Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 177 (2007) 91–106
www.elsevier.com/locate/entcs

A Framework for Interpreting Traces of Functional Logic Computations

Bernd Braßel[1,2]

Institute of Computer Science
Christian-Albrechts-University of Kiel
Germany

Abstract

This paper is part of a comprehensive approach to debugging for functional logic languages. The basic idea of the whole project is to trace the execution of functional logic programs by side effects and then give different views on the recorded data. In this way well known debugging techniques like declarative debugging, expression observation, redex trailing but also step-by-step debuggers and cost center oriented symbolic profiling can be implemented as special views on the recorded data. In addition, creating new views for special debugging purposes should be easy to implement. This is where the contribution of this work sets in. We describe how the recorded data is interpreted and preprocessed in order to yield an extremely simple yet versatile interface to base the different views on. Using this interface, formulating the basic functionality of declarative debugging, for example, is a matter of a few lines.

Keywords: debugging, functional logic programming

1 Introduction

1.1 The Problem

It is the basic credo of declarative programming that abstracting from certain aspects of program executions greatly improves the quality of the written code: Typical sources of errors are principally omitted, like issues of memory management, type errors and multiple allocation of variables. The program is much nearer to the *logic* of the implemented algorithm than to its execution. This makes code much more readable, comprehensive and maintainable.

There seems to be at first glance, however, a great drawback to these techniques: As there is such a far abstraction from the actual program execution, the executed program becomes a black box. Where an imperative programmer is able to step

[1] This work has been partially supported by the DFG under grant Ha 2457/5-1.
[2] Email:bbr@informatik.uni-kiel.de

doi:10.1016/j.entcs.2007.01.006

through his program's execution and recognize parts of his programs, the declarative programmer is usually not able to draw any such connections. This is of course an especially severe problem for *debugging*.

1.2 Related Work

There are many approaches in the literature to close this gap between the source code and its execution. Among the many techniques proposed so far we can only name a few and give a broad categorization:

Visualization of Computation A straightforward approach to search bugs is to represent the actual program execution in a human readable form and to provide tools to comfortably browse this representation. Such tools, beginning with step-by-step debuggers, have been developed for many languages, imperative and declarative alike. These tools normally depend on a specific backend of the supported language and seldom aim at portability. Some very elaborated examples for declarative languages include ViMer [11] for the logic language Mercury [28], Ozcar [21] for the Mozart system [3], a backend for the language Oz [27] and TeaBag [3] for the FLVM implementation [2] of the functional logic language Curry [18].

Value Oriented Debugging approaches based on analyzing what values have been computed by evaluating a given expression within the program are for instance declarative debugging (cf. [26] for logic, [22,23] for functional, [10] for functional logic programming), observations for lazy languages (cf. [15] for functional [5,19] for functional logic languages [4]), backward stepping and redex trailing (for functional languages only, cf. [4] resp. [29]).

Performance Oriented Sometimes the bug is not in the computed values but in its failing efficiency. The general approach to analyze the frequency and duration of function calls is mostly known as "profiling". Profilers measuring actual run times are naturally dependent on a specific backend. Traditional profiling methods do not readily translate to lazy languages. A solution to this problem – attributing execution costs to user defined cost centers – was proposed in [25] for the GHC [5] for the functional language Haskell [24] and ported for PAKCS [17], an implementation of the functional logic language Curry, in [8]. In adition to runtime profiling, both approaches feature a more abstract and therefore much more portable approach to profiling which is called "symbolic profiling". Such abstract measurements are not only more portable but also accessible to verification.

Special Purpose Tools Under this catch-all category we would like to mention some approaches which give backend depending information about special features of the program execution. Among many existing systems are stack inspection for the GHC [14], a statistic overview of the search space available in the Oz debugger [21], the graphical representation of profiling data for the GHC [24] and GHood, an animated graphical viewer of observations [15].

[3] http://www.mozart-oz.org

[4] [19] is part of this volume.

[5] http://www.haskell.org/ghc/

The tools and categories above can only give a remote hint to the magnitude of tools giving information about the execution of declarative programs. As is often the case with such a multi-faceted research field: the same problems are solved many times and many basic approaches have to be reinvented time and again. How to cope with large applications? How to obtain information if no direct access to the back end is given? Is the represented data correct and is it complete or do we miss something? Wouldn't it be nice to have the same tool they got for that other backend for our language? The first approach that gave the basic idea that these problems might be solvable after all was the further development of redex trailing as proposed in [12]. There the authors observed that the data collected for redex trailing was also sufficient to provide declarative debugging as in the systems Freja [23] and observations like in Hood [15]. The approach of [12] is also more portable than Freja and a more powerful implementation of Hood. Freja was implemented as a special Haskell compiler available only for the Solaris operation system, and the more powerful version of Hood had to be integrated in the Haskell interpreter Hugs [6] in order to achieve some additional features. The key idea to obtain this portability was to transform the given program and collect the information by side effects rather than relying on a specific backend.

1.3 Our Approach

In [6,9], we have extended the basic ideas of [12] in several ways. First, our approach supports the additional features available in functional *logic* languages, i.e., free variables and non-deterministic functions. In addition, we have based our approach on a *core language* which features the main concepts of functional logic languages. This language, called "Flat Curry", is described in detail in [1] and cannot be fully developed here.

Functional logic languages like Toy [20] or Curry can be translated to this core language (and actually are in some implementations of Curry). On one hand this is one step away from the original source program but on the other hand this approach has some important advantages:

Portability At least conceptually, our approach is open to be ported to all declarative languages which can be translated to Flat Curry, including lazy functional languages. The program transformation, cf. [6], maps a valid Flat Curry program to another valid Flat Curry program. The only features the backend has to support in order to execute the transformed program are some basic functionality to create side effects like "`unsafePerformIO`".

Verifiability A considerable part of the formal foundation of functional logic languages has been developed with respect to Flat Curry, cf. [1]. [7] Therefore, we were able to give proves about correctness and completeness of the collected data

[6] http://www.haskell.org/hugs/

[7] The other main thread of formal reasoning about functional logic languages is based on [16]. It is still a desideratum to give the missing prove link between the CLN calculus of [16] or one of its further developed successors with the big-step semantics of [1]. A good place to start might be the DN calculus of [13].

in [9] which was not yet possible for the approach of [12] [8]

In addition to the points above, we also extended the approach of [12] by recording information about the actual pattern matching performed while executing the program. This information turns out to be crucial when integrating more of the tools described in Section 1.2. The HAT system of [12] was able to emulate, among others, redex trailing, observations and declarative debugging because these are all value oriented techniques. These techniques are concerned with the *denotational* mapping between expressions and their values. The other tools mentioned in Section 1.2 are concerned with *operational* aspects of the program execution. As it is not possible to reconstruct the state transformation induced by executing the program from the data recorded by HAT, it is not possible to integrate such more operational tools. In our approach, in contrast, the operational behavior of the program can be reconstructed and, thus, at least conceptually, the whole range of tools mentioned can be emulated as special views on the recorded data.

1.4 Contribution of this Work

Up to now we have described the comprehensive approach of the overall project. Naturally, this article can only make a partial contribution to this project.

The present paper is concerned with how to provide a simple yet versatile interface to the traces of program executions in the functional logic language Curry. It describes on one hand how the traced data is represented in Curry (Section 2) and proposes a much simpler data structure to represent general computations (Section 3.1). In addition, techniques of how to elegantly obtain and process this data structure are described (Section 3.2). Using this structure it should be easy to implement tools like the ones mentioned in Section 1.2, at least as far as the access to run-time data about program executions is concerned.

The basic idea is that representing computations in the framework of functional logic languages can be as simple as categorizing computation steps into a) single step b) subcomputation c) branching. A single step might be further distinguished to be an unfolding, the binding of a variable, the suspending of a computation or perhaps some representation of a side effect. Value oriented techniques are then characterized by subcomputation to the *most evaluated* form whereas operation oriented tools feature subcomputations to *head* normal form only. These different kinds of subcomputations can be seen as interpreting the program trace in the light of different evaluation strategies. Computing the most evaluated form is like employing a strict strategy while stopping at head normal form is lazy evaluation.

This paper represents work in progress in several respects: 1) It is meant as a proposal of a simple yet versatile interface. So far, declarative debugging, step by step debuggers and visualization as proof trees have been implemented as simple views. A more sophisticated view allows the user to interactively browse the program's interpretation, enabling him to open and close subderivations and

[8] According to personal communication with O. Chitil, formal reasoning for the approach of [12] is forthcoming.

non-deterministic choices at will. Whether or not he uses these views with a value oriented or operation oriented interpretation of the execution trace is up to the user. 2) The implementation is unstable but will be available soon under http://www-ps.informatik.uni-kiel.de/~bbr/. 3) For the overall project of tracing functional logic programs some work is still to be done in order to cope with large applications.

2 Tracing and its Results

As mentioned above, this paper is based on two preliminary works. In [9], we have extended a semantics for functional logic languages by the construction of a trace graph. In [7] we have presented a program transformation which writes by side effects information into a file from which a graph in the sense of [9] can be produced. [9] includes a proof that the computed graph correctly represents relevant aspects of the program executions. Therefore we can omit deeper details here and simply present an example of how the computed graphs look like.

Example 2.1

```
data Nat = Z | S Nat
add Z y = y
add (S x) y = S (add x y)
eq Z Z = True
eq (S x) (S y) = eq x y
main = eq (add x x) Z where x free
```

Before discussing the trace graph corresponding to the evaluation of `main`, we give a version of functions `add` and `eq` with explicit pattern matching. This will be useful when comprehending the graph.

```
add x y = fcase x of {Z -> y; S x' -> S (add x' y)}
eq  x y = fcase x of {Z    -> fcase y of {Z -> True};
                      S x' -> fcase y of {S y' -> eq x' y'}}
```

The trace graph resulting from the evaluation of `main` is depicted in Figure 1. Note, that there are some differences to the graphs as defined in [9] which will be discussed below. In Figure 1 you can see three kinds of arrows:

- Successor arrows have a normal shape and represent a reduction. For instance, there is a successor arrow between the node labeled `main` and the node with label `eq`. This corresponds to the fact that the function `main` directly reduces to a call to function `eq`.
- Parent arrows are in dashed style. There are two basic cases in which parent arrows appear: 1) If node A is successor of node B then B is the parent of A. In the example, the node labeled `main` is the parent of the node labeled `eq`. 2) if the evaluation of an expression was demanded by pattern matching (represented by

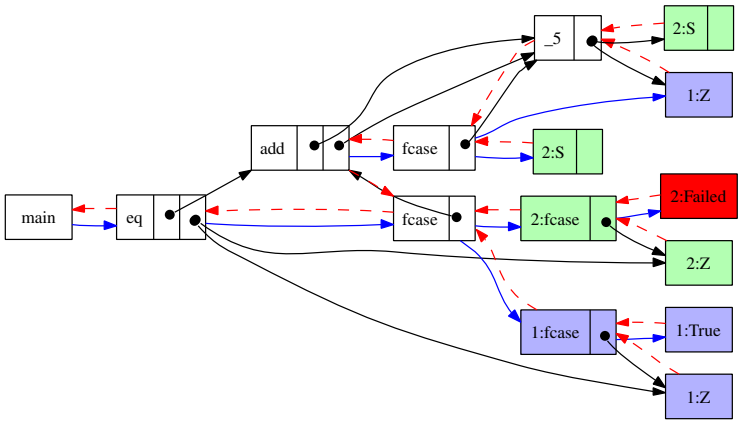

Fig. 1. Trace Graph of Example 2.1

`case` nodes, cf. below) then the parent of the node representing this expression is a node representing the pattern matching. In the example, the node labeled `1:Z` at the bottom line represents the `Z` in the call `(eq (add x x) Z)` in the example. This `Z` was evaluated by the pattern matching of the first rule of function `eq` and its parent is therefore a node labeled `fcase`.

- Argument arrows have a dot at their origin. A node referred to by an argument arrow represents an expression which was an argument of the function represented by the node from whence the arrow came. In the example, the node labeled `add` is the origin of two argument arrows both pointing to the node `_5`, which represents a free variable. This corresponds to the expression `(add x x)` of the example program.

The description of the arrows suggests that there are also different kinds of nodes in the trace graph:

- There are application nodes like the ones labeled with `main`, `add` or `True`. These nodes represent the unfolding of the function (resp. application of a constructor) corresponding to the label. Each application node has one position for each argument of the corresponding function (or constructor).
- Case nodes represent pattern matching and are either labeled `case` or `fcase`. This corresponds to the distinction between residuation (`case`) and narrowing (also called flexible matching and therefore written `fcase`). Each case node has one argument position, where the expression is referred to which is evaluated to head normal form in order to match with a given pattern. In the example the `fcase` node which is the successor of the node labeled `eq` represents the pattern matching on `eq`'s first argument. To match the pattern, the expression `(add x x)` has to be evaluated and therefore the corresponding node is referenced by the argument arrow of that `fcase`.
- Variable nodes are labeled with `_` and a number for identification. Each variable has one argument position in which the binding(s) of the variable are referenced. In the example, `_5` represents a free variable, which is bound to `S _` and `Z` re-

spectively. Note that the argument of `S _` was never evaluated in this example and is therefore not represented in the graph.

- Failure nodes, labeled with `Failed` represent an unsuccessful pattern matching. In the example, the matching `(fcase Z of {S y' -> eq x' y'})` (part of evaluating function `eq` in the program) is responsible for the only `Failed` node in the graph.

There is only one detail of Figure 1 left to explain. Some node labels are headed by numbers like `1:Z` or `2:Failed`. These numbers denote the so called *path* of a computation. (For convenience, in Figure 1 nodes with the same path also have the same color. The only exception are failure nodes, which are always red.) Each computation has a path starting with the empty path for `main`. This path is extended whenever a non-deterministic branching occurs. Each branch gets a different number and thus, different paths mean that the nodes belong to different branches of the computation. The original tracing semantics [9] non-deterministically computes two graphs for the above example, as shown in Figure 2. It can be seen immediately

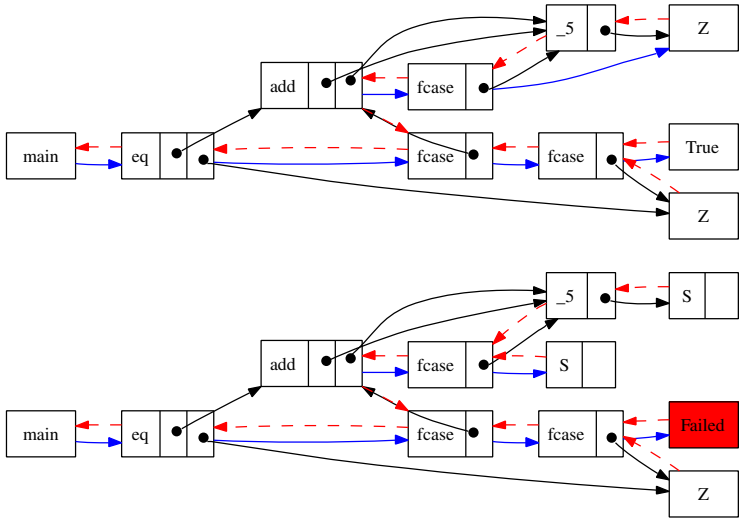

Fig. 2. The two Graphs of Example 2.1 produced by the Semantics of [9]

that it is much more economic to produce a single graph, which is an overlay of all the graphs produced by the original semantics. The connection between the graphs can be seen immediately when considering the paths. We call a path p equal or smaller than a path q, with the usual notation $p \leq q$, if p is a prefix of q. When we take the set of all paths attached to nodes in the overlay graph, each path of this set which is maximal with respect to $\leq$ corresponds to a graph produced by the original semantics. For each maximal element m of this set, the corresponding graph can be obtained by taking only those nodes, whose attached path q satisfies $q \leq m$. An example can be obtained by comparing Figures 1 and 2.

In [7] we described how to transform a given Flat Curry program such that during its execution a file is written by sideeffects. The generated file contains a codified

version of the graphs introduced above. This codified version has to be rehashed into a more declarative structure, which is described in the next subsections.

2.1 Representation of Trace Graphs

In *lazy* functional (logic) languages there are possibilities to construct graphs in an elegant way. First, *sharing* already introduces directed acyclic graphs. For instance, both arguments of the tuple introduced by `(let x=e in (x,x))` physically refer to the same memory address at run time. But also cyclic graphs can be constructed where recursive let expressions are allowed. For instance, the expression `(let ones=1:ones in ones)` introduces at run time a structure with a cyclic reference in the heap. Representing graphs in this way has some advantages:

- Following edges in the graph is an operation with a constant cost.
- Programming by pattern matching is possible.
- Unreferenced parts of the graph can be detected by garbage collection.

Therefore, we can represent trace graphs with the simple structure:

```
type Path = [Int]
data TraceGraph = Nil Path
                | Node Int Path TraceGraph [TraceGraph] TraceInfo
```

The graph consists of nodes (`Node`) and leafs (`Nil`). Leafs represent subexpressions which were not evaluated during program execution. Each node of the graph has a reference of type Int (to allow node identification) and a path which is represented by a list of integers (cf. the discussion above). Note, that leafs also have paths in order to support language implementations which do not feature sharing of evaluations across non-deterministic branches. In such an implementation, subexpressions might be evaluated in one branch but stay unevaluated in another and, thus, a leaf might belong to a special path only. In addition to reference and path, nodes also have a parent node and a list of successor nodes. (There is always a single parent but there may be more than one successors, cf. Figure 1 above.) In addition, each node has some special information which represents what kind of node it is:

```
data CaseMode  = Flex | Rigid
data TraceInfo = App String [[TraceGraph]] | Or
  | Case CaseMode TraceGraph | Free Int [TraceGraph] | Failed
```

Note that application nodes (`App`) contain a list of lists of trace graphs. This is because in different computation branches the arguments of an application node might point to different expressions.[9] For example, the node labeled `eq` in Figure 1 has two different pointers in its second argument. This `eq` node is represented as

```
Node 1 [] (Node 0 [] (Nil []) (App "main" []))
   (App "eq" [[Node  3  [] (Node  2 ... (Case Flex (Node 3 ...))) (App "add" [...])],
              [Node  9 [1] (Node  8 ...) (App "Z" []),
               Node 14 [2] (Node 13 ...) (App "Z" [])]])
```

[9] This also happens only if there is no sharing of evaluations across non-deterministic branches.

The "..." are not only to shorten the example. Because of the cycles in the structure it is impossible to give a complete term representation. For instance, in the run-time heap, the argument node of the flexible case (`Case Flex (Node 3 ...)`) is identical with the first argument of `eq`, `(Node 3 ...)` as you can see in Figure 1.

2.2 Implementation of Trace Graph Building

During the execution of the transformed program a file is generated, in which all parts of the graph are codified by numbers, called *references*. There are two separate spaces of references, one for the successor and parent relation and one for argument pointers. Such a trace is a sequence of pieces of information of the three kinds:

```
data TraceItem = Successor Int Int
               | RedirectArg Int Path Int
               | TNode Int Path Int ItemInfo
type Trace = [TraceItem]
```

Successor i j The node with reference `j` is successor of the node with reference `i`.

RedirectArg p ar nr Each application node with argument reference `ar` belonging to the computation of path `p` should be replaced by a reference to the node with number `nr`.

TNode r p par info The node with number `r` belongs to the computation of path `p` and has the node with number `par` as parent. The kind of the node (application, failure, free variable or case, cf. above) is then given in the info part which will not be considered in the following.

If we assume a data structure to associate integer keys with data elements like a search tree, hash table, array or similar, with the following interface:

```
data Mapping a = ...
lookup :: Mapping a -> Int -> a
insert :: Int -> a -> Mapping a -> Mapping a
empty  :: Mapping a
```

Then the building of the graph as a cyclic data structure can be implemented as a function manipulating three of these search structures: 1) a mapping of node references to the list of their successor references 2) a mapping of argument references to the list of their corresponding node references and their paths 3) one mapping of node references to trace nodes. (The Structure of trace nodes was defined in Section 2.1).

```
type Maps = (Mapping [Int],Mapping [(Int,Path)],Mapping TraceGraph)

traceToCycGraph :: Trace -> TraceGraph
traceToCycGraph tr = let (_,_,ns) = cycle tr (empty,empty,empty) in
  lookup ns mainReference

cycle :: Trace -> Maps -> Maps
```

```
cycle [] maps = maps
cycle (Successor x y:xs) (sMap,aMap,nMap) =
  cycle xs (insert x y sMap,aMap,nMap)
cycle (RedirectArg v p ref:xs) (sMap,aMap,nMap) =
  cycle xs (sMap,insert v (ref,p) aMap,nMap)
cycle (TNode ref path par info:xs) (sMap,aMap,nMap) =
  let maps = cycle xs (sMap,aMap,insert ref node nMap)
      (sMap2,aMap2,nMap2) = maps
      sucs = map (lookup nMap2) (lookup sMap2 ref)
      node = TraceNode ref path (lookup nMap2 par) sucs
                       (buildInfo maps info)
   in maps
```

The rules for `Successor` and `RedirectArg` only add information to the maps. The last rule contains the recursive let which adds the information of the current trace node to the node map. The elements of these trace nodes depend on the call to cycle on the thus updated map. This ties the loop and makes sure that the result of `cycle` is a cyclic structure in the heap which directly resembles the trace graph. An elegant definition in this way is only possible in lazy languages.

There are, however, drawbacks to this technique: This definition can only work efficiently if the whole trace fits into memory. This is not to be expected for all applications we would like to be able to debug. Therefore there is an alternative implementation to build the trace graph. This alternative implementation represents the graph as a potentially infinite term. Each node is upon demand retrieved from the trace file by side effects. This is comparable to lazy file access by the Curry standard function `readFile`. The access to the parents, successors or arguments is not possible in constant time as it involves some kind of binary search on the file for each access. But as there are no cycles in the graph, the degree of heap referencing is much lower and therefore trace nodes can become garbage much more often. This ensures that the program will only have parts of the trace graph in memory at each moment.

Advantages and disadvantages of the two alternative implementations can be summarized as follows:

	Cyclic Graph	Infinite Graph
Access to successor	in constant time	in logarithmic time
Access to value	in constant time	linear in chain length
Processed nodes	not always garbage	always garbage
application	normal traces	huge traces

It remains to be evaluated where the border between “normal” and “huge” is.

3 A Framework for Interpreting Trace Graphs

The basic idea of providing a simple yet versatile interface to program views on the traced program executions is to represent the trace as a sequence of computation steps. These steps are categorized into a) single step b) subcomputation c) branching. A single step might be further distinguished to be an unfolding, the binding of a variable, the suspending of a computation or perhaps some representation of a side effect. Value oriented techniques are then characterized by subcomputation to the most evaluated form whereas operation oriented tools feature subcomputations to head normal form only. These different kinds of subcomputations can be seen as interpreting the program trace in the light of different evaluation strategies. Computing the most evaluated form is like employing a strict strategy while stopping at head normal form is lazy evaluation. For debugging, the main idea is that it is much easier to understand the execution of a program, if it is evaluated with a simple strategy. It is therefore better to understand a strict evaluation of the program than a lazy one. This is just another way of saying that value oriented approaches (cf. Section 1.2) try to show the results as if they were evaluated strictly.

Of course, evaluating the given expression in a fully strict manner is not going to work, as the expression might contain potentially infinite structures. Therefore, strict evaluation is generalized to what we call *strict evaluation with oracle*. Beside the unusual name, the basic idea should be familiar from denotational semantics. The semantics of a potentially infinite structure like the one denoted by `(repeat 1)` for the definition

```
repeat x = x : repeat x
```

is a set of values whose least upper border is the infinite value rather than that infinite value itself:

$$[\![\mathit{repeat}\ 1]\!] = \{\bot, 1 : \bot, 1 : 1 : \bot, \ldots\}$$

A "strict semantics with oracle" can be understood as a non-deterministic choice of one element of the set as result of the evaluation of `(repeat 1)`. For debugging we choose exactly that element that corresponds to how far the expression was evaluated during the traced program execution. If, for instance, we have traced

```
main = take 2 (repeat 1)
```

we choose $1 : 1 : \bot$ as the semantics of `(repeat 1)`. This means in particular that we can have different choices, should `(repeat 1)` be called in different contexts during the program's execution.

3.1 Representation of Computations

As metinoed above, computations are categorized into three basic kinds of steps, as shown in Figure 3. Simple steps denote for instance a function unfolding or the binding of a free variable, forks denote the non-deterministic branchings induced by logic search and short cuts embed subcomputations, i.e. reductions *inside* the

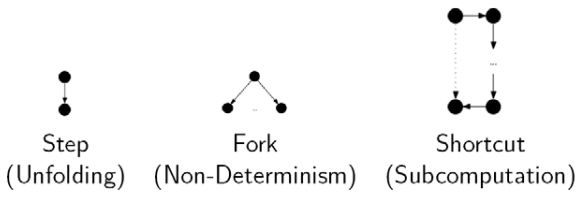

Fig. 3. The three kinds of Steps

given term. Each computation is terminated when it produces a value. For reasons developed in the next subsection, we also need to represent invalid computations and to augment each value with a computation state. Thus, we have:

```
data Computation step state = Deadend
  | Goal state
  | Step step (Computation step state)
  | Fork [Computation step state]
  | Sub (Computation step ()) (Computation step state)
```

There is good reason to have the content of a single step as a type variable. Many views can be formulated without any knowledge of what these steps consist of, as long as there is a way to represent them. Therefore we can have different definitions of a step depending on the strategy we want to represent and the detail level we would like to include. As an example of what a single step consists of, we might define:

```
type Narrowing state = Computation NarrowingStep state
data NarrowingStep = Unfold Term | Bind Int Term | Fail
data Term = Term String [Term] | Var Int Term | Unevaluated
```

This is enough for value oriented tools, whereas operational oriented tools might need to include more information like suspending goals.

Example 3.1 The evaluation of `main` in example 2.1 can be represented as follows, where the value `Unevaluated` is abbreviated as `_`:

```
Step (Unfold (Term "main" [])) (
Step (Unfold (Term "eq" [Term "add" [Var 1 _,Var 1 _],Term "Z" []])) (
Sub (
  Step (Unfold (Term "add" [Var 1 _,Var 1 _]))
  Fork [Step (Bind 1 (Term "Z" [])) (
        Step (Unfold (Term "add" [Term "Z" [],Term "Z" []])) (
        Step (Unfold (Term "Z" [])) (Goal ())))))
       ,Step (Bind 1 (Term "S" [_])) (
        Step (Unfold (Term "add" [Term "S" [_], (Term "Z" [])])) (
        Step (Unfold (Term "S" [_])) (Goal ())))]
Fork [Step (Unfold (Term "eq" [Term "Z" [],Term "Z" []])) (
      Step (Unfold (Term "True" [])) (Goal ()))
     ,Step (Unfold (Term "eq" [Term "S" [_],Term "Z" []])) (Step Fail (Goal ()))])))
```

which can be shown to the user in different ways, for instance in form of two independent proof trees, cf. also Figure 2:

```
main                              main
eq (add _A _A) Z                  eq (add _A _A) Z
/add _A _A                        /add _A _A
|_A\Z                            |_A\S _
|add Z Z                         |add (S _) (S _)
\Z                                \S _
eq Z Z                            eq (S _) Z
True                              FAIL
```

3.2 Generating Computations

The definition of computations above allows to generate, combine and process computations in a monadic programming style. Computations are a combination of list and state monads. As is well known, list monads are very expressive for nondeterminism and a state monad is useful to abstract from information which has to be updated regularly during computations. In our case, this information includes for instance the path for which a given subgraph has to be interpreted. (Cf. the discussion of the path concept above.)

The introduction of dead ends has the purpose of making interpretations satisfy the additional axioms of plus on monads, see below. This is also very helpful when implementing interpretations. When dead ends are added, we have to exchange the original constructors `Step`, `Fork` and `Sub` with constructing functions `step`, `fork`, `sub`, which make sure that dead ends eliminate a whole subway up to a next fork:

```
step :: a -> Computation a b -> Computation a b
step x w = if noDeadend w then Step x w else Deadend
sub :: Computation a () -> Computation a b -> Computation a b
sub x w = if noDeadend x && noDeadend w then Sub x w else Deadend
```

The function to construct forks makes sure that each fork has at least two subways:

```
fork :: [Computation a b] -> Computation a b
fork ws = mkFork (filter noDeadend ws)
  where mkFork []       = Deadend
        mkFork [x]      = x
        mkFork (x:y:xs) = Fork (x:y:xs)
```

Relative to these constructing functions, the following functions on ways satisfy the monadic axioms:

```
return = Goal
(Step x w) >>= b = step x (w >>= b)
(Fork ws)  >>= b = fork (map (>>= b) ws)
(Sub d w)  >>= b = sub d (w >>= b)
Deadend    >>= _ = Deadend
Goal o     >>= b = b o
```

Computations also satisfy the additional axioms of monad plus:

```
mzero = Deadend
mplus Deadend     w  = w
```

```
mplus (Goal o)    w  = if noDeadend w then w else Goal o
mplus (Step x w1) w2 = step x (mplus w1 w2)
mplus (Fork ws)   w2 = fork (map (flip mplus w2) ws)
mplus (Sub d w1)  w2 = sub d (mplus w1 w2)
```

It is straightforward to ensure that the monadic laws for these definitions indeed hold with respect to the constructing functions.

The huge advantage of this technique lies in the way it allows to abstract from the details of both the non-determinism and the manipulation of the state. For instance, if we interpret a given node of the trace graph, we can proceed like this:

```
interpretNodes :: [TraceGraph] -> State -> Narrowing State
interpretNodes [Node _ _ _ successors info] =
  interpretInfo info >>= interpretNodes successors
```

We do not have to care about whether the interpretation of the successors yields a deterministic sequence of steps or if there will be forks in the result. The operator `(>>=)` automatically makes sure that the interpretation of the successor is added to all branches when necessary.

Likewise, if we wish to make sure that the node we interpret is compatible with the current path (which is part of the state), we can define like this:

```
interpretNodes :: [TraceGraph] -> State -> Narrowing State
interpretNodes [Node _ p _ successors info] =
  ensurePath p >>= interpretInfo info >>= interpretNodes successors

ensurePath :: Path -> State -> Narrowing State
ensurePath p st = if p <= path st then return st else Deadend
```

This basic framework to implement interpretations makes it very convenient (if not possible in the first place) to define interpretations.

4 Summary

We have given an account of the current state of the project to unify different approaches to debugging into a single methodology. The flexibility needed to realize different aspects of debugging within a single framework comes from dividing the process into several steps:

(i) Trace the execution of the program to collect the relevant data for all different approaches.

(ii) Interpret and preprocess the recorded data to provide a simple yet versatile interface to the run-time information.

(iii) Based on the interface it is easy to create views on the recorded data, such that many of the tools successfully employed in debugging can be quickly integrated into the setting.

This paper was concerned with step (ii) of the above. We have described how the data recorded by program tracing is made available for libraries in the functional logic language Curry. We have then presented a proposal for a simple interface to this data. The main idea was that it suffices to represent the executed program as a sequence of computation steps, categorizing the steps into a) simple steps b) subcomputations and c) non-deterministic branchings. The different interpretations needed to cover value oriented techniques like declarative debugging as well as operational oriented techniques like symbolic profiling is only a question of how the computation is broken into subcomputations.

After describing the representation of computations we gave an account of how these representations can easily be generated and processed in a monadic programming style. Overall we have shown how different advanced features of Curry can be used to lift the low level information contained in the execution trace to a higher abstraction. Advanced programming techniques work together in order to create a framework in which interpretation for traces can elegantly be formulated. Future work includes giving an overview of the different strategies and views we realized using this framework.

References

[1] Albert, E., M. Hanus, F. Huch, J. Oliver and G. Vidal, *Operational semantics for declarative multi-paradigm languages*, Journal of Symbolic Computation **40** (2005), pp. 795–829.

[2] Antoy, S., M. Hanus, J. Liu and A. Tolmach, *A virtual machine for functional logic computations*, in: *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)* (2004), pp. 169–184.

[3] Antoy, S. and S. Johnson, *Teabag: A functional logic language debugger*, in: *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)* (2004), pp. 4–18.

[4] Booth, S. P. and S. B. Jones, *Walk backwards to happiness - debugging by time travel*, in: *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG)*, 1997, pp. 171–183.

[5] Braßel, B., O. Chitil, M. Hanus and F. Huch, *Observing functional logic computations*, in: *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)* (2004), pp. 193–208.

[6] Brassel, B., S. Fischer and F. Huch, *A program transformation for tracing functional logic computations*, in: *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)* (2006), pp. 141–157.

[7] Braßel, B., S. Fischer and F. Huch, *A program transformation for tracing functional logic computations*, in: *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)* (2006), pp. 141–157.

[8] Braßel, B., M. Hanus, F. Huch, J. Silva and G. Vidal, *Run-time profiling of functional logic programs*, in: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)* (2005), pp. 182–197.

[9] Braßel, B., M. Hanus, F. Huch and G. Vidal, *A semantics for tracing declarative multi-paradigm programs*, in: *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)* (2004), pp. 179–190.

[10] Caballero, R. and M. Rodríguez-Artalejo, *Ddt: a declarative debugging tool for functional-logic languages*, in: *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)* (2004), pp. 70–84.

[11] Cameron, M., M. García de la Banda, K. Marriott and P. Moulder, *Vimer: A visual debugger for mercury*, in: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)* (2003), pp. 56–66.

[12] Chitil, O., C. Runciman and M. Wallace, *Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs*, in: *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)* (2001), pp. 176–193.

[13] del Vado Virseda, R., *A demand-driven narrowing calculus with overlapping definitional trees*, in: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)* (2003), pp. 253–263.

[14] Ennals, R. and S. Peyton Jones, *Hsdebug : Debugging lazy programs by not being lazy* (2003).

[15] Gill, A., *Debugging Haskell by observing intermediate datastructures*, Electronic Notes in Theoretical Computer Science **41** (2001).

[16] González-Moreno, J., M. Hortalá-González, F. López-Fraguas and M. Rodríguez-Artalejo, *An approach to declarative programming based on a rewriting logic*, Journal of Logic Programming **40** (1999), pp. 47–87.

[17] Hanus, M., S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre and F. Steiner, *PAKCS: The Portland Aachen Kiel Curry System*, `http://www.informatik.uni-kiel.de/~pakcs/` (2006).

[18] Hanus (ed.), M., *Curry: An integrated functional logic language (vers. 0.8.2)*, Available at `http://www.informatik.uni-kiel.de/~curry` (2006).

[19] Huch, F. and P. H. Sadeghi, *The interactive Curry observation debugger COOiSY*, in: *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)* (2006).

[20] López-Fraguas, F. and J. Sánchez-Hernández, *TOY: A multiparadigm declarative system*, in: *Proc. of RTA'99* (1999), pp. 244–247.

[21] Lorenz, B., "Ein Debugger für Oz," Master's thesis, Fachbereich Informatik, Universität des Saarlandes (1999).

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

[23] Nilsson, H. and J. Sparud, *The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging*, Automated Software Engineering **4** (1997), pp. 121–150.

[24] Peyton Jones, S., editor, "Haskell 98 Language and Libraries—The Revised Report," Cambridge University Press, 2003.

[25] Sansom, P. and S. Peyton Jones, *Formally based profiling for higher-order functional languages*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 334–385.

[26] Shapiro, E., "Algorithmic Program Debugging," MIT Press, Cambridge, Massachusetts, 1983.

[27] Smolka, G., *The oz programming model*, in: J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments* (1995), pp. 324–343.

[28] Somogyi, Z. and F. Henderson, *The design and implementation of mercury*, Slides of a tutorial at JICSLP'96 (1996).

[29] Sparud, J. and C. Runciman, *Tracing Lazy Functional Computations Using Redex Trails*, in: *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)* (1997), pp. 291–308.