there was no rule for `sort []`. So we added `sort[] = []` to the program.

We then ran `sort[4,3,2,1,6]` again. This time the computation did not terminate. So we killed and restarted the FLVM within the debugger and set a breakpoint on `sort`. Next, we ran `sort[4,3,2,1,6]` again. When the breakpoint for `sort` was hit we added a trace to the term rooted with `sort`, removed the breakpoint on `sort`, and ran the FLVM. We then killed the FLVM from the debugger. The trace for `sort` shows all of the steps up to the point where we killed the FLVM for `sort[4,3,2,1,6]`. When stepping through this trace we noticed that

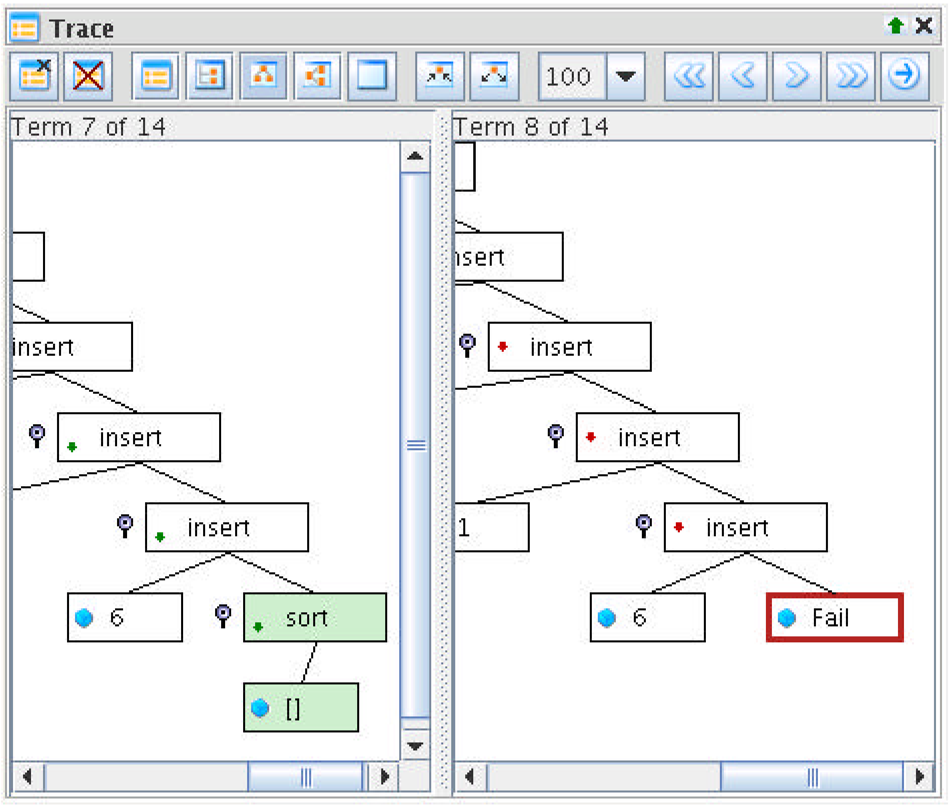


Figure 6.1: `sort []` rewritten to `Fail`

it inserted 2 twice into the list. The first time was when it rewrote insert(2,[1]) to 1:insert(2,[1]):[]. It should have rewritten insert(2,[1]) to [1,2]. We changed the recursive call insert x (y:ys) to insert x ys.

Once again we ran sort[4,3,2,1,6]. This time the result was [1,2,3,4]. The program terminated but with the wrong result. It was missing 6 form the list. Looking at the trace for sort[4,3,2,1,6] we saw that insert(1,[6]) was rewritten in a few steps to [1]. This prompted us to change if x <= y then x:ys to if x <= y then x:y:ys. Then when we ran sort[4,3,2,1,6] and we got [1,2,3,4,6].

6.2 Lambda Calculus Interpreter Example

The following Curry program contains a larger example that demonstrates context hiding and eager evaluation in TeaBag. It is a basic lambda calculus (chapter 5 of [56]) interpreter.

```
{-
 - Basic Lambda Calculus Interpreter
 - Created by Dr. Andrew Tolmach
 - Modified by Stephen Johnson
 -}
import List

type Var = String

{-
 - The lambda terms for this interpreter
 - are abstractions (Abs), applications (App),
 - and variables (Var).
 -}
```

```
data Exp = Abs Var Exp
         | App Exp Exp
         | Var Var


{- (fr e) returns the fr variables in e -}
fr :: Exp -> [Var]
fr (Abs v e) = delete v (fr e)
fr (App e1 e2) = union (fr e1) (fr e2)
fr (Var v) = [v]

maximum :: [[Char]] -> [Char] -> [Char]
maximum [] v = v
maximum (x:xs) v | length x > length v = maximum xs x
                 | otherwise = maximum xs v

{- (fresh vs) returns a Var guaranteed not to be in vs -}
fresh :: [Var] -> Var
fresh vs = (maximum vs "") ++ "x"  -- one simple way!

{- (subst m x e) returns the capture-avoiding
   substitution [M/x] e  -}
subst :: Exp -> Var -> Exp -> Exp
subst m x (Var y) | x == y  = m
                  | otherwise = Var y
subst m x (App a b) = App (subst m x a) (subst m x b)
subst m x (Abs y b) | x == y = (Abs y b)
             | notElem x (fr b) || notElem y (fr m) =
                Abs y (subst m x b)
             | otherwise =
                let z = fresh(union (fr b) (fr m))
                in Abs z (subst m x (subst (Var z) y b ))



{- (reduce e) returns e with all outermost redexes reduced.
   This may of course create new redexes. -}
reduce :: Exp -> Exp
reduce (App (Abs v a) b) = subst a v a
reduce (App (App a c) b) = App (reduce (App a c)) (reduce b)
reduce (App (Var v)   b) = App (reduce (Var v)) (reduce b)
reduce (Abs v e) = Abs v (reduce e)
reduce (Var v) = Var v

containsAbs :: Exp -> Bool
containsAbs (App (Abs v a) b) = True
containsAbs (App (App a c) b) = (containsAbs (App a c)) ||
                                (containsAbs b)
containsAbs (App (Var v)   b) = containsAbs b
containsAbs (Abs v e)         = False
containsAbs (Var v)           = False

leftInner :: Exp -> Maybe Exp
leftInner (App (Abs v a) b)
    | (containsAbs a) == False &&
      (containsAbs b) == False    = Just
                                    (reduce (App (Abs v a) b))
    | isNothing (leftInner a) &&
      isNothing (leftInner b)     = Nothing
    | isJust (leftInner a)        = Just (App (Abs v
                                      (fromJust (leftInner a))) b)
    | otherwise                   = Just (App (Abs v a)
                                      (fromJust (leftInner b)))
leftInner (App (App a c) b)
    | isNothing (leftInner (App a c)) &&
      isNothing (leftInner b)      = Nothing
    | isJust (leftInner (App a c)) = Just (App (fromJust
                                          (leftInner (App a c))) b)
    | otherwise                    = Just (App (App a c) (fromJust
                                          (leftInner b)))
leftInner (App (Var v) b)
    | isNothing (leftInner (Var v)) &&
      isNothing (leftInner b)     = Nothing
    | isJust (leftInner (Var v))  = Just (App (fromJust
                                         (leftInner (Var v))) b)
    | otherwise                   = Just (App (Var v) (fromJust
                                         (leftInner b)))
leftInner (Abs v a)
```

```
    | isNothing (leftInner a) = Nothing
    | otherwise               = Just (Abs v (fromJust
                                          (leftInner a)))
leftInner (Var v)             = Nothing

{- Call By Value: This is the same as reducing the
   leftmost inner redex each time -}
cbv :: Exp -> Exp
cbv e | isNothing (leftInner e) = e
      | otherwise               = cbv (fromJust (leftInner e))
```

This interpreter encodes $\lambda x.t$ as `Abs "x" t`, $t_1 t_2$ as `App` $\mathtt{t}_1$ $\mathtt{t}_2$, and variables as `Var "x"`. Since reading these data structures can be cumbersome the traditional lambda calculus form will be used where appropriate. Thus, instead of saying

$$\mathtt{App(App(Abs"x"(Abs"y"(App(Var"x")(Var"y"))))(Var"a"))(Var"b")}$$

we will say $((\lambda x.\lambda y.xy)a)b$.

When the interpreter evaluates $((\lambda x.\lambda y.xy)a)b$ to head normal form it returns $x(x(x(xy)))$. The evaluation of $((\lambda x.\lambda y.xy)a)b$ to head normal form should be ab. The steps the lambda calculus interpreter should take are shown in (6.1).

$$((\lambda x.\lambda y.xy)a)b \rightarrow (\lambda y.ay)b \rightarrow ab \tag{6.1}$$

We will show how we used TeaBag to find the cause of this bug. We started by generating a trace of all the rewrite steps. This trace turned out to be 267 steps long. Since this is too many steps to examine we decided to try and narrow in on some portion of the code where we thought the bug was occurring. Doing this would enable us to generate a shorter trace. The first place we looked at was the individual reduction steps of the lambda calculus interpreter. We can use the fact that the steps should follow the sequence in display 6.1 to find which step in the lambda calculus interpreter the bug is located. To do this we set a breakpoint on `cbv` which is a recursive function that calls the function `leftInner` to perform one lambda calculus reduction. By seeing what `leftInner` evaluates to we can see each of the reduction steps the lambda calculus interpreter takes in reaching head normal form for $((\lambda x.\lambda y.xy)a)b$. We used on demand eager evaluation to evaluate `leftInner` to head normal form as illustrated in figures 6.2 and 6.3.

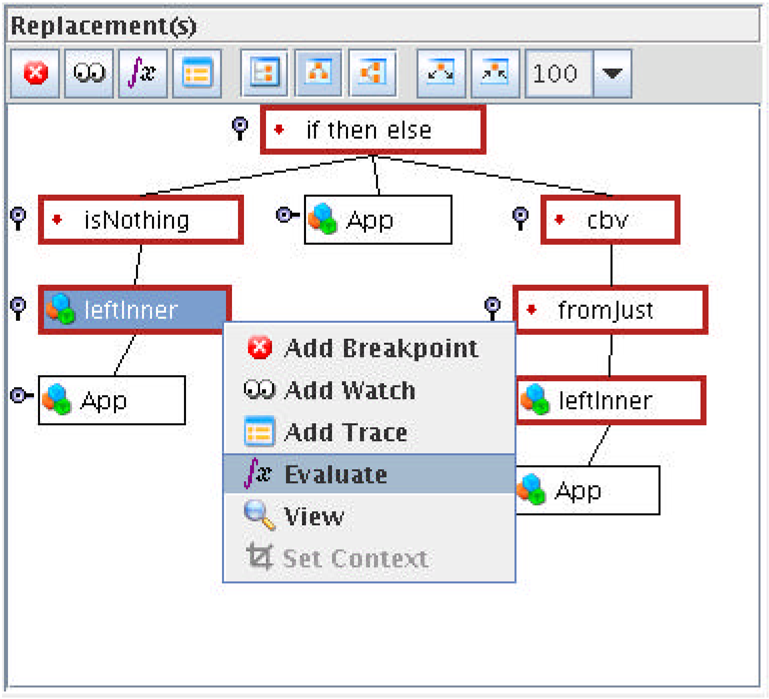


Figure 6.2: Selecting `leftInner` to evaluate.

For the first step of evaluating $((\lambda x.\lambda y.xy)a)b$ our lambda calculus interpreter reduced $((\lambda x.\lambda y.xy)a)b$ to $(\lambda y.((\lambda y.xy)y))b$. From display 6.1 you can see that this is not the correct second step. Part of this rewrite step is correct. The part

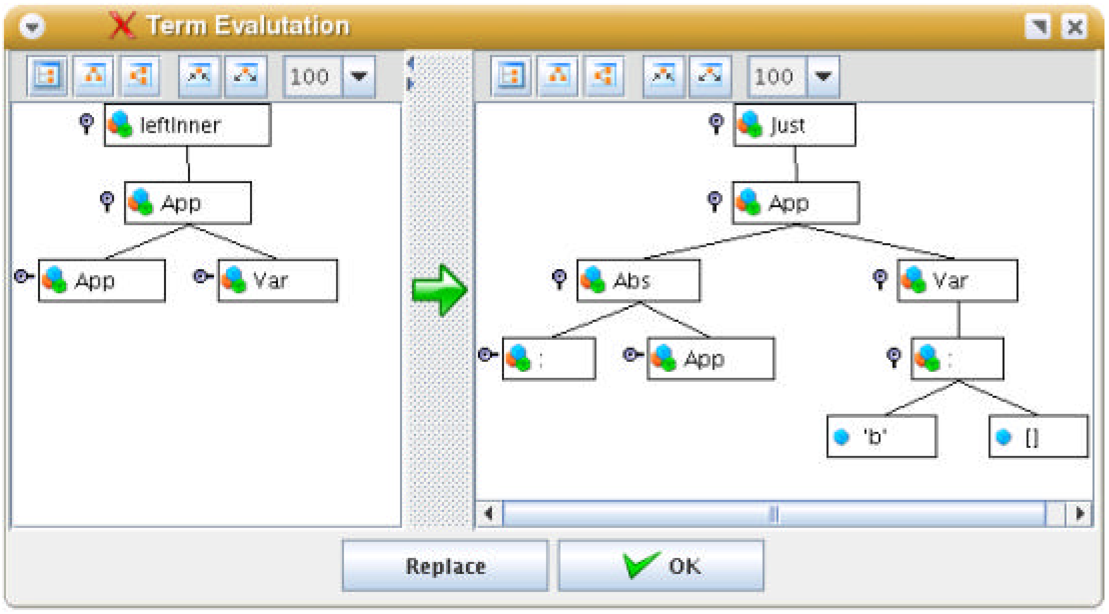


Figure 6.3: Result of evaluating `leftInner (...)`.

matching $(\lambda y.(...))b$ is correct. Since we now know that the bug is manifest during the first step of our lambda calculus interpreter we added a trace to the term rooted with `leftInner` to trace the first reduction step of our lambda calculus interpreter. This trace ended up being 96 steps long. While this is better than the 267 step trace, it is still too many steps to examine in detail to find the bug.

To further cut down the size of this trace we decided to perform on demand eager evaluation during runtime on the conditions of `if then else` terms and replace the conditions with `True` or `False`. The trace will only show the step for the replacement of the condition with `True` or `False`. It will not show the rewrite steps taken to obtain `True` or `False`. We first had to add a breakpoint on `cbv`. We then reran $((\lambda x.\lambda y.xy)a)b$. When the debugger halted for the breakpoint on `cbv` we added a breakpoint to the term rooted with `leftInner`. Now the debugger will halt when that term is rewritten.

When the debugger halted for the breakpoint on the term rooted with `leftInner` we added a trace to that term. This term is rewritten to an `if then else` term. We eagerly evaluated and replaced the condition for this term as illustrated by the sequence of figures 6.4, figure 6.5, and figure 6.6. Since this condition evaluated

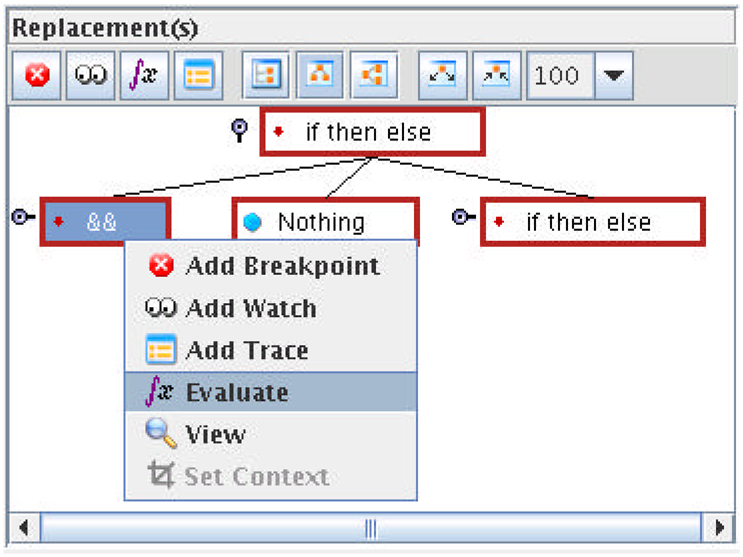


Figure 6.4: Selecting the condition of the `if then else` term for eager evaluation.

to `False` we knew that the third argument of the `if then else` term would be the replacement. This term is also an `if then else` term so we eagerly evaluated and replaced its condition. This condition evaluated to `True`. Since this condition evaluated to `True` we knew that the `if then else` term would be replaced with its second argument. So we expanded the second argument until we reached a term rooted with a function that we defined. This happened to be `leftInner`. We then set a breakpoint on this term. Next we ran the FLVM until this breakpoint caused FLVM to halt. We continued to perform these eager evaluations and re-

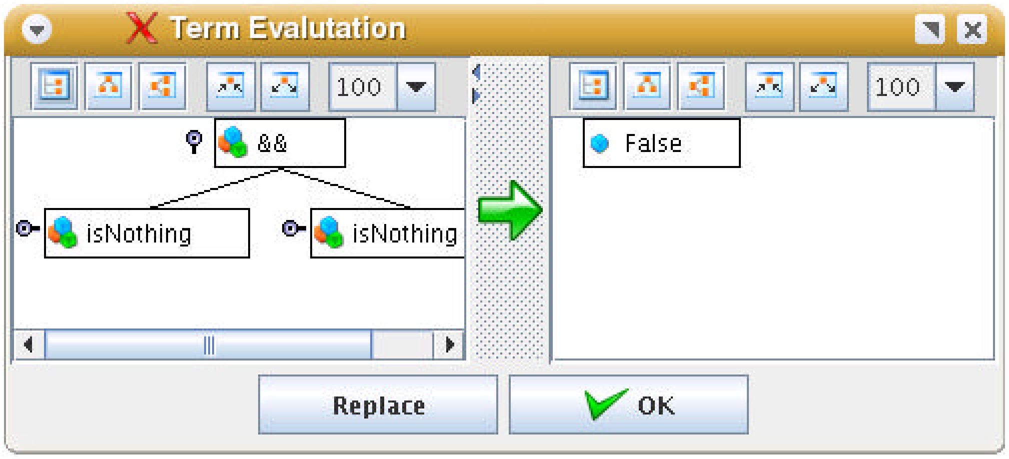


Figure 6.5: Result of eagerly evaluating the condition of the `if then else` term.

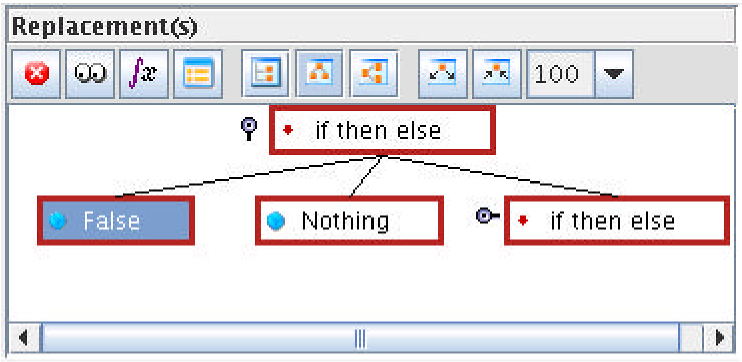


Figure 6.6: Replacing the condition of the `if then else` term with `False`.

placements on the conditions of `if then else` terms until there were no more to eagerly evaluate. Each time FLVM halted we added a breakpoint to the highest term that is rooted with a function we wrote. If there was an `if then else` term then we took the highest term rooted with a function we wrote under the branch that corresponds to how the condition evaluated and set a breakpoint on it. Doing this only required setting seven breakpoints and performing six eager evaluations and replacements of conditions.

By using on demand eager evaluation the trace of the first lambda calculus reduction was only 23 steps. Since this is a manageable number of trace steps to examine we started stepping through the trace looking for the bug. Looking at step 20 (figure 6.7) we noticed that for this to be correct the highlighted term needs to be `Var "a"`. This term needs to be `Var "a"` so that the result of this single step is $(\lambda y.ay)b$. However, this step shows that at the end of the first step of the lambda calculus interpreter we will end up with an expression of the form $(\lambda y.(\lambda y.xy)X)b$ where X stands for an unevaluated expression. We then worked our way backwards through the trace trying to see why we ended up with `Abs("y",App(Var "x", Var "y"))` instead of `Var "a"`. When we looked at steps 10 and 11 (figure 6.8) we noticed that we went from having one copy of `Abs("y",App(Var "x", Var "y"))` in step 10 to having two copies in step 11. We also noticed that in step 10 there is a subterm for `Var "a"` and that this subterm is not passed on to step 11. `Var "a"` is a term that should be in the reduction of the first step of the lambda calculus interpreter but is not. This prompted us to look more closely at this step. From the code highlighting we could see that `reduce (App (Abs v a) b) = subst a v a` was the rewrite rule

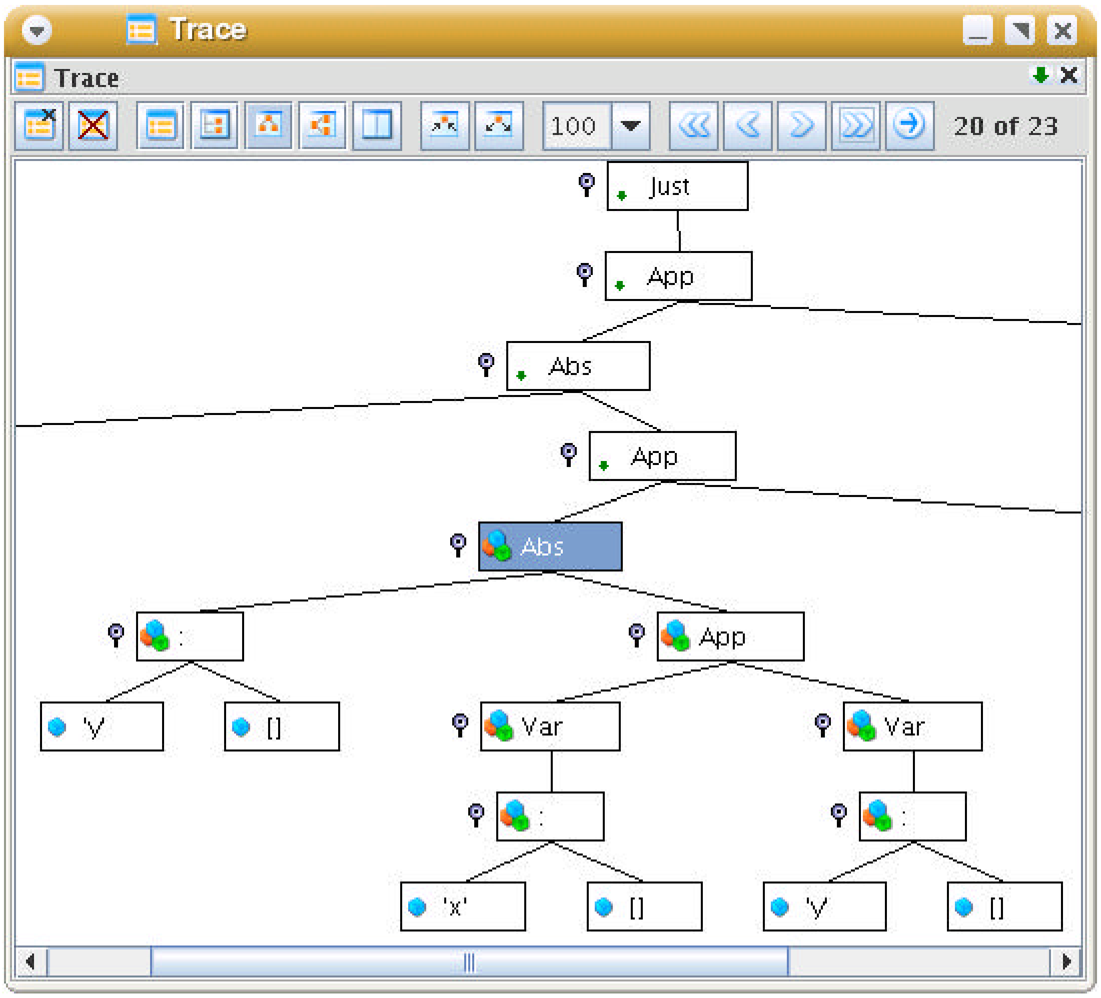


Figure 6.7: The highlighted term should be `Var a`.

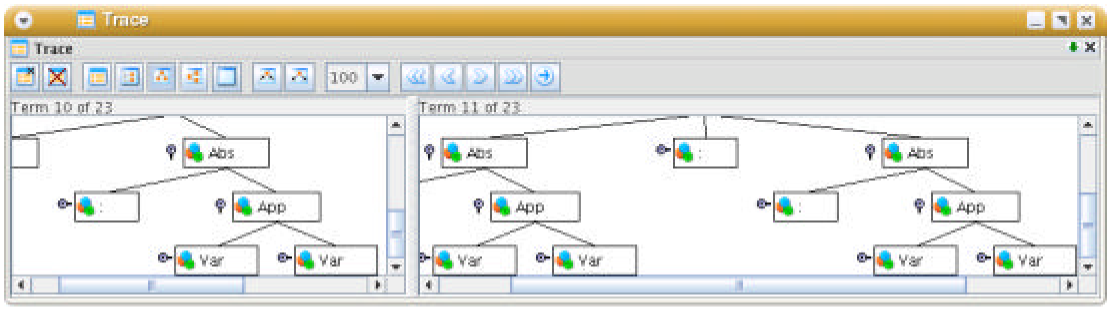


Figure 6.8: One ($\lambda y.xy$) in step 10 was duplicated to two copies in step 11.

executed to go from step 10 to step 11. We noticed that the variable `b` on the lefthand side of this rewrite rule did not appear on the righthand side and that it corresponded to the term `Var "a"` in the trace. This caused us to look at the definition of `subst`. `subst m x e` returns the capture-avoiding substitution [m/x] e. [m/x] e should substitute m for all occurrences of the variable x in e. At this point we found the problem. From step 10 to 11 in the trace, the substitution, `Var "a"`, was not being passed to `subst`. We changed this rewrite rule to be `reduce (App (Abs v a) b) = subst b v a`. With this fix our lambda calculus interpreter correctly reduced $((\lambda x.\lambda y.xy)a)b$ to ab.

6.3 Dutch National Flag Problem Example

The following example demonstrates the computation structure of TeaBag. Consider the *Dutch National Flag* program in figure 6.9.

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 6.10. We decided to follow the path through the computation structure that 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

```
data Color = red | white | blue

mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ red:z
           = solve (x ++ red:y ++ white:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red:z
           = solve (x ++ red:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= blue:y ++ white:z
           = solve (white:y ++ blue:z)
           where y,z free
solve flag | flag =:= mono red ++ mono white ++ mono blue
           = flag
```

Figure 6.9: Buggy Dutch National Flag Program

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

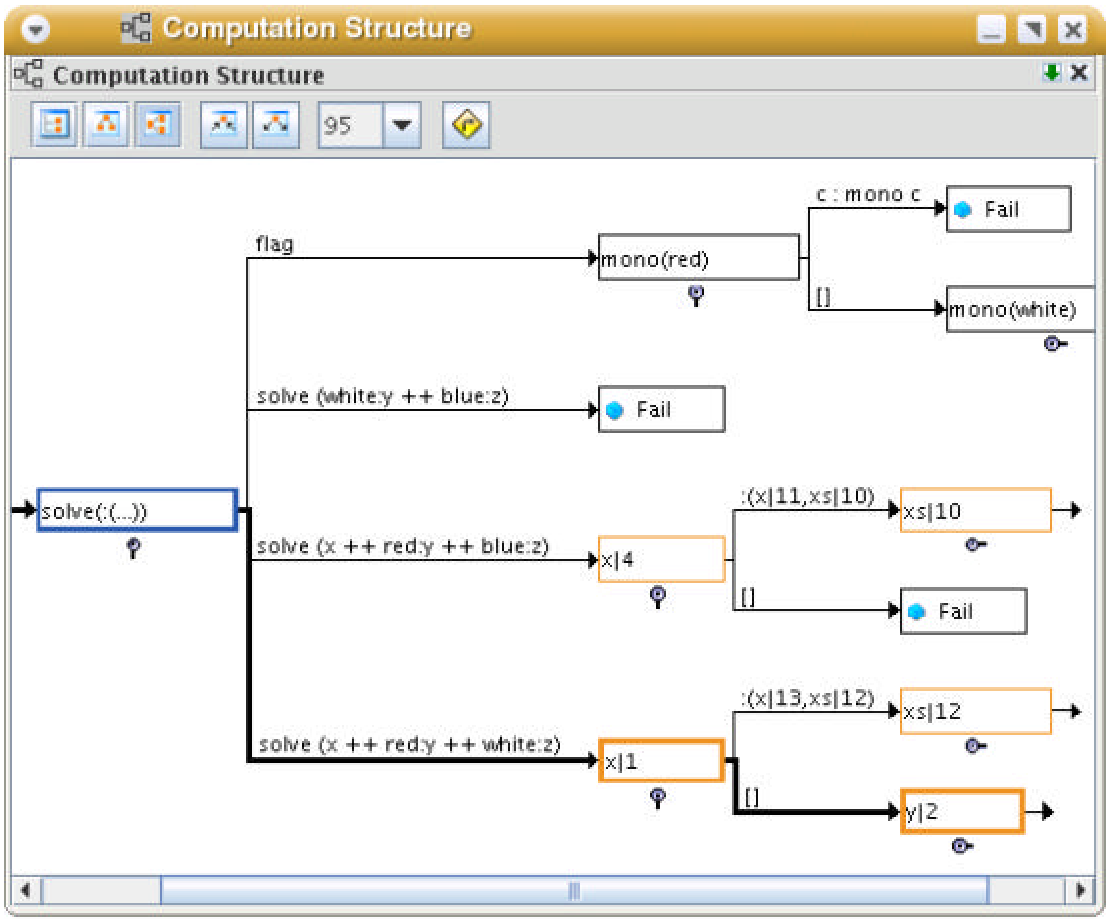
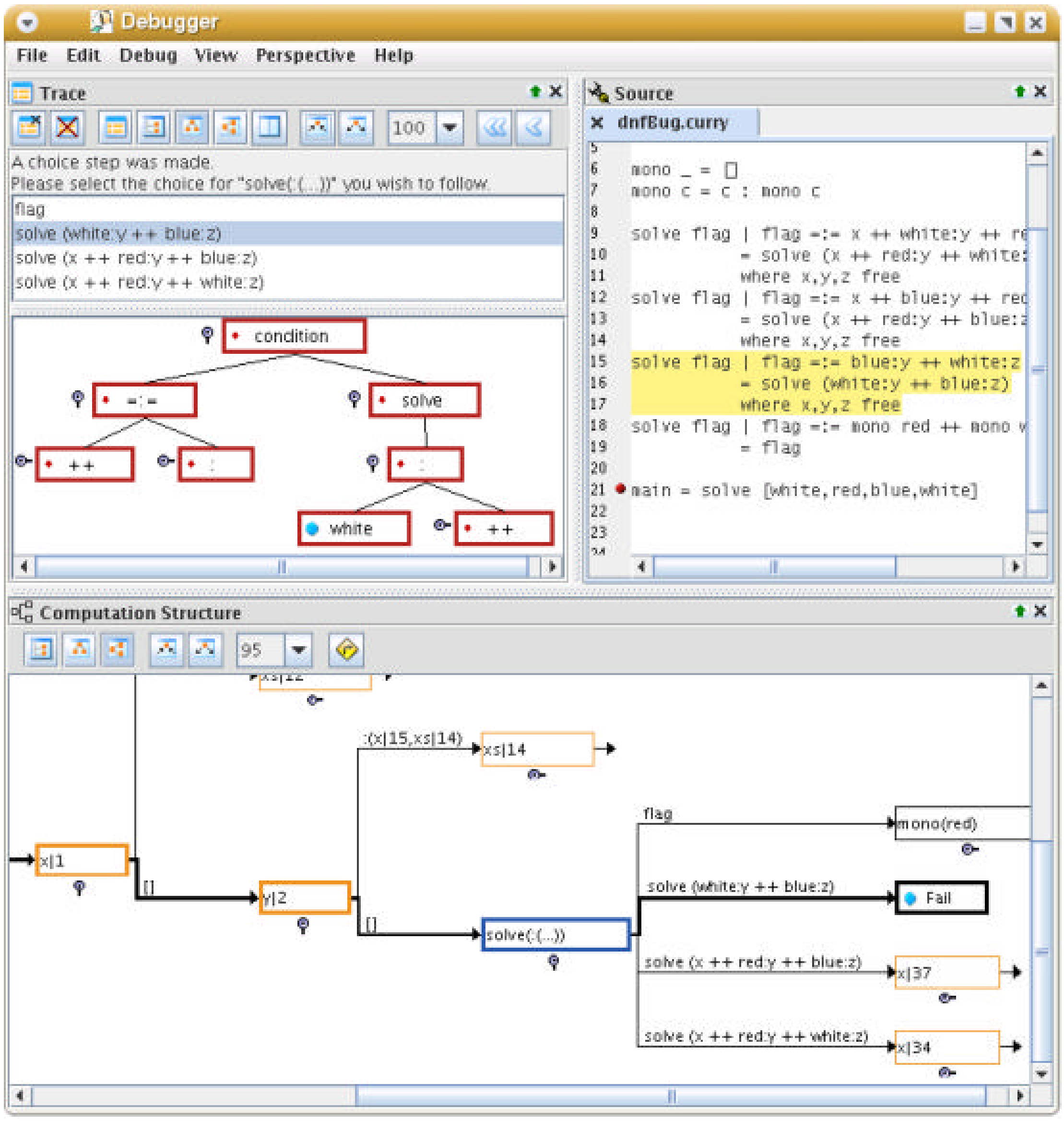


Figure 6.10: Computation structure for buggy Dutch national flag program

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 6.11 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



A choice step was made.
Please select the choice for "solve(:(...))" you wish to follow.

flag
solve (white:y ++ blue:z)
solve (x ++ red:y ++ blue:z)
solve (x ++ red:y ++ white:z)

```
mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ re
           = solve (x ++ red:y ++ white:
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red
           = solve (x ++ red:y ++ blue:z
           where x,y,z free
solve flag | flag =:= blue:y ++ white:z
           = solve (white:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= mono red ++ mono w
           = flag

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

Figure 6.11: Trace of buggy Dutch national flag program

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 6.11. 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 in figure 6.12.

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 6.10.

```
data Color = red | white | blue

mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ red:z
           = solve (x ++ red:y ++ white:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red:z
           = solve (x ++ red:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ white:z
           = solve (x ++ white:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= mono red ++ mono white ++ mono blue
           = flag
```

Figure 6.12: Correct Dutch national flag program

Chapter 7

Conclusion

7.1 Future Work

While we have made much progress on creating a debugger for Curry, there is still more that can be done. Firstly, we would like to get feedback from users about TeaBag. Our views and concepts about the usability of TeaBag for debugging contain natural prejudices since we created it. Thus, we would like to gain feedback from unbiased individuals on how well TeaBag was able to help them find bugs in their programs. Also, we would like to find out how well TeaBag performs as a debugger in true software development. Currently, there is no compliant compiler for TeaBag. Thus, debugging a program involves compiling the program partially by hand. Without a compliant compiler it is impossible to judge how well TeaBag performs during software development since hand compilation takes too long and is too error prone to be practical.

To be able to gain this feedback about TeaBag and to make TeaBag a debugger that can be used in real software development, a compliant compiler is needed. TeaBag is designed to use the FLVM for running Curry programs. The FLVM also has no compliant compiler. Thus this problem extends beyond TeaBag. However, TeaBag requires more compiled information, e.g. file positions and variable names, than the FLVM does. Adding this information by hand to the textual representation of the byte-code is tedious and time consuming. Thus, one of the first things that needs to be done for TeaBag is a compliant compiler needs to be created.

Curry programs that work with TeaBag are compiled to a textual representation of byte-code that is loaded by the FLVM. The instructions in this byte-code were modified to allow for debugging information such as file positions for highlighting. However, this couples the debugging code with the set of "normal" byte-code instructions. We would like to decouple this by creating new byte-code instructions that are debug specific. These instructions could be injected into the byte-code stream by a compiler. Thus information like file positions would be specified in these debug instructions. This would help to keep a separation of debugging information from normal information in the FLVM. We believe this will help with the long term maintenance of the FLVM's compliance with TeaBag's interface. By decoupling the "normal" instructions from the debug instructions both sets of instructions can be changed independent of the other.

In TeaBag the process of finding a particular result in the computation structure can be tedious. This is especially true when the search space is large. We would like to add a search feature to the computation structure that would allow the user to search for particular terms in the search space. This feature would be useful for wrong answer bugs. It would help the user quickly identify the path in the search space for the wrong answer.

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 watch events sent by the FLVM 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.

Many times it would be nice to see some of the previous steps during runtime debugging. We would like to make it possible for the user to view a user defined number of previous steps when a breakpoint is encountered during runtime debugging. The user should be able to define how many previous steps they wish to be able to see. The higher this number the more overhead there will be and thus the debugger will run slower. Conversely, having more steps available to the user gives them more information for debugging. To not incur the overhead of sending all steps over the socket to TeaBag the virtual machine would track these steps in a circular buffer and then send them to TeaBag when a breakpoint is encountered.

7.2 Related Work

Much effect has gone into declarative debugging of functional logic languages [45, 19, 44, 25, 6, 7, 5, 8, 26, 24, 4] while not as much work has been spent on tracing them. Three avenues for tracing functional logic languages have been researched. The first one is related to using box oriented debugging [36, 17, 16]. However, research in box oriented debugging of functional logic languages has been dormant for nearly ten years. The next one is expression evaluation tracers. This has been researched in the context of CIDER [37, 38]. Since CIDER is an IDE for Curry it does not focus on debugging. Rather the debugger is just one of the tools that the IDE provides. The authors of CIDER point out that their debugger is better suited as a teaching tool than for debugging large programs. This is understandable since the authors of CIDER were focusing on the plug-in static analysis tools that CIDER provides. Our research has focused on how to make the trace of narrowing steps for a functional logic language useful for debugging. Thus, our research falls into this second category of functional logic language tracers. The third category of tracers for functional logic languages is observational [21]. The work on this tracer was being done at the same time as our research. Observational debuggers show the user the values that functions and data structures evaluate to during runtime.

TeaBag is a debugger for Curry. There are three other debuggers for Curry: Münster [24], COOSy [21], and CIDER [38, 37]. Each of these debuggers take a different approach to debugging. Münster is declarative, COOSy is observational, and CIDER is an expression evaluation tracer.

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 narrowing 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 [33] 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 expression evaluation tracers for their debugger. CIDER is an IDE for Curry that contains a debugger. 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.

In both CIDER and TeaBag the user can set breakpoints. The use of breakpoints is different between the two debuggers. In CIDER breakpoints can be set in the trace browser. Clicking on the next button in the trace browser will move the trace to the next step where the redex is rooted with the function that the breakpoint is set on. However, I could not find a way to remove the breakpoint and then single step through the remaining rewrite steps. Being able to set breakpoints in the trace browser is a good way to deal with the size of the trace. It lets the user jump to places in the trace where a term rooted with a function they are interested in is being rewritten. However, once a user has moved the trace to a point they are interested in, they will typically want to single step through the trace. As best I could tell CIDER does not let the user do this. I could not find a way to remove the breakpoint without setting another breakpoint. So this means that in CIDER once a breakpoint is set the user can only jump from one breakpoint to the next. TeaBag does not let the user set breakpoints in the trace browser. Breakpoints are only used in the runtime debugger. The user can set a breakpoint on the function they are interested in. Then when the runtime debugger halts for that breakpoint they can trace the evaluation of the term rooted with the function with the breakpoint. So TeaBag lets users find particular steps in a trace by not starting the trace until the user finds a term they want to trace in the runtime debugger. While this provides the same functionality as CIDER it is not as convenient to use for jumping to a particular step in a trace. However, this method of setting breakpoints and adding traces in TeaBag provides context hiding. It can make the size of the traces smaller in TeaBag. It can also reduce the size of the terms that are displayed to the user. Thus, traces are typically easier to read in TeaBag. CIDER's idea of having breakpoints in the trace browser would be a nice feature to add to TeaBag.

The big difference between CIDER and TeaBag is how non-determinism is presented to the user. CIDER is a debugger for Curry programs running on the PAKCS system [35]. PAKCS compiles Curry programs by translating them to Prolog. Prolog uses backtracking to implement non-determinism. CIDER present the trace to the user as a linear sequence of steps where some of the steps are backtracking steps. (See section 3.2 for a discussion on linear traces in functional logic languages.) TeaBag also presents the trace as a linear sequence of steps. However, there are no backtracking steps in this trace. The trace is the sequence of rewrite steps taken to rewrite a term to one result. The trace of a term consists of multiple traces. There is one trace for each path in the search space. Thus the the user is presented with multiple traces. They can then select which trace they want to look at by selecting a path in the search space. This separation of trace steps from the search space is very beneficial when presenting a trace to the user. In CIDER the search space is linearized via backtracking and presented to the user as a linear sequence of steps. Thus the search space and the rewrite steps are mixed together. In TeaBag the search space is separated from the rewrite steps. The search space is presented as a tree of non-deterministic steps and the trace is presented as a linear sequence of rewrite steps.

7.3 Final Conclusion

This thesis has presented TeaBag which is a debugger for functional logic computations. TeaBag has been developed as an accessory of the FLVM, a virtual machine intended for the execution of functional logic 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] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2(1):51–89, February 1989.

[4] M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Correction of functional logic programs. In P. Degano, editor, *Proc. of the European Symp. on Programming, ESOP 2003*. Springer LNCS, 2003.

[5] M. Alpuente and F. Correa. Buggy user's manual. http://www.dsic.upv.es/users/elp/buggy/.

[6] 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.

[7] M. Alpuente, F. Correa, and M. Falaschi. Declarative debugging of funtional logic programs. In B. Gramlich and S. Lucas, editors, *Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 2001.

[8] M. Alpuente, F. J. Correa, M. Falaschi, and S. Marson. The debugging system Buggy. Technical report, DSIC-II/1/01, 2001.

[9] M. Alpuente, S. Escobar, and S. Lucas. UPV-Curry: An incremental Curry interpreter. In J. Pavelka, G. Tel, and M. Bartosek, editors, *Proc. of 26th Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'99*, volume 1725 of *Lecture Notes in Computer Science*, pages 327–335, Milovy, Czech Republic, November 1999. Springer-Verlag, Berlin.

[10] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

[11] 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.

[12] S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206, Florence, Italy, Sept. 2001. ACM.

[13] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[14] 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`.

[15] S. Antoy and S. Johnson. TeaBag: A functional logic language debugger. In Herbert Kuchen, editor, *Proc. of the 13th International Workshop on Functional and (constraint) Logic Programming (WFLP'04)*, pages 4–18, Aachen, Germany, June 2004.

[16] P. Arenas-Sánchez and A. Gil-Luezas. A debugging model for lazy functional logic languages. Technical Report DIA 94/6, 1994.

[17] P. Arenas-Sánchez and A. Gil-Luezas. A debugging model for lazy narrowing. In *PLILP*, pages 453–454, 1995.

[18] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[19] T. Barbour and L. Naish. Declarative debugging of a logical-functional language. Master's thesis, University of Melbourne, Melbourne, Australia, December 1994. Technical Report 94/28, Department of Computer Science, University of Melbourne.

[20] M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems.* Cambridge University Press, 2003.

[21] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proc. of the 6th Internationl Symposium on Practical Aspects of Declarative Languages*, Dallas, Texas, USA, June 2004.

[22] T. Brehm. A toolkit for multi-view tracing of Haskell programs. Master's thesis, RWTH Aachen, 2001.

[23] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proc. of the Logic Programming Workshop*, pages 127–138, 1980.

[24] R. Caballero and W. Lux. Declarative debugging for encapsulated search. In M. Comini and M. Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.

[25] R. Caballero and M. Rodrguez-Artalejo. A declarative debugger of wrong answers for lazy functional logic programs. In *Proc. of the 10th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, 2001.

[26] R. Caballero and M. Rodrguez-Artalejo. A declarative debugging system for lazy functional logic programs. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier, 2002.

[27] 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 M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 176–193, Aachen, Germany, September 2000.

[28] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In R. Pena and T. Arts, editors, *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, LNCS 2670, pages 165–181, March 2003. Madrid, Spain, 16–18 September 2002.

[29] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[30] M. Ducassé. Abstract views of Prolog executions in Opium. In B. du Boulay, P. Brna, and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Ablex, Cognitive Science and Technology. 1999.

[31] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport de recherche INRIA RR-3257 and Publication Interne IRISA PI-1127.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 2001.

[33] A. Gill. Debugging Haskell by observing intermediate data structures. In *Proc. of the 4th Haskell Workshop*, 2000. Technical report of the University of Nottingham.

[34] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[35] 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.

[36] 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.

[37] M. Hanus and J. Koj. CIDER: An integrated development environment for Curry. In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming, (WFLP'01)*, Kiel (Germany), 2001.

[38] 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`.

[39] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[40] J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.

[41] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[42] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[43] 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.

[44] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[45] L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In G. Forsyth and M. 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. Defense Science and Technology Organization, DSTO General Document 51.

[46] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, January 1996.

[47] H. Nilsson. Tracing piece by piece: Affordable debugging for lazy functional languages. In *Proc. of the 1999 ACM SIGPLAN international conference on Functional programming*, pages 36–47, Paris, France, September 1999. ACM Press.

[48] 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.

[49] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

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

[51] Object Modeling Group. OMG unified modeling language specification. http://www.omg.org/technology/documents/formal/uml.htm.

[52] J. O'Donnell and C. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, September 1988.

[53] R. A. O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, MA, 1990.

[54] A. Penney. *Augmenting Trace-based Functional Debugging.* PhD thesis, Department of Computer Science, University of Bristol, September 1999.

[55] S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, 1999.

[56] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[57] B. Pope. Buddha: A declarative debugger for Haskell. Honors thesis, University of Melbourne, June 1998.

[58] C. Reinke. GHood – graphical visualisation and animation of Haskell object observations. In *Proc. of ACM Sigplan Haskell Workshop 2001*, September 2001.

[59] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[60] J. R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *J. ACM*, 21(4):622–642, 1974.

[61] 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.

[62] Ian Sommerville. *Software Engineering.* Addison Wesley, sixth edition, 2001.

[63] 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.

[64] J. Sparud. Towards a Haskell debugger. In *Chalmers DofCs annual Wintermeeting*, 1994.

[65] J. Sparud and H. Nilsson. An embryo to a debugger for Haskell. In *Proc. of the annual internal workshop Wintermotet, held by the Department of Computing Sciences, Chalmers University of Technology*, Goteborg, Sweden, 1994.

[66] J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional langauges. In M. Ducassé, editor, *Proc. 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.

[67] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In *ninth international Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177, 1997.

[68] J. Sparud and C. Runciman. Tracing lazy functional computations using Redex Trails. In *PLILP*, pages 291–308, 1997.

[69] Jan Sparud. A transformational approach to debugging lazy functional programs. Master's thesis, Chalmers University of Technology, January 1996.

[70] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, MA, 1986.

[71] The ART Team. Hat tutorial: Part 1, 2002. http://www.haskell.org/hat/.

[72] 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.

[73] A. Tolmach. *Debugging Standard ML.* PhD thesis, Princeton University, 1992.

[74] A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and store. In *Proc. of the 9th International Conference on Functional Programming (ICFP 2004)*, 2004. to appear.

[75] P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.

[76] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: A new Hat. In *Proc. of the Haskell Workshop 2001*, Firenze, Italy, 2001. Final version to appear in ENTCS.

[77] R. Watson. *Tracing Lazy Evaluation by Program Transformations.* PhD thesis, School of Multimedia and Information Technology, Southern Cross University, March 1997.

[78] R. Watson and E. Salzman. A trace browser for a lazy functional language. In *Proc. of the Twentieth Australian Computer Science Conference*, pages 356–363, 1997.

[79] R. Watson and E. Salzman. Tracing the evaluation of lazy functional languages: A model and its implementation. In *Asian Computing Science Conference*, pages 336–350, 1997.

[80] J. Wilcox. Improving the efficiency of narrowing computations in Curry. Portland State University, Honors Thesis, 2004. http://www.cs.pdx.edu/∼jasonw/thesis.