Using Trace Data to Diagnose Non-Termination Errors [*]

Mike Dodds & Colin Runciman, University of York

March 5, 2006

Abstract:. This paper discusses BLACK-HAT and HAT-NONTERM, two tools for locating and diagnosing non-termination errors in Haskell programs. Both of these tools give a small trace which is intended to illuminate the cause of the non-termination error. BLACK-HAT analyses programs which contain *black holes*, a particularly restricted kind of non-termination error, while HAT-NONTERM applies the approach used in BLACK-HAT to more general non-terminating programs. This paper discusses the traces generated by BLACK-HAT and HAT-NONTERM, as well as the approach used to generate these traces.

1 Introduction

Non-termination errors are one of the two kinds of error that can cause Haskell programs to fail. In general, non-termination errors cannot be detected at run time, and non-terminating programs must be terminated by hand. The source of the error may also be difficult to locate and fix in a large program. Section 3 below discusses discuss HAT-NONTERM, a tool for generating traces from non-terminating programs.

Black holes are a kind of simple non-termination error that *can* be detected at run time. A black hole occurs when the process of reducing a variable[1] to its value results in an expresssion which contains the variable. Black holes can be extremely simple, as can be seen from this example:

```
a = b + 2
b = a * 3
```

It should be clear why the variable a cannot get a final result: getting the value of a requires a value for b, which requires the value of a. Section

[*]Included as part of the University of York technical report HAT *Day 2005: work in progress on the* HAT *tracing system for Haskell*

[1]that is, a function of arity zero

2 below discusses Black Hat, which is intended to locate and diagnose these black hole errors.

2 Tracing Black Holes

BLACK-HAT is a tool for analysing programs that have failed as a result of a black hole. Programs compiled with HAT support generate an Augmented Redex Trail (ART) file which describes the execution of a program. BLACK-HAT uses this ART file to locate and diagnose the causes of black holes, by giving the series of reductions which lead to the reoccurrance of a variable. To begin with a simple example, let us consider the small function `a`, discussed above. BLACK-HAT gives us the following analysis:

```
---- black-hat: simple-bh ----
> print a
> b + fromInteger 2
> a * fromInteger 3
```

BLACK-HAT's output shows a series of expressions generated by the program. Each line holds the result of rewriting one of the previous line's subexpressions. The rewritten subexpression is highlighted in each expression. We call this series of reductions and subexpressions the *black-hole path*.

The analysis for the `ab` program shows that `a` reduces to an expression containing `b`, which then reduces to an expression containing `a` again.

For small programs the black-hole path can be constructed quite easily by hand, so let us consider slightly more complicated example. The program below is a faulty program for generating Hamming numbers:

```
ham = merge ham' ham''
ham' = map (3*) ham
ham'' = map (2*) ham
```

It should be clear that there are in fact two black-hole paths for `ham` — one from evaluating `ham'`, and one from `ham''`. Which which one will result in the error depends on the order of reduction in the program. BLACK-HAT generates the following analysis when applied to this program:

```
---- black-hat: ham-bh ----
> print ham
> merge ham' ham''
> map (fromInteger 2 *) ham
```

Because BLACK-HAT works with the ART graph, which is generated from the execution of the program, it can work out which of the two potential black holes actually caused the error. The analysis makes it clear that the black hole occurs through `ham''` — through the right-hand path through the result-subexpression graph.

BLACK-HAT successfully locates black hole errors in all of the examples which it has been tested with. With simpler programs, such as the ones discussed above, BLACK-HAT also makes the source of the error obvious immediately. More complicated programs make this more difficult, as it becomes harder to follow the sequence of reductions and subexpressions. Also, programs that use Haskell's list comprehension can be difficult to understand, as they are transformed into other functions before being compiled.

2.1 Black Holes in the ART Graph

The ART file produced by HAT corresponds to a rooted graph with the intermediate expressions generated while executing the program as nodes. Several different kinds of edges exist between expression nodes, but BLACK-HAT uses two in particular: the result edge which point from an expression to its result, and subexpression edges, which point from an expression to its subexpressions.

An expression node in an ART files is generated when an expression is evaluated at run-time. When an expression is being evaluated, before it's result has been generated, the expression's result pointer has a value of `entered`. However, in the case of a black hole, the execution terminates because an expression has been reduced to a variable which has also been entered. This means that the result pointer for the final executed expression points back to the variable which caused the black hole, and so the graph will contain a loop in the result and sub-expression edges. This loop corresponds to the black-hole path displayed by BLACK-HAT, and it can be located by a simple search of the ART graph.

3 Tracing Non-Termination Errors

HAT-NONTERM applies the same approach as BLACK-HAT to more general non-termination problems involving functions with an arity greater than zero. That is, it generates a small trace from the ART which illuminates the source of the non-termination error.

HAT-NONTERM assumes that the execution of a non-terminating program will consist of two general phases. Firstly, a non-terminating program will execute the bug-free 'head' of the program. Then the faulty, non-terminating 'tail' of the program will execute until the program is interrupted by the user. Because a non-terminating program can be run for an arbitrary amount of time in the faulty section of the computation, the tail of the computation can always be made large relative to the head. For the same reason, in any non-terminating program, a trace can be generated where the number of function applications is much larger than the number of defined functions.

Because of these two facts, a non-termination will contain large numbers of calls to the same functions. These calls will be recursive calls originating from previous instances of this function, although the function arguments may be different. This presents a way of displaying non-terminating functions. If the suspicious function involved in the non-termination can be identified, then the path from one instance of the function to the next can be displayed. The path between two calls to the same function may not always be the same, and so it would be a good idea to show the path between several calls to the function. We call this the *non-termination path.* For example, consider this faulty Fibonacci-number program:

```
fib 0 = 1
fib 1 = 1
fib n = fib n + fib (n-1)
```

HAT-NONTERM gives us the following analysis:

```
---- Hat Non-Term: fib-nt ----
suspicious function is fib in module Main
> print (fib (fromInteger 5))
> fib (fromInteger 5) | False
> fib (fromInteger 5) | False | False
> fib (fromInteger 5) + fib (fromInteger 5 - fromInteger 1)
> fib (fromInteger 5) | False
> fib (fromInteger 5) | False | False
> fib (fromInteger 5) + fib (fromInteger 5 - fromInteger 1)
```

This non-termination path correctly identifies the `fib` function as the faulty function, and displays the responsible sequence of subexpressions and results.

3.1 Non-Terminations in the ART Graph

The ART file explicitly records the fact that a program was interrupted by the user, rather than terminating normally. As with black holes, nodes that are being evaluated, or with subexpressions that are evaluating are marked `entered`. When a computation is interrupted by the user, all of the nodes that are marked `entered` are re-marked `interrupted`. There will also be a large number of other nodes with a result pointer which points through to an `entered` if the pointer is followed repeatedly. Taken together, these nodes will form a chain of expressions from the ART root to the expression which was interrupted by the user.

As was discussed above, the tail of a non-termination will consist of a large number of reductions involving a small set of functions. Some of these may be terminating functions, but unless the computation is interrupted inside one of these functions, the corresponding nodes will not be marked `interrupted`. This means that the majority of nodes which are marked `interrupted` will be the functions involved with the non-termination loop. The set of suspicious functions which are involved in the non-termination can therefore be identified by the search process.

This method of selecting suspicious functions can be complicated by non-terminating programs that call expensive terminating functions. If the program is interrupted, it is likely to be interrupted during the execution of the expensive function, and so this function will be marked as `interrupted`, even though the function is not part of the non-termination path.

Once the set of suspicious functions have been identified, HAT-NONTERM selects a single function for tracing. Several heuristics are possible. The simplest is to select the interrupted function that appears last in the ART file. Another possibility is to take the interrupted function that appears most often in the ART file. Unfortunately, both of these ideas will fail with non-terminations that call expensive terminating functions. The execution of such a program is likely to be interrupted inside an instance of the expensive function. This means that the last-interrupted heuristic will incorrectly select it as the suspicious function. Similarly, if the expensive function is large enough it may occur more often in the file than real non-terminating functions. The most-common heuristic would then also incorrectly select this function.

The solution used by HAT-NONTERM is to select the function that appears furthest apart in the ART file. The functions involved in the non-termination will not only be marked `interrupted` at the point of interruption; They will also be `interrupted` all the way through the non-terminating program tail. This is in contrast to an expensive terminating function called by the non-terminating loop, which will terminate in all but the final case, and so will only be marked `interrupted` right at the end of the ART.

Once a suspicious function has been selected, HAT-NONTERM looks for a path through the ART graph which contains the required number of interrupted instances of the selected function.

3.2 Evaluating HAT-NONTERM

HAT-NONTERM generally deals well with simple non-terminations. However, it is quite easy to come up with examples where the tool's analysis is obscure or nonexistant. HAT-NONTERM suffers from the same problem as BLACK-HAT, in that the sequence of subexpressions and reductions can be difficult to follow. The tool can also have problems with functions that are passed infinite data structures — if a normal function is passed an infinite data structure, it may not terminate, but HAT-NONTERM will give no indication of why the data structure is infinite.

Another problem with HAT-NONTERM's approach is that it fails to correctly analyse non-terminating programs that generate data-structures as they run. This is because these programs do not produce large numbers of `interrupted` nodes in the ART, as functions are constantly begin reduced to constructors. For example, consider the program:

```
repeat a = a : repeat a
```

Almost all of the instances of `repeat` will evaluate to a result where the constructor is a '`:`' node. Only the final instance of repeat will be interrupted when the program terminates. This means that HAT-NONTERM will fail to locate the non-termination in this program. One solution may be to follow the parent edges from the terminated expression up towards the ART root, and record the functions which are found along this path, but this approach has not yet been investigated.

It is interesting that HAT-NONTERM fails on 'structure-generating' non-terminating functions. These are quite different from non-terminating functions which do not generate data structures, and which therefore cause any function that evaluates them to fail. In contrast, Haskell's lazyness means that structure-generating non-terminating functions can sometimes be used safely in terminating programs. For this reason, it may be inaccurate to always describe such functions as erroneous.

4 Conclusions

BLACK-HAT is generally successful in locating the causes of black holes in test programs. However, it can be difficult to interpret the trace data that it produces for complicated programs. HAT-NONTERM successfully locates and traces some simple non-terminating programs, but it is quite easy to find programs for which it generates poor or nonexistant analysis. Much more detail and analysis of BLACK-HAT and HAT-NONTERM can be found in [1], along with a number of larger examples.

References

[1] Mike Dodds. Using trace data to diagnose non-termination errors. Final-year MEng project at York University, May 2004.