Searching Bugs by Visualizing Computations[*]

Bernd Braßel

May 15, 2006

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 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.*

As a very simple example we can see that at run time it is not easy to see how a run-time failure came to pass:

```
main = print (last (take 10 (repeat
                    (head (tail [0])))))

repeat x = x:repeat x

take n (x:xs) | n==0      = []
              | otherwise = x:take (n-1) xs

last [x] = x

last (_:(x:xs)) = last (x:xs)
```

This example is written in the language Curry, which is a functional logic language, employing advanced features like higher order, lazy evaluation and concurrency by constraint solving. The exact semantics of the given functions is not so relevant, apart from the meaning of the `head` and `tail`. Both functions are selectors on non-empty lists, e.g. `(head [0]) = 0` and `(tail [0]) = []`. Consequently, neither function is defined on the empty list. The point of the example is that there is a run-time error for `(head (tail [0])) = head []`, which is undefined. In an imperative language the run-time system would be able to give a good error message for this example by simply printing the function stack. This stack would show a connection from `print` over `last, take, repeat` to the undefined call to `head`, giving the programmer a good idea where to look for the bug. In the context of lazy evaluation, in contrast, the run-time system is not able to give a comparably good message. Printing the function stack miserably amounts to:

```
at print
non-exhaustive patterns in function head
```

There can be no mention of `last, take, repeat` because in a lazy context their task is done early and their further evaluation is suspended.

But laziness is not the only hindrance to understand what is going on in the run-time system. Higher order also has a great impact on the "blackness of the box." To see this, consider the slightly altered example in a strict declarative language:

```
main = print $ last $ replicate 10
                     $ head $ tail $ [True]


f $ x = f x

replicate n x | n==0      = []
              | otherwise = x:replicate (n-1) x
```

Executing this program would produce the stack:

```
at $
at $
at $
non-exhaustive patterns in function head
```

This stack is just as uninformative as the lazy one before.

Printing execution stack is only the tip of the iceberg. When debugging real world applications, there is of course much more need of ways to "look into the black box."

2 Our Approach

There is a way to turn the draw back into an advantage. We have extended an approach to debugging for the functional language Haskell by features to cope with logic languages. The basic idea is as follows:

- A program transformation instruments a given program such that
- its execution records a trace of what was going on in the run-time system.
- This trace can be interpreted in different ways,

[*]This work has been partially supported by the DFG under grant Ha 2457/5-1.

i.e., degrees of abstraction from the actual operational semantics.

- Each interpretation can be browsed by different views, giving the user the possibility to choose the kind of information he is interested in.

For instance, the very simple *failstack* view will – when employed with the interpretation *leftmost innermost with oracle* on (a mix of) the above example(s) – produce the following output:

```
main
at (print $)
at (last $)
at (take 10 $)
at (repeat $)
head []
FAIL
```

Note, that `main` is also included in the output, giving the proper context from the beginning of the execution, but omitting successful sub derivations like the one of `(tail [0])`. Like the other views, *failstack* is adaptable. For instance, one can limit the depth up to which argument terms are printed, keeping the output readable also for the execution of complex programs.

More sophisticated views allow the user to interactively browse the program's interpretation, enabling him to open and close sub derivations and non-deterministic choices at will. When he suspects an error in his program's logic he will choose an interpretation like the *innermost* mentioned above. When his problem seems to be rooted in the operational semantics of his program, e.g., concerning efficiency, the search strategy or sharing, he will rather choose to browse an *outermost* interpretation of his program although eventually using the same interactive view to browse it.

3 A Glimpse at the Implementation

The trace produced by an instrumented program is not human readable. It represents – by means of a vast number of numerical references – a graph. This graph has a direct connection to the executed program. For instance, in Figure 1 contains a small cut-out of the trace for the above example. There are two kinds of

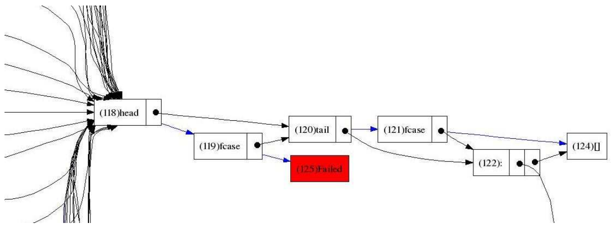

Figure 1: Section of the Trace Graph

arrows visible. Arrows with a black dot are argument pointers, whereas arrows without dot identify a reduction. For example, in the clip one can see that function `head` was called with a call to `tail` as argument. `tail` reduces to a pattern matching ("`fcase`") which then reduces to `[]`. Function `head` also performs a matching which fails. There is a whole forest of nodes referencing the head function. This is because `head` is the argument throughout the whole recursive decent of functions `take` and `repeat`.

Although we are able to make a connection between execution and program, the graph is not something we whish to show to the user. The more abstract concept of an interpretation is in terms of *steps*. There are three kinds of steps as depicted in Figure 2

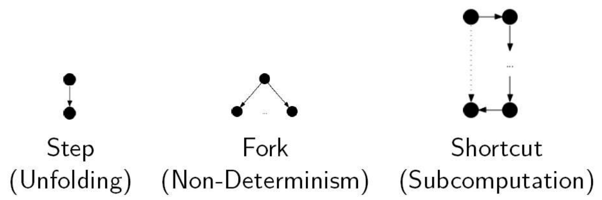

Figure 2: The three kinds of Steps

Simple steps denote a function unfolding, forks denote the non-deterministic branchings induced by logic search and shortcuts embed sub computations. Furthermore, in interpretations the pattern matching is only used to compute the interpretation but not included explicitly in the result.

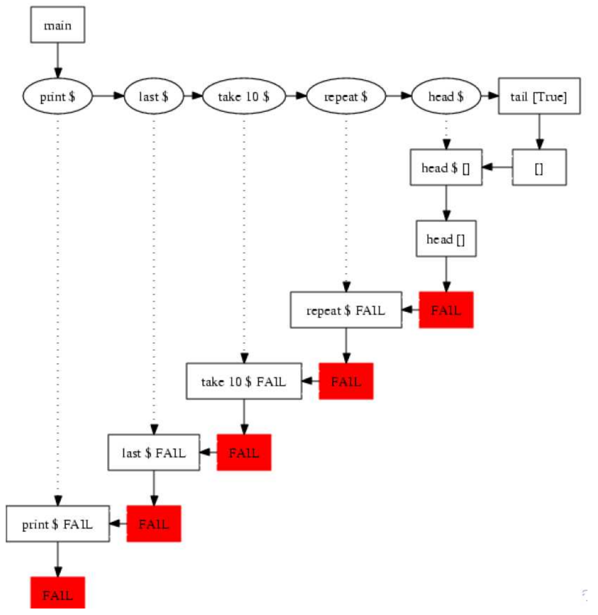

Figure 3: Innermost Interpretation of the example

For the example the result is shown in Figure 3. It is easy to see how to extract useful information from this view. The *failstack* for the above example is obtained by taking the shortest path from main to the first failure node.