The GHC Heap Management “Bug.”

If we were to give a cute title to this “bug,” it would be “Lazy kids in the greedy toolbox.” We analyze a problem that arises when one moves from lazy data to strict or eager data. Given a term, f, applied to another term, f x, there are two possibilities of what one can do. One can reduce x to some normal form x' and then substitute x' into the body of f; this is called by-value, eager, or strict evaluation, and is use by most programming languages. Alternatively, one can substitute x directly into the body of f; this strategy is called normal order reduction, and has a (very) slight theoretical advantage. However, it is clear that this can be very inefficient since x could be copied multiple times, and then later each of those copies may need to be evaluated. Lazy evaluation makes normal order reduction computationally feasible by substituting a pointer into the body of f, and then later if x is evaluated it is evaluated once, and kept at the same end of the pointer. Then any other calls needs the value of x can just retrieve the already computed version of x. However, there are some disadvantages to lazy evaluation; in particular, it makes reasoning about the space behavior of programs hard. Indeed, Haskell language designer Simon Jones said that lazy evaluation might not be theoretically required.,

> Purity is more important than and quite independent of laziness. It is still unclear exactly how to add laziness to a strict language. For example, do we want a type distinction between lazy Int and strict Int[1].

Background:
We analyze a bug caused by an unintended functionality of GHC's heap management system. After a discussion with some Haskell developers, the bug we will present below is not really in GHC, but in the way programs treat data, and move from the lazy world of Haskell to the eager world of C. Further, such bugs will likely not be exploitable due to the way that Haskell interfaces to the operating system, and there is no interface to access memory. However, I am not a hacker, and there may be some way to abuse this assumption that Haskell and the operating system will interact correctly. Indeed there are a few issues that do not fit any model, and I leave them on the table for future discussion.

GHC[2] compiles programs for the Haskell programming language to native code, C, and llvm. Haskell programs are represented in the lambda calculus, and are compiled to a “pointerless” abstract machine[3]. For our purposes, this means that we do not have access to the state, variables, stack, or heap of a running program[4]. Further, Haskell does not compile functions in a “traditional” way; Haskell treats functions as objects to be reduced, so running Haskell programs through gdb does not give predictable, expectable, or reliable results[5]. That is, tools for analyzing running Haskell programs are rare, immature, and limited.

1 “Wearing the Hair Shirt: a Retrospective on Haskell.” Simon P Jones, POPL, 2003.
2 http://www.haskell.org/ghc/
3 The Spineless Tagless G-Machine. http://research.microsoft.com/apps/pubs/default.aspx?id=67083
4 GHC debugger. http://www.haskell.org/ghc/docs/7.0.3/html/users_guide/ghci-debugger.html
5 It is possible to use gdb on GHC code, but it is not possible to simply set a breakpoint and run. http://hackage.haskell.org/trac/ghc/wiki/Debugging/CompiledCode?redirectedfrom=DebuggingGhcCrashes

Analysis:

The bug was first encountered in a program called Leksah [6]. I tried opening a large (but reasonable) file, and my window manager crashed. I rebooted, and tried opening the file again. Again the window manager crashed. Thus, I became suspicious, and initially attributed the problem to the binding between GHC and GTK. We will show how the bug arises just before this binding, and we will separate the bug from GTK, show that the problem is on the Haskell side, then we will carefully watch our system and see how Haskell can be used to cause programs to crash. We will narrow in on the bug, and show exactly where the problem is, and at the same time offer a generic fix. As to whether the bug is exploitable, the answer is probably not; in fact, the bug is a bit boring, but is overlooked.

6 http://www.leksah.org/

Experiment 1: Crash the operating system.
Hypothesis: Running a Haskell program can cause other programs to crash..
Methods: We used the following Haskell program to create a very large file.

```
main :: IO ()
main = do
  let str =
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n"
  writeOver str
  where
    writeOver str = do
      appendFile "BigOne.txt" str
      writeOver str
```

We let the program run for a few seconds which generated a file > 100 MB in size. Then we performed the following steps.

1. Start the program Leksah
2. Goto the Open file dialog in Leksah's toolbar
3. Select BigOne.txt
4. Wait

We then observed that windows stopped responding, the text area of editors greyed out, and we could not even Ctrl-Alt-F* to get to a terminal. We had to do a cold shutdown of the computer to regain control. We then replicated this experiment several times with 100% accuracy, every experiment had the same outcome.
Results: We confirm that loading a really big file into Leksah causes the operating system to crash.

However, we cannot attribute a cause to the crash at this point, but we have warranted further investigation. To separate out the functionality of the text editor component of Leksah, we wrote a small text editor using the gtk2hs Haskell-GTK interface to see if the results are the same.

Experiment 2: Crash the operating system part 2.
This time we crafted the simplest possible text editor: it can only load and save files. We use this to isolate the cause of the crash in Experiment 1 to an aspect of interacting with the text editor component. We have attached the editor in an appendix; moreover, we used the same convention for loading the buffer as used in Leksah.
Hypothesis: Loading a really big file in our customized text editor will cause the system to crash.
Methods:
We repeat the first experiment on a stripped down text editor that performs only the open and save functions. We replicate the results multiple times.

Results: We confirm that loading a very large into the buffer of the custom editor causes the operating system to crash.

On the other hand, it could just be that evaluating the string causes system resources to be overused, and is slowing the system down and not crashing it. We really want to know if GHC runtimes are capable of causing other processes to not work correctly, not just that we have blocked the system. It would be necessary (but not necessarily sufficient) to find processes or programs that are crashed by the GHC runtime even after the GHC program is closed. This would demonstrate that another program's has been crashed by the GHC program.

Experiment 3: Recovery

Hypothesis: After loading a very large text file into the custom text editor and the system crashes, the system or program that crashed will be unable to recover.

Methods: Unfortunately, this test has an element of luck to it. In order to perform this test, we have to start the editor, load a large test file, then wait for the system to start crashing, then kill the editor. The problem is that if we wait to long, we are unable to kill the editor, but if we do not wait long enough, then the system will not become corrupt. Estimating the bottleneck of file reads to be about 40 MB/Sec on my computer[7] (thus GHC programs will access them even slower), letting the program try to open the file for 60 seconds should put enough data into memory to start crashing things. By trial and error we found 3 minutes was enough to force a cold shutdown. We test this hypothesis by running three programs alongside the text editor: Eclipse[8], Firefox[9], KDE4.6[10]. For the record, my eclipse installation has the Haskell plugin and the CVS plugin additionally installed, and was not installed on my computer, just run from the Home directory.

We performed five repetitions of the same test. Following is our procedure. Prestep: set all unnecessary KDE functionality (networking,bluetooth,printing, compositing) to not start at boot.

1. Reboot computer.
2. Log in to the KDE workspace.
3. Let KDE bootstrap.
4. Start Eclipse. Choose Default Workspace.
5. Start Firefox.
6. Run the text editor and detach the process ($ ./Sedit &).
7. Write down the process id PID
8. Load the really large text file
9. Run the bash program ($sleep 60; kill PID)
10. Go make tea, or become occupied for 10 minutes. This is to let the GHC program terminate by being killed, to let the GHC runtime system perform garbage collection, and to give the operating system a few cycles to clean things up.
11. Check if the applications are working correctly. For firefox, goto www.google.com, search “frog,” and select the wikipedia article. This should seem as fast as usual. Also drag the window in a few circles to see that it's painting functions are working. It should be obvious if KDE crashed, but in case open dolphin (file manager) and konqueror (web browser). For Eclipse, build and run the program in the default workspace.

Results: We confirmed our hypothesis that the GHC editor can affect running programs, even after the editor halts and the operating system continues running for several minutes.

For 2 out of the 5 tests, there were no observable problems. For one of the tests, KDE crashed. None of the windows could be moved, none of the menus worked, and the Ctrl-F2 application launcher did not respond. We were able to Ctrl-Alt-F2 to the command line, use typical linux programs (ls,pwd,cd,reboot). The final test is quite revealing. In two tests, Eclipse broke. I had htop running to view the memory load. I did not want to be able to attribute any error to the system running out of memory, so I was watching the output of htop to ensure I always had 1GB of RAM left. I noticed

7 Using hdparm.http://en.wikipedia.org/wiki/Hdparm
8 http://eclipse.org/
9 http://www.mozilla.com/en-US/firefox/fx/
10 http://www.kde.org/

something strange. Ten minutes after the GHC program was terminated by bash, the Eclipse application was still in a non working state. All the buttons, icons, and text areas disappeared. The application window would not drag, resize, or close. The application had to be shutdown manually. This confirms that the GHC program can cause other programs to crash in such a way that they cannot recover.

We only know that the GHC-produced program can cause other programs to crash. We do not know whether the problem is on the Haskell side, the GTK side, or the GHC-GTK bridge. We need to separate out where the program crashes from the three possibilities. Our current model suggests that when a Haskell program marshalls a string to an array in C, it can potentially consume too much memory. This suggests that GTK never gets invoked on the crash runs.

Experiment 4: Finding out what GHC is spending time on in this program.

Hypothesis: When a file is loaded into the custom editor, the memory will slowly increase until Haskell has finished evaluating the string to an array, and then the memory usage for the string in Haskell will drop and be picked up by C. When the file is saved, we will say a fast but steady increase in the memory used by Haskell as it inductively rebuilds the string, and then we will see this memory decrease as the string is written to memory.

Method: To test this hypothesis, we instrumented our Haskell program with profiling support. This allows us to see the runtime memory allocation by datatype of the Haskell Program. To do this, compile the Haskell program as follows

ghc –make -o ProgName Program.hs -prof -auto-all -caf-all

Which turns on profiling support, automatically adds symbols to closures of expressions[11], and shows the costs of constant applicative form reduction[12]. We then run the program with

./ProgName +RTS -h[d|y] -i0.001

This tells the runtime program to take a snapshot of heap usage every 0.001 seconds. h tells the runtime to collect space allocation data. Depending on whether d or y is chosen after h, we either get the constructors or the types that the runtime is spending it's resources on. The procedure we follow in the test is:

1. Compile and run the custom editor as above
2. Load a large file (approximately 30 MB) into the editor
3. Wait for the load to complete and then wait a few seconds
4. Save the file and wait for the save to complete
5. Exit the editor
6. Run hp2ps ProgName.hp which returns a ps file. hp2ps is included with the compiler.

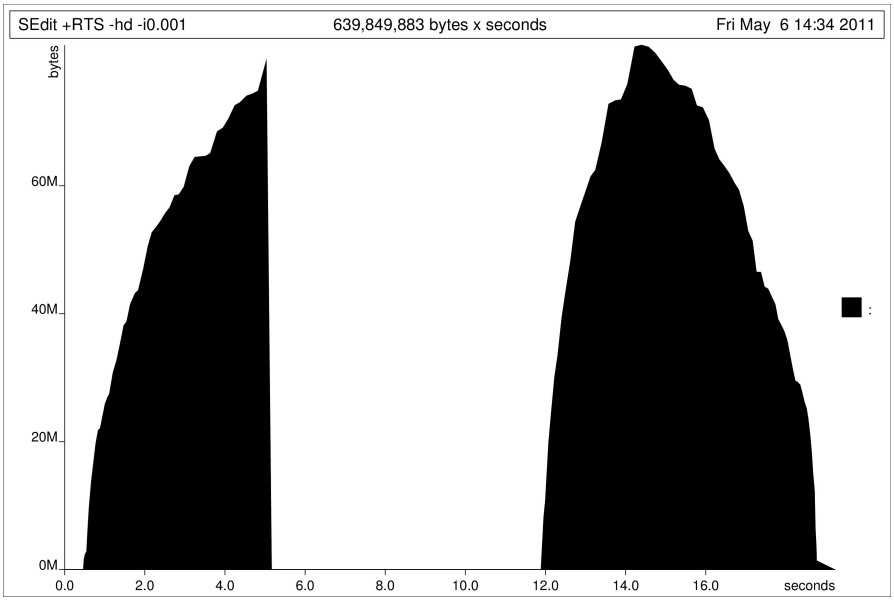
We replicated the test several times alternating between checking constructors and checking types.

---

11 When compiling lambda calculus to machine code, one passes through an intermediate language called the categorical abstract machine (CAM), the sequence eval compute dump machine (SECD), or Krivine's machine. A lambda term may in general have free variables, or locally free variables; however, to use the efficient abstract machine computations such as CAM, one can only work with combinators (terms that have no free variables); thus the compiler must generate a term with no free variables called the **closure** of the term. Typically a closure is generated by including the environment it is in (the list of bound variables and their types which will be instantiated when the top level term is). In a worst case scenario, the compiler will curry a fresh environment to the term to close it. Types and Programming Languages. Benjamin Pierce, 2002 MIT Press.

12 A **constant applicative form** is a technical way of saying that a piece of an expression can be lifted to the top level of a program and computed only once. This is also sometimes called lambda lifting. You must ask for these to be computed by the profiler because they are not part of the call graph, even though they are computed and take time. The Implementation of Functional Programming Languages. Simon Jones, 1987, Prentice Hall.

Results:



Drawing 1: Heap Profile Loading and Saving Strict

The above graph is representative of the trials despite whether checking constructors or types. Notice how the graph increases gradually as the file is loaded from memory into the Haskell program, and then abruptly cuts off; this is where the handoff takes place. Also, we confirm the second part of our hypothesis, the memory usage on save increases until the entire string is generated from C. Note that the memory is not returned abruptly like it left; this is because Haskell manages the resource and builds the string inductively.

This evidence suggests that our bug above can be totally attributed to the GHC compiled Haskell program. When the Haskell program evaluates the string, it can consume too much memory. GTK never gets called if the entire file is not loaded.

Experiment 5: A GHC program in isolation.

Hypothesis: A Haskell program compiled with GHC can cause the unning programs to become corrupt.

Methods: There is no place in Haskell that guarantees strictness, not even IO[13]. However, there is a mechanism for forcing complete evaluation of a datatype; in fact, the intent of the package DeepSeq[14] is to "force pending exceptions, remove space leaks, or force lazy I/O to happen." This package is part of the standardized library set for Haskell (regardless of the compiler used). The function we will use is

```
deepseq :: NFData a => a -> b -> b
```

NFData is supposed to mean that x :: NFData a, can have a normal form computed in finite time. However, for the instances they give, one needs to assert that the structure is finite. Then up to exceptions generated by evaluating x and y,

```
(deepseq x) z = (deepseq y) z = z.
```

To simulate the action of GHC fully evaluating a file and preparing it to be handed to C, we will simply use a function that fully evaluates a string. Here is an example that fully evaluates a string and then returns the string.

```
strictString xs = (deepseq xs) xs
```

We use this function to fully evaluate the > 100 MB file. To test whether we can recreate the effect of crashing running programs we attempt to crash Eclipse and Amarok. We choose Amarok since it is a large application that prints debug information at runtime. Our procedure follows. Note that we lowered the run times of programs heuristically to ensure that we always had approximately 1 GB of ram according to htop, so that one could not as easily attribute any errors we find to system overload.

For Eclipse:

1. Reboot
2. Download, unpack, and run a fresh version of Eclipse
3. Start Eclipse and load the default workspace.
4. Run the above Haskell program for 5 seconds.
5. Observe the state of Eclipse.
6. Observe the state of Eclipse at 10,15, and 30 minutes.

For Amarok:

1. Reboot
2. Start amarok in debug mode (amarok –debug).
3. Begin playing an audio file.
4. Run the above Haskell program for 5 seconds.
5. Observe the state of Amarok.
6. Observe the state of Amaork at 10,15, and 30 minutes.

Results:

For Eclipse:

We confirmed our hypothesis 2/5 times, while 3 times the window manager crashed. The application became unusable, the widgets all greyed out, the window became unresponsive, could not be moved. At 10,15,30, minutes the window could be moved but remained greyed out. Eclipse could be minimized.

For Amarok:

We confirmed our hypothesis 5/5 times with Amarok; the window manager never crashed. We

13 IO in Haskell is not uncomplicated. IO is the type IO (RealWorld# -> (a,RealWorld#)), where RealWorld# is a special strictly evaluated state which is meant to represent the environment in which Haskell is running.

14 http://hackage.haskell.org/packages/archive/deepseq/1.1.0.2/doc/html/Control-DeepSeq.html

were not able to get any relevant debug information from Amarok. However, Amarok gave other evidence that its was crashing. We had a song playing during the test. Around 4 seconds, the track continued to play; however, it had been replaced by odd noises. It actually sounded like farm-animal techno. Through the end of the song, the odd noise continued. However, 5/5 times when I selected a new song to play, the song played as normal. Further pausing the song for a few minutes did not fix the wierd noise. I replicated the same resulting noise and the result that loading a new song stopped the noise in two other music players Enna[15] and VLC[16].

A few points immediately come up. All the problems we have seen can be fixed by using DeepSeq in the appropriate place, either in the editor, in the GTK-Haskell bridge, or in the foreign function interface.

Using the DeepSeq package one can test if there will be a memory problem. Define

```
sLen [] = 0
sLen (x:xs) = deepseq t (1+t)
where t = sLen xs
```

Indeed, inserting the following

```
file <- readFile name
let lenF = sLen file
...
```

Causes a runtime exception to be made when the large file is loaded. Further, for normal files, sLen runs very fast. On a thousand trials, sLen averaged over 10 times faster than the standard length function. Indeed, since GTK2HS is bridging data structures from Haskell to C, they could perform a similar analysis which will throw exceptions instead of allowing the program to consume too much memory and interfere with other programs. At an even lower level, any bridge that leaves Haskell and connects to another language (with the exception of maybe Clean), should have such a sanity check performed. I have left this suggestion to gtk2hs, but have not received any feedback.

There are two major shortcoming, or pieces of information that are not well understood. It was suggested that the problem is that the operating system slows to a crawl, and either the resources are too sparse or changing too fast for the operating system to direct the programs efficiently. However, we left 1 GB in RAM at all times, and further (at least) half an hour after the offending program was terminated, the applications were still in a crash state. I cannot explain this from my data, and more experimentation needs to be done to understand why things crash. There is also one further point that I would like to bring up. GHC compiles directly to machine code; however, it can compile to C as an intermediate step. One advantage is that one can use gcc's optimizations, but another advantage is that one also has access to gcc's error catching mechanisms. However, in our examples, compilation failed when trying to compile through C. C caught an error that Haskell is sending bad, lazy data to a strict function, and it would not compile. The reason gcc is catches this error is also not understood.

15 http://enna.geexbox.org/
16 http://www.videolan.org/vlc/