Space leaks exploration in Haskell

Seminar Report

Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology (Honours)(*)
Masters of Technology(+)

Students:
Sumith Kulal(*)
Rupanshu Ganvir(*)
Suresh Sudhakaran(+)

Guide:
Prof. Amitabha Sanyal

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

A *space leak* occurs when a computer program uses more memory than necessary. Advanced language features such as lazy evaluation or closures lead to more complex memory layout, making it harder to predict what memory looks like, potentially leading to space leaks. Haskell is vulnerable to space leaks and we make this our topic of study. In this report, we present an extensive literature survey on space leaks in Haskell. We stress on classification of space leaks into different classes and try to reason about them individually. We present a large pool of example space leaks, their complete analysis and fixes. We also touch upon relevant blog posts present on the internet. Corresponding papers that discuss possible fixes have also been presented. Towards the end of this report we present a few ideas for future work that we picked up as we progressed along the survey. This is a first of the kind survey done on space leaks.

Acknowledgement

We wish to express our sincere gratitude and indebtedness to our guide, Prof. Amitabha Sanyal for his constant support and guidance throughout the project. His enthusiasm for functional programming has rubbed off on us. All of us feel that Haskell will be an important part of our programming life from here on. We are also thankful to Edward Z Yang, Neil Mitchell and Prof. Colin Runciman for taking time out of their busy schedule and answering all the questions we had. We are also thankful to Prasanna Kumar for being present in some meetings and giving us his inputs.

Sumith, Rupanshu, Suresh
CSE, IIT Bombay

Contents

Chapter 1

Introduction

"Beware of little expenses. A small leak will sink a great ship."

- Benjamin Franklin

A *space leak* occurs when there exists a point in the computer program where it uses more memory than necessary. Hence, a space leak causes the program to use more space than one would expect. Our primary language of study would be Haskell. Many researchers have pointed out that space leaks are a common problem in functional languages with lazy evaluation[1] [2][3]. Note that a space leak is different from a memory leak. In contrast to memory leaks, where the leaked memory is never released, the memory consumed by a space leak is released, but later than expected.

1.1 Motivating simple examples

Here, we present two real-life examples to throw some light on the two basic classes of space leaks that we are going to discuss further in the report. We shall present example programs in Haskell for these classes in the latter chapters of the report.

Say Mr. X is very strong in English vocabulary except for words starting with the alphabet X. Unable to comprehend complete pieces of text, Mr. X buys a dictionary for himself and uses it to feel more comfortable. In this situation, Mr. X doesnt need the rest of the dictionary but needs only pages that are relevant (pages with alphabet X). Hence, the whole dictionary occupies more *space* on his bookshelf than the few pages pertaining to the alphabet X which serves the purpose as well. Note here that if Mr. X had cleverly known what his *garbage* was and had *collected* it earlier then it would have saved him some *space*. This kind of space leak falls into the first category which we refer to as strictness class of examples throughout

the report.

Say Ms. Y is health conscious and bought a lot of fruits to make herself a mixed fruit salad. Given that Ms. Y eventually makes the salad, the time between her buying the fruits and making the salad, the fruits occupy *space* on her kitchen shelf. Note here that if Ms. Y *strict* on her diet and realized that she will eventually make the salad, she could have *evaluated* this salad early and hence this would have saved her *space*. This kind of space leak falls into the second category which we refer to as liveness class of examples throughout the report.

1.2 Strictness - Forcing evaluation

Consider the strictness class of space leaks, a space leak occurs when the program contains an expression which grows regularly but the evaluated value or the final result would not grow in space. Making some variables strict and forcing evaluation earlier than the normal evaluation would eliminate such space leaks.

In this section, we intend to introduce a few terminologies used in evaluation and describe how to force evaluation.

A functional program can be considered as an expression, whose execution is the same as evaluating that expression. At any stage, the program will contain an expression that can be evaluated at that stage. Such expressions are called reducible expressions, or *redex*. The execution of a program consists of repeatedly selecting a redex and reducing it. For instance:
(+ (* 6 4) (- 8 2)) has two redexes, (* 6 4) and (- 8 2).

Let us first define how an expression is evaluated:
- An expression is in normal form if it cannot be evaluated further.
- An expression is in WHNF (weak head normal form) if the outermost part does not require further evaluation.

As an example, $(1 + 2) : []$ is in WHNF since the outermost part is (:), but it is not in normal form since the (+) can be evaluated to produce 3. It is pretty clear that normal form is a stronger definition and all values in normal form are by definition also in WHNF.

```
func x = do

 print x
```

Printing x will evaluate x to normal form, so the function output will evaluate x to normal form just before printing it. To force evaluation to

WHNF, Haskell provides strictness annotations with the commonly used bang patterns extension. Bang patterns is an extension that will evaluate specific arguments to weak head normal form regardless of the pattern match performed.

```
func !x = do
      ...
       print x
```

Adding an exclamation mark as a strictness annotation will force evaluation of x sooner. Now, x is evaluated to WHNF as soon as the argument is passed and then the function output will evaluate x to normal form just before printing it.

Given that introducing strictness avoids space leak, one might feel as making all values strict. But this falls into the age-old debate of whether one needs lazy evaluation at all. Lazy evaluation is a tradeoff with space leaks being a disadvantage, but there are also many advantages. We do not find the need to enumerate the advantages of lazy evaluation. Hence, avoiding lazy evaluation is not a solution to this problem. We need to fix the space leak issue the lazy paradigm itself.

1.3 Space leaks in other programming languages

Garbage-collected languages other than Haskell are also susceptible to space leaks. Many languages support closures. One popular language that makes extensive use of closures is JavaScript. Below we present a Javascript example of space leak [4].

```
request.onreadystatechange = function(){
  ...
  context.decodeAudioData(request.response, function(audio){
    document.getElementById("status").onclick = function(){
    alert("MP3␣is␣" + audio.duration + "␣seconds␣long");
  }
  });
};
...
```

This function is a part of a program that uses XMLHttpRequest API to load an MP3 file, then uses the Web Audio API to decode the file. Using the decoded audio value, you can add an action that tells the user the MP3s duration whenever a status button is clicked.

The "status" button has an onclick event that runs the following code:

```
alert("MP3␣is␣" + audio.duration + "␣seconds␣long");
```

This code references the audio object, which stores the audio data which takes at least as much memory as the original MP3. The only thing ever accessed, however, is the duration field, which is a number and occupies space as much as an integer. The result is a space leak. This leak is very similar to the strictness based space leak as the code just uses the expression audio.duration but retains the whole audio in memory. If evaluation was forced earlier, then this could have been avoided. For example:

```
var duration = audio.duration;
document.getElementById("status").onclick = function(){
   alert("MP3␣is␣" + duration + "␣seconds␣long");
};
```

Now the duration is computed before the onclick event is registered, and the audio element is no longer referenced, allowing it to be garbage-collected.

1.4 Detecting space leaks

The GHC compiler provides a number of built-in profiling tools that help in detecting the source of space leaks. A more detailed section on profiling shall be covered in the later chapters.

Detecting space leaking fragments of code in a large program is extremely challenging. One thing to contrast on is memory leaks and space leaks. In space leaks, considerable space is referenced but is eventually freed on termination. A program with a space leak will often reach its peak memory use in the middle of the execution, compared with memory leaks that never decrease. A standard technique for diagnosing memory leaks is to look at the memory after the program has finished. However, this technique is not applicable to space leaks.

As of now, the literature *lacks* a dynamic/static analysis approach for the same.

1.5 How troublesome are space leaks?

Advanced language features such as lazy evaluation or closures lead to more complex memory layout, making it harder to predict what memory looks like, potentially leading to space leaks. There have been changes to compilation techniques and modifications to the garbage collector and profilers to pinpoint space leaks when they do occur.

Space leaks are often detected relatively late in the development process, and often only in response to user complaints of high memory usage. If space leaks could be detected earlier say for example as soon as they were introduced then they would be easier to fix and would never reach end users.

Despite all the improvements, space leaks remain a *thorn* in the side of lazy evaluation, producing a significant disadvantage. Pinpointing space leaks is a skill that takes practice and perseverance. Better tools could significantly simplify the process. Space leaks are indeed worrisome in production but not fatal at the word go.

1.6 Organization of the Report

The rest of the report is organized as follows: In Chapter 2, we introduce general concepts which are introduced as preliminaries. In Chapter 3, we present a large pool of examples and discuss their detailed profiling. In Chapter 4, we present a series of blog posts present on the internet pertaining to space leaks. In Chapter 5, we present few papers that discuss solutions to detect and possibly fix a space leak. Finally we conclude the report in Chapter 6 by presenting a few ideas for future work.

Chapter 2

Preliminaries

In this chapter we shall present some preliminaries that will aid in the understanding of subsequent chapters.

2.1 Strictness in Haskell

Haskell uses lazy evaluation. This means that the computation is deferred until it becomes absolutely necessary. We call this the evaluation *by-need* paradigm. Laziness takes away some control from the programmer over the order of evaluation. It can also introduce some drawbacks in performance. Delayed evaluation results in storing a large number of thunks consuming a lot of memory.

Some of these problems could be prevented if we determine the strict arguments and evaluate them right when the function is called.[5] Since these arguments are strict, they would have been evaluated at some point or the other, and doing it sooner reduces the memory usage.

GHC when run with $-O$ flag, performs such optimizations. It also provides *seq* and *Bang* patterns to the programmer to explicitly specify the arguments of a function that are strict.

2.1.1 Seq

The *seq* function is a means of introducing strictness in Haskell.[6] *seq* takes two arguments of any type, evaluates the first and returns the second. It is essentially strict in the first argument.

It does so by introducing a dependency between the first and second arguments. However this dependency is quite superficial and essentially only

forces the evaluation of the first argument.

For instance: if x is of type Integer, then following program snippet

```
seq2 x y = if (x == 0) then y else y
```

returns the value of y regardless of the value of x. The comparison of x with 0 is done just to force the evaluation of x in the lazy model. *Seq* is defined along the same lines, only the first argument can be of any datatype. The definition of seq should necessarily satisfy the following two properties:

$$seq \ \perp \ b \ = \perp \tag{2.1}$$

$$seq \ a \ b \ = \ b \tag{2.2}$$

where $\perp$ refers to an infinite/unsuccessful computation.

2.1.2 Bang Patterns

We can declare a function to be strict on an argument by placing a **!** before the argument. So when we match an expression e with a pattern !p, we first evaluate e to weak head normal form (WHNF).

```
f !x y = x + y
```

The above function is strict in the first argument but not on the second. Bang patterns are basically syntactic sugar for the *seq* function. When using bang patterns, the program should be compiled with the flag **-XBangPatterns**.

2.2 Profiling

Profiling refers to collecting information about a program's behavior as it executes. It is useful in understanding the resources used by the program and identifying parts of the program that should be optimized to improve overall performance.

GHC provides a profiling system to help monitor the time and space usage of a Haskell program. The points in the program where statistics are generated and collected are called cost centres. Every cost centre has a cost (in terms of time and space required for evaluation) assigned to it. By default all top level functions are considered cost centres when we use the *-fprof-auto* flag during compilation. We can specify cost centres in the program. To manually add another cost centre to a program we use

```
{-# SCC "name" #-} <expression>
```

SCC stands for Set Cost Center. If the program is not compiled with the *-prof* flag, then the SCC annotations are ignored.
While running a program with profiling, GHC attributes any time or memory allocation done at any point to the cost centre at that point. We can find the number of entries corresponding to the cost centre and the percentage of time spent and space allocated in that cost centre.

To profile a program,

1. Compile the program with the *-prof* and (optionally) *-rtsopts* options.
2. When running the program, specify the *-p* (for general profiling information) and *-hc* (for heap profile which can be converted to graph) flags within *+RTS -RTS*.
3. Use hp2ps utility to generate graph for the heap profile.

There are other useful options besides *-hc* to generated different types of profiles. [7]

-hy: Generates profile based on the type of data on the heap.

-hm: Generates profile based on the module containing code which generates the data.

-hd: Generates profile based on the closure description of the data on the heap.

We typically profile a program when we find some anamoly in the expected vs actual usage of resources, typically by poor performance. We can then generate the profile for our program and try to understand if a space leak is present, and if so, where. Heap profiles of programs that are leaky due to thunks show a steady increase in the amount of memory (the building of a large number of thunks) and then a fall as the thunks get evaluated. So the nature of the graph could be used as a heuristic to predict whether there is a space leak in the program.

Chapter 3

Examples of Space Leaks

In this chapter we shall present some examples that will aid in the understanding of space leaks in general.

3.1 Example - foldl

Foldl is the left fold of a list. Here is its definition in Haskell.

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

The second argument is a potential source of space leak.[8] Consider the following steps in the evaluation of **foldl (+) 0 [1..10]**

```
foldl (+) 0 [1..10]
= foldl (+) 0 (1:[2..10])
= foldl (+) (0 + 1) [2..10]
= foldl (+) (0 + 1) (2:[3..10])
= foldl (+) ((0 + 1) + 2) [3..10]
...
= foldl (+) ((((((((((0 + 1) + 2) + 3) + 4) + 5) + 6) + 7) + 8)
          + 9) + 10) []
         -- Peak usage here
= ((((((((((0 + 1) + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10)
= (((((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10)
...
= (45 + 10)
= 55
```

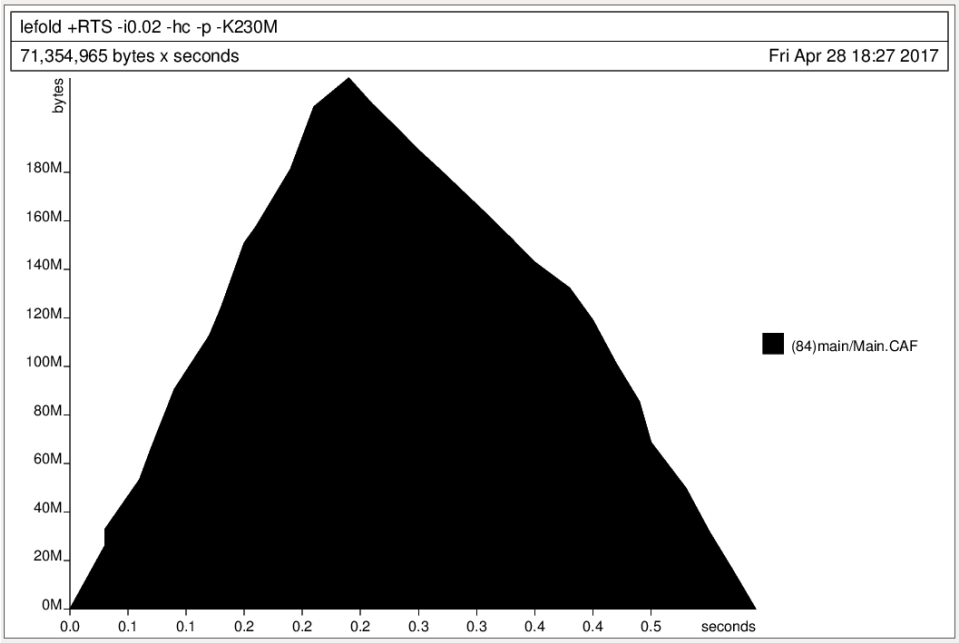

Figure 3.1: Heap profile for foldl (+) 0 [1..4000000]

The accumulating parameter is a thunk which is consuming more and more space as the computation goes on. For sufficiently large sized lists, we encounter a stack overflow. Had we evaluated all the + (keeping the foldl strict in the second argument), the space usage would have been a lot less. Infact the Data.List package contains a strict version of foldl, **foldl'**. This is how the computation would have proceeded with foldl'.

```
foldl' (+) 0 [1..10]
= foldl' (+) 0 (1:[2..10])
= foldl' (+) (0 + 1) [2..10]
= foldl' (+) 1 (2:[3..10])
= foldl' (+) (1 + 2) [3..10]
...
= foldl' (+) (45 + 10) []
= foldl' (+) 55 []
= 55
```

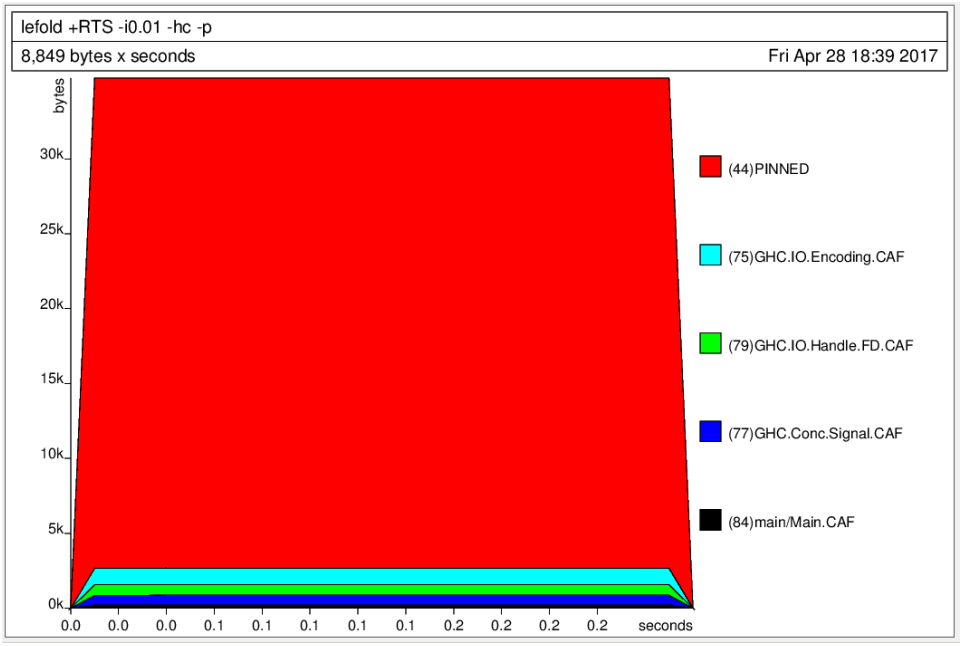

Figure 3.2: Heap profile for foldl' (+) 0 [1..4000000]

Notice the difference in scales on the y-axis in the two figures. Also notice the flatness for 3.2 as compared to 3.1.

3.2 Example - indexInto

This example is taken from ND Mitchell's blogpost[9].

Consider the following implementation of function `indexinto`:

```
indexInto :: Eq a => Int -> a -> [a] -> Maybe Int
indexInto _ _ []              = Nothing
indexInto i x (y:ys) | x == y = Just i
                     | otherwise = indexInto (i+1) x ys
```

`indexInto` when called as `indexInto 0 x ls` returns the index at which `x` occurs in `ls`.

Intuitively, this should consume constant space, but this implementation actually ends up using space linear in order to the index of the occurence of `x` into the list. This is due to the chain of accumulating `+1`'s.

```
indexInto 0 7 [1..10]
= indexInto (0 + 1) 7 [2..10]
= indexInto ((0 + 1) + 1) 7 [3..10]
...
= indexInto ((((((0 + 1) + 1) + 1) + 1) + 1) + 1) 7 [7..10]
                            -- peak usage
= ((((((0 + 1) + 1) + 1) + 1) + 1) + 1)
                            -- second argument == head of list
...
= 6
```

This behaviour is undesirable. The chain of `+1`'s has length equal to the depth of occurence of the element inside the list. This is the source of the space leak.

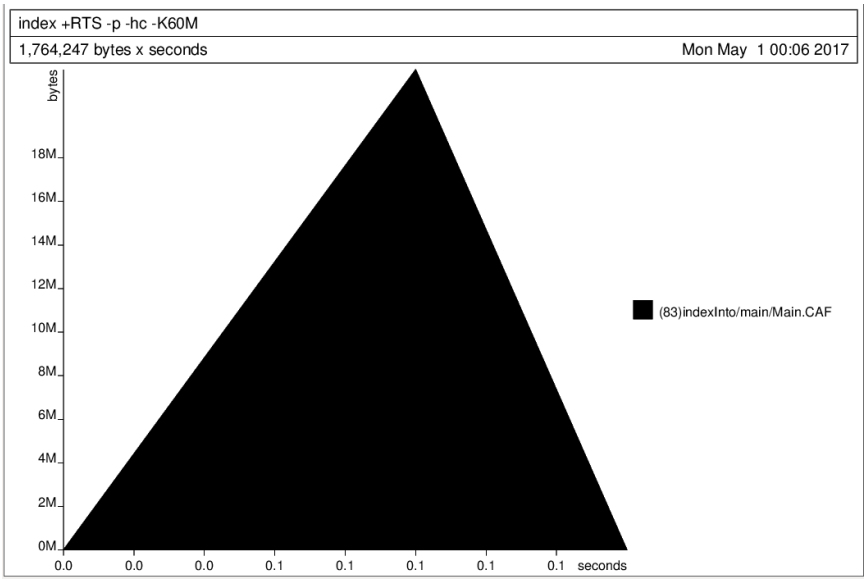

Figure 3.3: Heap profile for indexInto

We would like for these `+1`'s to be evaluated before application of another function. Hence, to fix this one should use constructs that would force evaluation of evaluate the additions. Banging the `i` in the implementation is the perfect solution:

```
indexInto :: Eq a => Int -> a -> [a] -> Maybe Int
indexInto _ _ []              = Nothing
indexInto !i x (y:ys) | x == y = Just i
                      | otherwise = indexInto (i+1) x ys
```

The trace of evaluation would now be:

```
indexInto 0 7 [1..10]
= indexInto (0 + 1) 7 [2..10]
= indexInto 2 7 [2..10]
= indexInto (2+1) 7 [3..10]
                                    -- O(1) space usage
...
= indexInto 6 7 [7..10]
= 6
```

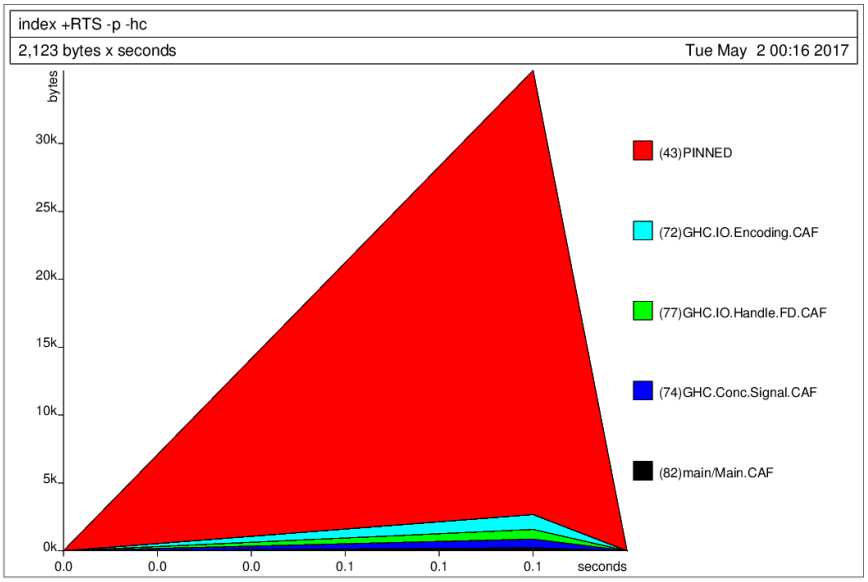

Figure 3.4: Heap profile after Fix

3.3 Example - surprise

[10] Here we present a classic example of a program in which the space leak can be fixed by modifying the garbage collector but a fix using strictness doesn't seem to exist.

```
surprise :: String -> String
surprise s = let b = break (== '\n') s
              in (fst b) + "\nsurprise\n" + (snd b)
```

`break` is an internal function of Haskell with functionality similar to:

```
break :: (a -> Bool) -> [a] -> ([a], [a])
break f [] = ([], [])
break f x:xs = | x == xs   = ([], xs)
     | otherwise  = let b = break f xs in (x:(fst b), (snd b))
```

The function `surprise` inserts a new line with "surprise" right after the first line in the string. We expect this to require constant space, but it ends up consuming space linear in order of size of the first line of the string. The example evaluation perhaps make this clear.

```
surprise "P␣not␣\nequal␣to␣NP"
= (fst b1) + "\nsurprise\n" + (snd b1)
  where b1 = break (!= '\n') "P␣not␣\nequal␣to␣NP"
= 'P' + ((fst b1) + "\nsurprise\n" + (snd b1))
  where b1 = ('P':(fst b2), snd b2)
        b2 = break (!= '\n') "␣not␣\nequal␣to␣NP"
= 'P' + ' ' + ((fst b3) + "\nsurprise\n" + (snd b1))
  where b1 = ('P':(fst b2), snd b2)
        b2 = (' ':(fst b3), snd b3)
        b3 = break (!= '\n') "not␣\nequal␣to␣NP"
...
= "P␣not␣\nsurprise\nequal␣to␣NP"
```

The trouble over here seems to be that `b1`(in `snd b1`) will not be evaluated until the string has been broken into first line and the rest.

One may try to find an alternative space leak free definition of `break`. However, it has been provedcite Hughes here that every definition of `break` has a space leak if a sequential evaluator is used. A sequential evaluator is one which continues to evaluate an expression and expressions demanded by it untill is is completely evaluated [10].

We won't discuss other solutions for this problem over here. Rather, they have been described in detail in later sections.

Note: This space leak is not reproducible in current version of GHC. The garbage collector solution proposed by Wadler[10] has been implemented in current versions of GHC and hence the mentioned space is leak is no longer reproducible.

3.4 Example - tick

[11]

```
main = print (f ([1..40000000]) (0 :: Int, 1 :: Int))

f [] c = c
f (x:xs) c = f xs (tick x c)

tick x (c0, c1) = if (even x) then (c0, c1 + 1)
              else (c0 + 1, c1)
```

Like in the example of foldl, the evaluation will proceed in two stages. Firstly a sequence of tick closures will be formed as we take elements from the list one by one. These tick closures are consuming more space than their evaluated counterparts would. To demostrate:

```
f [1,2,3,4] (0,1)
= f [2,3,4] (tick 1 (0,1))
= f [3,4] (tick 2 (tick 1 (0,1)))
= f [4] (tick 3 (tick 2 (tick 1 (0,1))))
= f [] (tick 4 (tick 3 (tick 2 (tick 1 (0,1)))))
= (tick 4 (tick 3 (tick 2 (tick 1 (0,1)))))
```

Now the ticks will be evaluated to the (+) closures. These closures are also kept as thunks. It is only after the ticks have been completely evaluated to (+) we can see a clear decrease in the amount of space required.

```
= (tick 4 (tick 3 (tick 2 ((0 + 1),1))))
= (tick 4 (tick 3 ((0 + 1),(1 + 1))))
= (tick 4 (((0 + 1) + 1),(1 + 1)))
= (((0 + 1) + 1),((1 + 1) + 1))
= ((1 + 1),(2 + 1))
= (2,3)
```

We can generate a profile for this program. We manually add cost centres to help identify where the leak is present.

```
main = do
   print (show (f [1..4000000] (0 :: Int, 1 :: Int)))

f [] c = {-# SCC allticks #-} c
f (x:xs) c = {-# SCC before2cl #-} f xs (tick x c)

tick x (c0, c1) = {-# SCC beforetick #-}
                  if (even x) then (c0, c1 + 1)
                              else (c0 + 1, c1)
```

Now we compile the program with profiling options and run it. (We initially encounter a stack overflow, so we increase the size of the stack using the -K option.) We obtain the profiling information shown in Figure 3.5.

```
   Sun Apr 30 02:00 2017 Time and Allocation Profiling Report  (Final)

      leak2 +RTS -i0.5 -hc -p -K225M -RTS

   total time  =        1.83 secs   (1830 ticks @ 1000 us, 1 processor)
   total alloc = 1,312,051,896 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

beforetick  Main     69.6   65.9
main        Main     13.2   24.4
tick        Main     10.3    0.0
before2cl   Main      3.9    9.8
f           Main      3.1    0.0


                                                     individual     inherited
COST CENTRE      MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN             MAIN                     43           0    0.0    0.0   100.0  100.0
 CAF             Main                     85           0    0.0    0.0   100.0  100.0
  main           Main                     86           1   13.2   24.4   100.0  100.0
   f             Main                     87     4000001    3.1    0.0    86.8   75.6
    allticks     Main                     89           1    0.0    0.0     0.0    0.0
    before2cl    Main                     88     4000000    3.9    9.8    83.8   75.6
     tick        Main                     90     4000000   10.3    0.0    79.8   65.9
      beforetick Main                     91     4000000   69.6   65.9    69.6   65.9
 CAF             GHC.IO.Handle.FD         81           0    0.0    0.0     0.0    0.0
 CAF             GHC.Conc.Signal          79           0    0.0    0.0     0.0    0.0
 CAF             GHC.IO.Encoding          76           0    0.0    0.0     0.0    0.0
 CAF             GHC.IO.Encoding.Iconv    64           0    0.0    0.0     0.0    0.0
```

Figure 3.5: Profiling information for tick example

The number of entries and the time columns show that the major space allocation is occuring in the **beforetick** cost centre. This corresponds to the large tick closure expression that is formed. The graph generated using the -rtsopts -h option shows the space consumption in the heap (Figure 3.6).

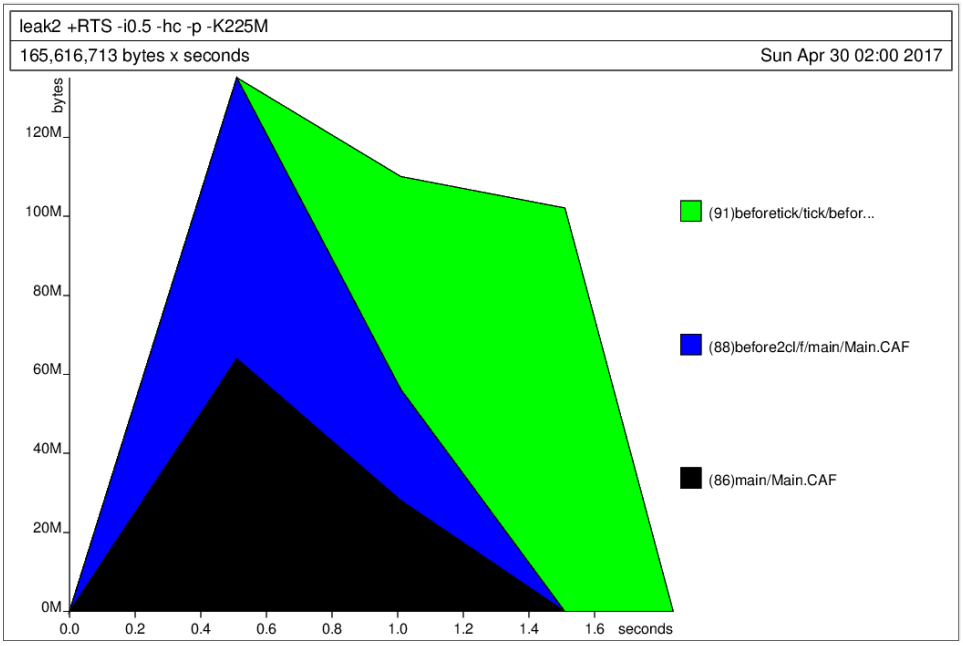

Figure 3.6: Heap profile for tick example

We can strictify c, the second argument of f to prevent the tick closure from building. But this will still leave us with the long series of (+), and while the memory consumption decreases, it is still high.(Figure 3.7).

```
f [] !c = c
f (x:xs) !c = f xs (tick x c)
   --Bang Pattern to make f strict in c
tick x (c0, c1) = if (even x) then (c0, c1 + 1)
                              else (c0 + 1, c1)
```

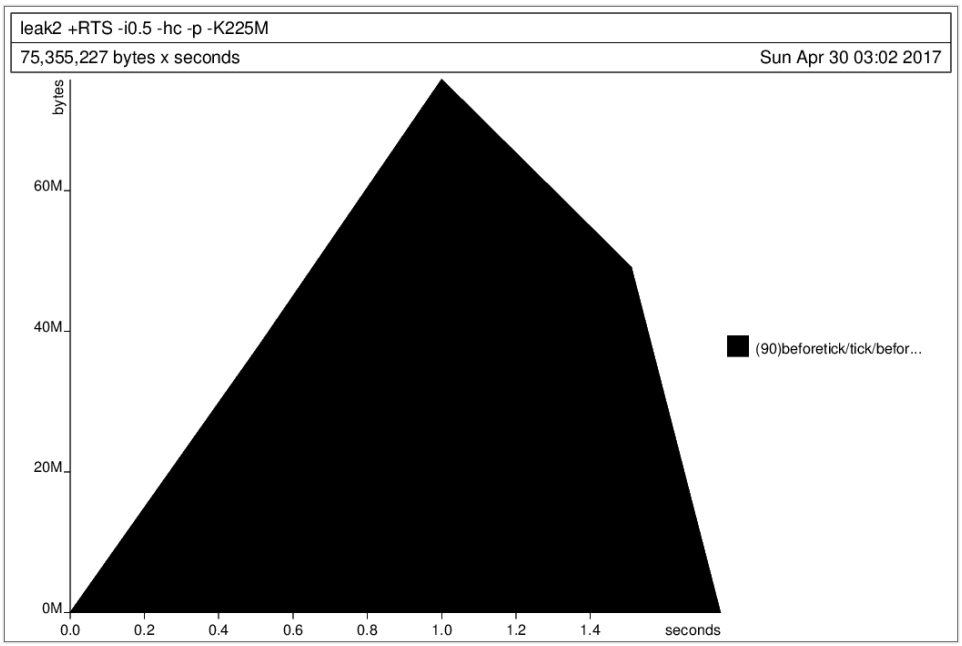

Figure 3.7: Heap profile for tick : strict f

Notice the difference in the scales between Figure 3.6 and Figure 3.7. The space consumption here is due to the (+) operations that are left unevaluated. Forcing their evaluation requires us to strictify *c0* and *c1* inside *tick*.

```
f [] !c = c
f (x:xs) !c = f xs (tick x c)
   --Bang Pattern to make f strict in c
tick x (!c0, !c1) = if (even x) then (c0, c1 + 1)
                                else (c0 + 1, c1)
        --Bang Pattern to make tick strict in c0 and c1
```

The profile corresponding to this code is shown in Figure 3.8. The memory usage is mostly constant throughout the runtime. The scale is now in kilobytes rather than megabytes, a clear sign that the space leak has been fixed. (We are getting information about the usage from cost centres which were neglected before when the space leak was present due to the difference in values.)

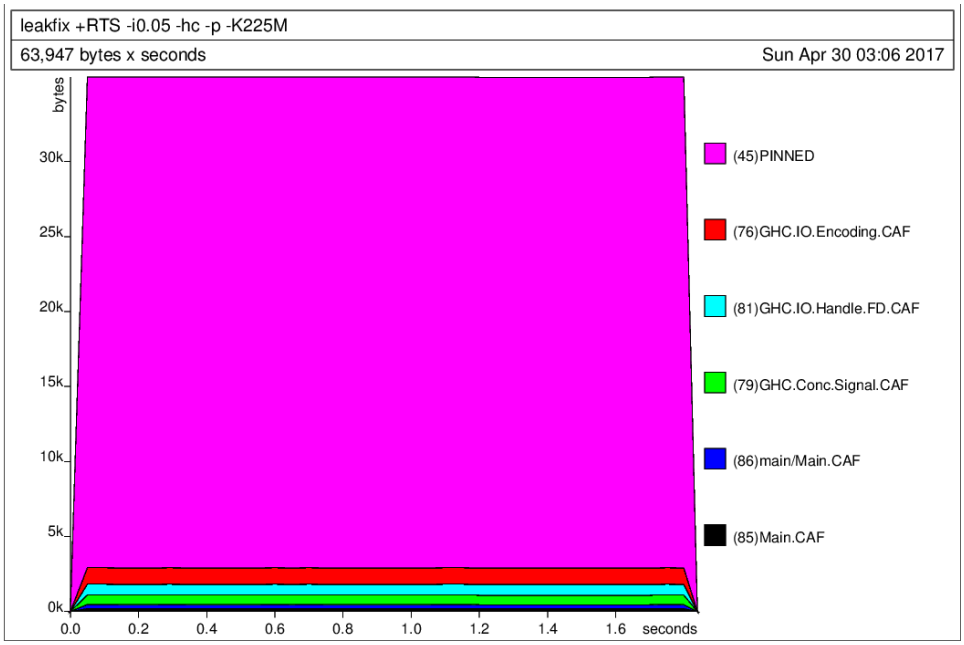

Figure 3.8: Heap profile for tick : strict f and strict tick

3.5 Example - foldr

The previous example may have suggested that strictifying arguments may be the quick fix to all space leak problems. But that is not the case. Consider a list say [1..10000]. The representation of this list in the current format takes only 3 nodes (.. as an infix function and 1 and 10000 as its two arguments). If we were to expand the list to all 10000 elements, the space consumed would be much larger. It is laziness that prevents this excessive and unnecessary space usage, by only evaluating arguments on the basis of need. Infinite data structures of the form

```
let x = 1:x  -- Generates an infinite list of ones.
```

can not work in the absence of laziness. If we are to work within the laziness model, the solution to some of these leaks would lie in identifying and strictifying only those arguments which we are certain would be evaluated. Unnecessary evaluations may force us into infinite loops while unnecessary delays in evaluation will build up closures that will waste space.

Foldr represents the right fold of the list. Here is its code in Haskell

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []    = z
foldr f z (x:xs) = f x (foldr f z xs)
```

In foldr, we open up the list element until we reach the empty list, building a chain of thunks (of f) as we go. Consider for instance, how foldr (+) 0 [1..10] will be evaluated.

```
foldr (+) 0 [1..10]
= 1 + (foldr (+) 0 [2..10])
= 1 + (2 + (foldr (+) 0 [3..10]))
= 1 + (2 + (3 + (foldr (+) 0 [4..10])))
...
= 1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + (10 +
                       foldr (+) 0 [])))))))))  --Peak usage here
= 1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + (10 + 0)))))))))
= 1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 10))))))))
...
= 1 + 54
= 55
```

As + is strict in both the arguments, none of the (+)'s can be reduced at any point until the list has been expanded out completely, no redex is generated until we reach the base case for foldr. Because of this strictifying any argument here is not going to help.

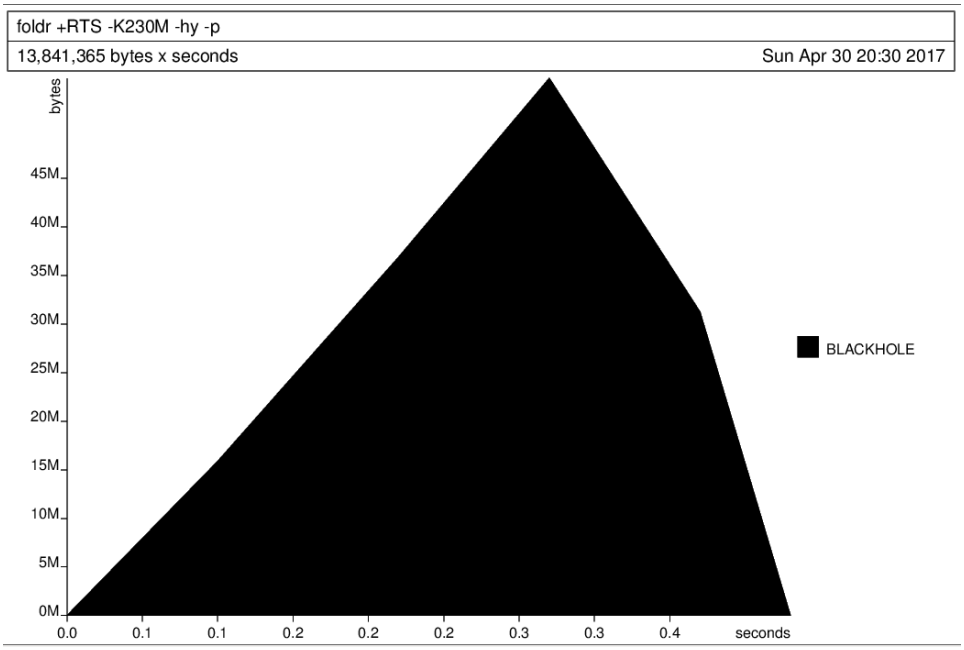

Figure 3.9: Heap profile for foldr
Blackhole represents a thunk currently being evaluated

Chapter 4

Space Leaks over the internet

Space leaks has ignited curiousity in many. A few people have pondered over this problem and written blog posts about the same. In this section we summarize two important series of blog posts written by Edward Z Yang and Neil Mitchell respectively.

4.1 Ezyang's post summary

Here we will be talking particularly about the blog post "Anatomy of a thunk leak" by ezyang[11].

```
main = evaluate (f [1..4000000] (0 :: Int))

f []    c = c
f (x:xs) c = f xs (c + 1)
```

Its evident that the $c+1$, causes accumulation of closures and hence leads to space leak. What is surprising is that compiling with optimiztions is enough to fix the leak(thanks to GHC). This is due to the fact that the compiler decides to rather use $GHC.Prim.Int\#$ - a primitive version of Int data type which doesn't form any closures.

The following is the core code generated with no optimizations.

```
f_aue [Occ=LoopBreaker] :: [Integer] -> Int -> Int
[LclId, Arity=2, Str=DmdType]
f_aue =
 \ (ds_dMW :: [Integer]) (c_atb :: Int) ->
   case ds_dMW of _ [Occ=Dead] {
     [] -> c_atb;
     : x_atc xs_atd ->
       f_aue
         xs_atd
         (+ @ Int
```

```
          GHC.Num.$fNumInt
          c_atb
          (fromInteger @ Int GHC.Num.$fNumInt (__integer 1)))
  };
```

And here is the optimized leak free code using the primitive integer - GHC.Prim.Int#

```
Main.$wf =
  \ (w_s2PM :: [Integer]) (ww_s2PQ :: GHC.Prim.Int#) ->
    case w_s2PM of _ [Occ=Dead] {
      [] -> ww_s2PQ;
      : x_auC xs_auD -> Main.$wf xs_auD (GHC.Prim.+# ww_s2PQ 1)
    }
```

Next, we consider a slightly more complicated example

```
import Control.Exception.Base

main = do
    evaluate (f [1..4000000] (0 :: Int, 1 :: Int))

f []     c = c
f (x:xs) c = f xs (tick x c)

tick x (c0, c1) | even x = (c0, c1 + 1)
                | otherwise = (c0 + 1, c1)
```

This piece of code leaks even with optimizations. Unlike the previous case, the problem is twofold over here. Here, say $GHS.Prim.Int\#$ is used with one level of optimization. Even then the leak persists. This is because the tuple constructor is still lazily evaluated. In the core code with no optimizations, we observe that `tick` is inlined.

```
f_a1ba [Occ=LoopBreaker] ::
[Integer] -> (Int, Int) -> (Int, Int)
[LclId, Arity=2, Str=DmdType]
f_a1ba =
  \ (ds_d1gJ :: [Integer]) (c_axV :: (Int, Int)) ->
    case ds_d1gJ of _ [Occ=Dead] {
      [] -> c_axV;
      : x_axW xs_axX ->
        f_a1ba xs_axX
        (case c_axV of _ [Occ=Dead] { (c0_ay0, c1_ay1) ->
         case even @ Integer GHC.Real.$fIntegralInteger x_axW
         of _ [Occ=Dead] {
           False -> (+ @ Int
```

```
          GHC.Num.$fNumInt
          c0_ay0
          (fromInteger @ Int GHC.Num.$fNumInt (__integer 1)),
          c1_ay1);
         True -> (c0_ay0, + @ Int
          GHC.Num.$fNumInt
          c1_ay1
          (fromInteger @ Int GHC.Num.$fNumInt (__integer 1)))
       }
       })
  };
```

Please note that the bang pattern(strictness) just enforces evaluation to WHNF and not any further. Bang pattern shouldn't be assumed to lead to complete evaluation.

For dicsussion on effects of strictness on the code, please refer to the `tick` program in the Example section.

4.2 Neil Mitchell's blog posts

Quoting Neil Mitchell - "Every large Haskell program almost inevitably contains space leaks". Over a series of blog posts he has reported a lot of space leaks in huge codebases, how he found them and how he fixed them. Most of them involve monads and IO actions. Here is one of the interesting examples just for the sake of it.

```
foldr (\(a,b) (c,d) -> (a+c,b+d)) (0,0) conflictList
```

The above line of code just sums up the first and second elements of `conflictList` separately as returns the cumulatives as a tuple.

Keeping in mind the semantics of `foldr` in Haskell, its easy to convince yourself that `foldr` is rather a bad choice and almost always leads to space leaks. Moreover, the leaks with `foldr` aren't fixable with strictness. So, the first step towards fixing the leak would be to replace with `foldl`.

However, in this case, even `foldl` leads to space leaks. Not only that, $foldl'$, the strict version of `foldl` forces evaluation of only the tuple constructor. So, even this implementation is not free of the accumulating addition closures. We need something that would go under the hood and force the additions inside the tuple constructors.

Solution 1:

```
foldl' (\(a,b) (c,d) ->
    let ac = a + c; bd = b + d
    in ac `seq` bd `seq` (ac,bd))
    (0,0) conflictList
```

Solution 2:

```
let strict2 f !x !y = f x y
in foldr (\(a,b) (c,d) ->
    strict2 (,) (a+b) (b+d))
    (0,0) conflictList
```

Chapter 5

Towards fixing Space leaks

A few attempts have been towards eliminating the problem of space leaks. Some are successful and have been incorporated into the GHC compiler itself. Some are hacky but have been very useful in identifying space leaks in practice. A full static analysis approach is still missing.

5.1 Fixing Space Leaks using Garbage Collection

[10]

Let us remind ourselves of the `surprise` example 3.3.

```
surprise :: String -> String
surprise s = let b = break (== '\n') s
             in (fst b) + "\nsurprise\n" + (snd b)
```

The problem over here is specifically in the expression:

```
(fst b) + "\nsurprise\n" + (snd b)
```

While evaluating, `(fst b)` is evaluated first. Even though the first element of `b` isn't required by evaluation of `(snd b)` at all, it is still held on(is live) via `(snd b)`. While we could have actually let go with the first element of the tuple, we are still holding on to it. This is the source of the leak. We also mentioned before that it has been proved that every definition of break leads to a space leak provided a sequential evaluator is being used.

Making an abstraction out of the problem we consider only expressions of the form:

```
E (fst b) (snd b)
where b = E'
```

Even after `(fst b)` is completely evaluated(and while is being evaluated), the closures reachable from first element of tuple will be treated as

live since `(snd b)` points via `b` to these closures.

As a generalization, consider this expression:

```
E (sel_n1 b) (sel_n2 b) ... (sel_nn b)
where b = E'
```

Here, it is supposed that `E'` returns an n-tuple and that `sel_ni` returns the i^{th} element of an n-tuple.

Now, it may seem natural to replace all terms of `(sel_ni (v1, v2, ..., vn))` to `vi`. If somehow, we managed to do this during the execution of the program, we may be able to plug the leak. This is where the garbage collector comes into play.

The idea formulated as a whole is to program the garbage collector to identify such expressions and replace them suitably with equavalent expressions which plug the leak. The garbage collector when kicked in during peak memory usage(or at periodic intervals) identifies all such expressions, replaces them suitably. A lot of memory which wasn't free earlier is now free and can be garbage collected.

Over here we would like to draw the attention of the reader to the fact that this reduction is not just limited to the specific example dicussed above. One could possibly identify a list of such expressions and their reductions. `head` and `tail` are some of them. One needs to be careful over here that the new expression which replaces the original expression shouldn't change the semantics of the expression.

In the context of the `surprise` example, consider following stage of evaluation:

```
= 'P' + ' ' + ((fst b3) + "\nsurprise\n" + (snd b1))
  where b1 = ('P':(fst b2), snd b2)
        b2 = (' ':(fst b3), snd b3)
        b3 = break (!= '\n') "not␣\nequal␣to␣NP"
```

If the garbage collector is kicked in at this point of time, all expressions of form `(fst (a , b))` and `(snd (a , b))` are reduced. Thus, `(snd b1)` is replaced by `(snd b2)` which is inturn is replaced by `(snd b3)`:

```
= 'P' + ' ' + ((fst b3) + "\nsurprise\n" + (snd b3))
  where b3 = break (!= '\n') "not␣\nequal␣to␣NP"
```

By this means, `surprise` runs in constant amount of space. Please note a subtle point over here. `(fst b)` or `(snd b)` are replaced only if `b` has already been evaluated to a pair i.e. it is present in weak head normal form. The evaluator itself never triggers the evaluation of any expression since this would change the semantics of the expression. Similarly, in case of `head ls`

or `tail ls`, the reduction is applied by colector only if `ls` has been evaluated aleady into `x:xs`. Also observe, that with this method, the evaluation is no longer sequential. This method is in no manner a contradiction to Hughes' [1] proof.

We mentioned before that the `surprise` space leak is no longer reproducible. This suggests that the current garbage collector employed in GHC already applies such reductions. We looked the issue up, and this indeed turned out to be the case.

Hughes, along with his proof gives a method for fixing such leaks, which we would like to mention here. The method suggests modifying the source code to indicate wherever parralization and synchronization could be used, in order to save the space leaks. Applied to our example, this would in essence mean writing `break` in fashion similar to this:

```
break ls = let (xs, ys) = (SYNCHLIST ls)
           in (PAR before '\n' ls, PAR after '\n' ls)
```

The `SYNCHLIST` operator bounds variables `xs` and `ys` to be bound to the list `xs`. The two variables are synchronized such that the next element of `xs` is not evaluated until it is demanded by `before '\n' ls` and `after '\n' ls`. The `PAR` makes the two expressions to be evaluated in parallel. Without `PAR`, the `SYNCHLIST` could lead to deadlocks.

Other than the fact that this expresssion is lot more uglier than before, careless use of synchronization and parrallelism could lead to deadly bugs. At the same time, with synchronization into the picture now, the complexity of the method is no longer necessarily the same.

As a conclusion, Wadler's method is lot better than the on suggested by Hughes.

5.2 The Stack Space Hack

[12]

In this section, we present an easy and simple hack for detecting the location of space leaks in a codebase. The main idea here is that leaky programs consume a lot of stack space. It does seem to be the case in our previous examples. The default stack size of was insufficient to run the programs, we increased the stack size using -K flag at runtime. So with a restricted stack space the program would overflow and on examining the stack trace, we would find the function that causes the overflow.

The steps to be followed are:

1. Compile the program for profiling

   ```
   ghc -rtsopts -prof -auto-all xyz.hs
   ```

2. Run the executable with limiting the stack space

   ```
   ./xyz +RTS -K100M
   ```

3. Find the stack size for which the program just runs successfully - say $67M$.

4. Reduce the stack space by a small amount and run with `-xc` flag. This will print out the stack traces for each of the exception.

5. Identify the one saying `stack overflow`. Use this trace to figure out the location of the space leak.

6. Attempt to fix the leak and rerun with same procedure. If the leak is indeed fixed, then the program should work even with very small stack size.

We demonstrate this using the following program.

```
import Data.List

main = do print (f [1..1000000])

f [] = 0
f x = 1 + g x

g [] = 0
g x = h (tail x)

h [] = 1
h x = foldl (+) 0 x
```

This program runs well with $56M$ but runs out of stack space with $55M$. Given below is the corresponding stack overflow trace:

```
*** Exception (reporting due to +RTS -xc): (THUNK_STATIC), stack trace:
  Main.h,
  called from Main.g,
  called from Main.f,
  called from Main.main,
  called from Main.CAF
Stack space overflow: current size 33568 bytes.
```

From the stack trace, we suspect that `h` has a space leak. Indeed `foldl` is the culprit over here. Using `foldl'` fixes the leak. The program now runs even with $1K$ stack space.

We call this a hack since, it is not a complete solution and does have a lot of drawbacks.

1. Stack trace produced by *-xc* ignores adjacent elements that are duplicates. However, they do contain information relevant to the leak which is lost.

2. There are no entries for functions from imported libraries, which makes the detection of leaks difficult.

3. The *-xc* flag prints stack information on all expressions. Furthermore different layers of the program can catch and rethrow exceptions. GHC does not provide a means to filter out different types of exceptions in the stack trace.

4. Not every function requiring linear stack has a space leak. So the presence of the stack overflow may represent a space leak or it may be an evaluation that just happens to require a lot of space.

5. The final function in the stack trace need not always be the culprit. It is possible that some function lower in the trace used up a lot of space before the final function.

Chapter 6

Conclusion

In this report we presented a basic survey of space leaks. An ideal tool for de In conclusion we would like to present a few ideas for future work.

6.1 Theoretical work

Some work has been done in *formalizing* a space leak [13]. One can explore on defining this problem at a more theoretical level. If the theoretical foundations of this is well laid out, then one can try to answer interesting questions. For example, since space leaks is strongly related to strictness and liveness, one can try to formally prove (or disprove) the claim that given a perfect strictness and a perfect liveness analyser, all kinds of space leaks can be eliminated.

6.2 Tool for detecting and fixing space leaks

Our primary aim during the start of the project was to build a tool that would help a programmer detect space leaks in code. We could also mature such a tool to point towards the right fixes for such space leaks. This tool could be complete static analysis tool or a part static part dynamic analysis tool. For a static analysis tool, annotating/instrumenting the code with necessary local information and then manually or automatically inferring global information needs to be done [14].

We can start off by considering a restricted set of Haskell, and using such annotations and instrumentations to identify program points that lead to the creation and destruction of thunks. This would help us, in some sense to obtain an idea of whether the number of thunks explode remain constant when a (self or mutually) recursive function executes. More thought needs to be put in this direction for developing a tool.

6.3 Stack space hack++

As we have seen the stack space hack proposed by Neil Mitchell, a few improvements can be done to improve that procedure. It could be automated at various points. One could automatically generate the stack overflow trace by finding out the maximum stack space where it flows. Once such a trace is generated, one can analyse the trace and point to exact location of the space leak automatically. One could also point towards the fix for the space leak. This kind of approach will lie in part dynamic part static analysis approach and this tool could be of great use [12].

6.4 Tool for memory analysis

Haskell could really help with a tool that lets you explore the memory (stack and heap) at any intermediate point of the program. An analogous to this would be the heap profiler present in Chrome for Javascript. The Chrome heap profiler allows a snapshot of the memory to be taken and explored. The profiler displays a tree of the memory, showing which values point at each other [4].

Pinpointing space leaks is a skill that takes practice and perseverance. Better tools could significantly simplify the process.

Bibliography

[1] John Hughes. "The design and implementation of programming languages". In: 1983.

[2] William Stoye. "The implementation of functional languages using custom hardware". In: 1985.

[3] Simon Peyton-Jones and David Lester. "Implementing Functional Languages". In: 1991.

[4] Neil Mitchel. "Leaking Space". In: 2013.

[5] *Performance/Strictness*. `http://wiki.haskell.org/Performance/Strictness`. Accessed: 2017-05-01.

[6] *Seq*. `http://wiki.haskell.org/Seq`. Accessed: 2017-05-01.

[7] *Profiling*. `http://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/profiling.html`. Accessed: 2017-05-01.

[8] *Folds*. `http://wiki.haskell.org/Foldr_Foldl_Foldl'`. Accessed: 2017-05-01.

[9] Neil Mitchel. *More space leaks: Alex/Happy edition*. 2016. URL: `http://neilmitchell.blogspot.in/2016/07/more-space-leaks-alexhappy-edition.html`.

[10] Philip Wadler. "Fixing some space leaks with a garbage collector". In: 1987.

[11] Edward Yang. *Anatomy of a Thunk Leak*. 2011. URL: `http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/`.

[12] Neil Mitchell. *Neil Mitchell's Haskell Blog*. URL: `http://http://neilmitchell.blogspot.in/`.

[13] Adam Bakewell. "An Operational Theory of Relative Space Efficiency". In: 2002.

[14] Devdeep Ray and Pratik Fagade. *Explorations in Haskell space leaks*.