A Review of Space Leaks in Haskell

CS490 Seminar Report, Spring 2020

Michael Reed[1]

[1]George Mason University

Abstract

Space leaks, or *space faults*, occur when a program uses more memory than necessary [19]. Space leaks are common in the Haskell programming language due to Haskell's use of lazy evaluation. Space leaks are a serious problem in Haskell programs, although heap profiling tools are satisfactory (but not exceptionally so) at locating and fixing them. This review starts by introducing Haskell, for those unfamiliar with it. The review then characterizes space leaks and attempt to classify them by type; explores remediation strategies; discusses existing techniques for locating and fixing space leaks; and finally recommends areas of future research.

Introduction to Haskell

Haskell is a *statically typed, pure, and functional* programming language with *lazy evaluation* semantics.

Haskell is *statically typed*, which means that all variables and functions belong to exactly one set of data, such as the set of integers, the set of booleans, or the set of functions from integers to booleans. Haskell's type system also enables programmers to concisely declare more complex data types, such as 2-tuples of integers, or a node in a binary search tree whose data is stored in the leaves. When compiling a Haskell program, the compiler checks that all functions are applied to the right data types, and they are returning the data types they are supposed to. This catches semantic errors, like `5 + ""` or `True == 5`, at compile time.

Haskell is *pure*, which means that functions in Haskell have no *side effects* [16].[1] A function has side effects if its execution modifies or is influenced by global program state; reading or writing a mutable global variable, for example, is a side effect. To illustrate, consider the function `swap(x, y)` which takes two pointers to integers `x` and `y`, and swaps their values. The `swap` function returns

[1]This is mostly true, but Haskell does support functions with side effects through the `IO` (read as "input output") monad.

no value and derives its usefulness by modifying variables outside its scope. Haskell's purity discourages writing functions like `swap`, which can make Haskell programs easier to reason about. Without side effects, a function's input will solely determine its output. This can also make Haskell functions easier to *compose* (as in "function composition") together, which leads to Haskell's next main feature.

Haskell is *functional* in that it emphasizes composing simple functions together to create complex programs. Functional languages are declarative: instead of describing *how* to do things, you describe *what* those things are. To illustrate this, consider the function `sum` which computes the sum of a list of integers. Here is its definition in Java:

```
int sum(int[] ns) {
  int sum = 0;
  for (int i = 0; i < ns; i++) {
    sum = sum + ns[i];
  }
  return sum;
}
```

The `sum` function must state explicitly *how* to calculate the sum of `ns`: first create a variable `sum`, initialized to `0`, to accumulate the sum thus far, then create a variable `i` which is used to access each element of `ns` and add it to the variable `sum`. After we have iterated through the entire array we then explicitly return our sum. Compare this definition to the equivalent function in Haskell:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = n + (sum ns)
```

The first line is `sum`'s type signature and means that `sum` takes one argument—a list of integers `[Int]`—and returns one result—an `Int`. The second and third lines use *pattern matching*, which allows performing different actions for different shapes of input. In particular, the second line says the sum of the empty list `[]` is `0`. The third line defines summation recursively: it says the sum of a non-empty list `n:ns`—where `n` is the first element and `ns` is the remaining elements—is `n` plus the sum of `ns`. While the Java version describes the algorithm for producing a sum, the Haskell version defines *what* a sum is. And unlike the Java definition, Haskell's definition avoids mutable variables entirely. These properties can make Haskell easier to reason about and enables, for example, the use of mathematical induction in proofs of correctness for Haskell programs [13].

Finally, Haskell uses *lazy evaluation*, otherwise known as *call-by-name* evaluation, which means that function arguments are evaluated only when needed [13]. This is in contrast to languages with strict evaluation, like Java, where all function arguments are unconditionally evaluated before function calls. Consider a function `f` that takes two arguments `x` and `y`. It returns `42` if `x` is `0` and returns the sum of `x` and `y` otherwise. Here is `f` in Java:

```
void f(int x, int y) {
  if (x == 0)
    return 42;
  else
    return x + y;
}
```

If we call `f` with some arguments:

```
f(0, 1 + 2 + 3 + 4 + 5);
```

Java will first evaluate the sum `1+2+3+4+5`:

```
f(0, 15);
```

Then Java will call `f` with the given arguments. Because the argument `x` equals `0`, `f` will return `42`, ignoring the value of `y`. In other words, computing `1+2+3+4+5` was unnecessary. Now consider `f` in Haskell:

```
f :: Int -> Int -> Int
f 0 y = 42
f x y = x + y
```

We call `f` with the same arguments as before:

```
f 0 (1 + 2 + 3 + 4 + 5)
```

Unlike in Java, when performing the function call Haskell has not yet evaluated its function arguments; it's delaying their evaluation until needed. The first argument is `0`, so this function call will match `f`'s first equation (`f 0 y = 42`) and return `42`. During this process the sum `y` is not evaluated, saving time. Had the second argument `y` been a more computationally intensive expression, such as `factorial 500`, then more time would be saved.

Besides saving time, laziness also allows for *infinite data-structures*:

```
positiveInts :: [Int]
positiveInts = [1,2 .. ]
```

`positiveInts` is a list of integers from `1` to infinity. When the Haskell compiler encounters this declaration, it does not declare an infinite list of integers, which would take infinite memory. Instead, the compiler maps the variable `positiveInts` to a sequence of integers, starting at 1, which will be evaluated *as needed*. For example, using the function `take n xs`, which returns the first n elements from the list `xs`, we can define the sum of the first 100 positive integers as follows:

```
sum (take 100 positiveInts)
```

Before evaluating this expression, only the first two elements of `positiveInts` (`1` and `2`) are in memory. Evaluating this expression forces an additional 98 elements of `positiveInts` to be computed and stored in memory. This works because, while `positiveInts` is infinite, we are only consuming a finite number of

integers from it. We cannot, for example, compute the length of `positiveInts` using the `length` function:

```
numPositives :: Int
numPositives = length positiveInts
```

If we force the evaluation of the variable `numPositives`, its evaluation would start but never finish. This is because `length` calculates the length of its argument by traversing it in full, which is impossible if its argument is infinite.

These expressions which have not yet been evaluated are called **suspended computations**, or **thunks** [18, 20]. The Haskell runtime allocates thunks on the heap and keeps track of each one until it is needed, in which case the underlying computation is performed and the thunk replaced by the result [25]. Because thunks themselves take up memory, retaining too many of them at once can use excessive space. This leads us to the topic of space leaks.

Space leaks

A *space leak* occurs when memory is allocated too early before its first use or kept around too late after its last use [19, 22]. With a memory leak, allocated memory is never freed; with a space leak, memory is eventually freed, but later than expected [19]. As noted by Hudak et al., space leaks are an inevitable side effect of lazy evaluation [12]:

> By abstracting away from evaluation order, lazy evaluation also abstracts away from object lifetimes, and that is why lazy evaluation contributes to space leaks. Programmers who cannot predict—and indeed do not think about—evaluation order also cannot predict which data structures will live for a long time.

Incidentally, our `sum` function from earlier has a space leak.

Example 1: A simple sum

Recall the definition of sum:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = n + (sum ns)
```

To see why `sum` leaks, we'll apply `sum` to a list of integers and print the result. The use of `print` is necessary to force the evaluation of `sum`, which would otherwise remain unevaluated. Here is the application:

```
print (sum [1,2,3,4,5])
print (1 + sum [2,3,4,5])
print (1 + (2 + sum [3,4,5]))
print (1 + (2 + (3 + sum [4,5])))
```

```
print (1 + (2 + (3 + (4 + sum [5]))))
print (1 + (2 + (3 + (4 + (5 + sum [])))))
print (1 + (2 + (3 + (4 + (5 + 0)))))
print (1 + (2 + (3 + (4 + 5))))
print (1 + (2 + (3 + 9)))
print (1 + (2 + 12))
print (1 + 14)
print 15
(print "15" to the screen)
```

In principle, computing the sum of a list of integers can be done with constant additional space—we need only store the sum of the list so far and the next element in the list. But this version of `sum` uses space linear with the size of the input list, as shown by the accumulation of +'s (each corresponding to a recursive call to `sum`) during evaluation.

This happens because, while we are consuming list elements from left to right, `sum`'s definition of `sum (n:ns) = n + (sum ns)` forces addition to associate to the right [19]. This ensures that no additions are performed until `sum` has consumed the entire list—observe how when `sum []` is evaluated to `0`, the expression `5 + 0` can finally be evaluated. While this leak may seem negligible, it makes `sum` impractical on larger lists. For example:

```
sum [1..5000000]
```

This problem can be avoided by rewriting `sum` so that its addition is left-associative, enabling us to interleave list consumption with addition [19]:

```
sum2 :: [Int] -> Int
sum2 xs = sum2helper 0 xs

sum2helper :: Int -> [Int] -> Int
sum2helper acc [] = acc
sum2helper acc (n:ns) = sum2helper (acc + n) ns
```

The function `sum2` now resembles the Java `sum` function, in that it accumulates the sum in an additional parameter `acc`. The function `sum2helper` does the brunt of the work, taking each element in the list of numbers (`n:ns`) and adding it to the accumulator `acc`. Using `sum2`, let's try again evaluating the list of numbers from `1` through `5`:

```
print (sum2 [1,2,3,4,5]))
print (sum2helper 0 [1,2,3,4,5])
print (sum2helper (0 + 1) [2,3,4,5])
print (sum2helper (0 + 1 + 2) [3,4,5])
print (sum2helper (0 + 1 + 2 + 3) [4,5])
print (sum2helper (0 + 1 + 2 + 3 + 4) [5])
print (sum2helper (0 + 1 + 2 + 3 + 4 + 5) [])
print (0 + 1 + 2 + 3 + 4 + 5)
```

```
print (1 + 2 + 3 + 4 + 5)
print (3 + 3 + 4 + 5)
print (6 + 4 + 5)
print (10 + 5)
print 15
(print "15" to the screen)
```

The space leak is still present,[2] as shown by the accumulation of `+`'s in the `acc` argument to sum. While the leak in `sum` is from an algorithmic flaw, the leak in `sum2` is from too much laziness. Haskell delays evaluating the calls to `+` in the accumulator `acc`, which builds up over successive function calls; the `+` thunks are only evaluated when they are the only things left to evaluate. To fix the leak we need to *eagerly* evaluate `sum2helper`'s parameter `acc`, which should prevent the accumulation of `+` thunks. This is done in Haskell with *bang patterns*:

```
sum3 :: [Int] -> Int
sum3 xs = sum3helper 0 xs

sum3helper :: Int -> [Int] -> Int
sum3helper !acc [] = acc
sum3helper !acc (n:ns) = sum3helper (acc + n) ns
```

The `!` in front of `acc` forces `acc` to be evaluated when `sum3helper` is called [19].[3] Here is a trace of `sum3`'s execution:

```
print (sum3 [1,2,3,4,5])
print (sum3helper 0 [1,2,3,4,5])
print (sum3helper (0 + 1) [2,3,4,5])
print (sum3helper 1 [2,3,4,5])
print (sum3helper (1 + 2) [3,4,5])
print (sum3helper 3 [3,4,5])
print (sum3helper (3+3) [4,5])
print (sum3helper 6 [4,5])
print (sum3helper (6+4) [5])
print (sum3helper 10 [5])
print (sum3helper (10 + 5) [])
print (sum3helper 15 [])
print 15
(print "15" to the screen)
```

There is now at most one suspended addition computation at any point during, so the space leak is fixed.

[2] Using GHC `8.8.3` and the `-O2` flag, this example does not actually leak. See the later section on *Optimization and Space Leaks*.

[3] Placing `!` in front of a parameter evaluates it to *weak head normal form*, not *normal form*. More on that in the next example.

Example 2: Leaky tuples and normal form

An expression is in *normal form* if it cannot be evaluated further. Examples include `1`, `False`, and `[1, 2, 3]`. An expression is in *weak head normal form* (*WHNF*) if its outermost term[4] does not require further evaluation [19]. Examples include `(1 + 1) : []`, whose outermost term is the list constructor (`:`); and `(0 + 1, 0 + 1)`, whose outermost term is the tuple constructor (`,`). Every expression in WHNF is also in normal form.

The following example illustrates a leak which cannot be fixed by evaluating the leaky expression to WHNF. Instead, we must evaluate these expressions to normal form. Consider the function `f` which takes a list of integers `xs` and returns a 2-tuple `(x, y)`, where `x` is the number of odd numbers in `xs` and `y` is the number of even numbers [5]:

```
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 )
```

As you might be able to tell, this code leaks:

```
print ( f [1,2,3,4] (0, 0) )
print ( f [2,3,4] (tick 1 (0, 0)) )
print ( f [2,3,4] (0 + 1, 0) )
print ( f [3,4] (tick 2 (0 + 1, 0)) )
print ( f [3,4] (0 + 1, 0 + 1) )
print ( f [4] (tick 3 (0 + 1, 0 + 1)) )
print ( f [4] (0 + 1 + 1, 0 + 1) )
print ( f [] (tick 4 (0 + 1 + 1, 0 + 1 + 1)) )
print ( f [] (0 + 1 + 1, 0 + 1 + 1) )
print (0 + 1 + 1, 0 + 1 + 1)
print (1 + 1, 0 + 1 + 1)
print (2, 0 + 1 + 1)
print (2, 1 + 1)
print (2, 2)
(print "(2, 2)" to the screen)
```

The problem is that the additions via `tick` are accumulating in the tuple instead of being evaluated. So let's make `f` strict in its argument `c` to get rid of the leak. Here's the new definition of `f`:

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

This doesn't work because the bang pattern only evaluates `c` to weak head normal form, which leaky expressions like `(0 + 1, 0 + 1)` are already in. Instead,

[4]The outermost term in an expression is the highest term in the corresponding parse tree.

we must forcibly evaluate the expressions *inside* the tuple, which we can do in `tick` using pattern matching:

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

Let's see if that fixes the leak by running the example again:

```
print ( f [1,2,3,4] (0, 0) )
print ( f [2,3,4] (tick 1 (0, 0)) )
print ( f [2,3,4] (0 + 1, 0) )
print ( f [3,4] (tick 2 (0 + 1, 0)) )
print ( f [3,4] (tick 2 (1, 0)) )
print ( f [3,4] (1, 0 + 1) )
print ( f [4] (tick 3 (1, 0 + 1)) )
print ( f [4] (tick 3 (1, 1)) )
print ( f [4] (1 + 1, 1) )
print ( f [] (tick 4 (1 + 1, 1)) )
print ( f [] (tick 4 (2, 1)) )
print ( f [] (2, 1 + 1) )
print ( f [] (2, 1 + 1) )
print (2, 1 + 1)
print (2, 2)
(print "(2, 2)" to the screen)
```

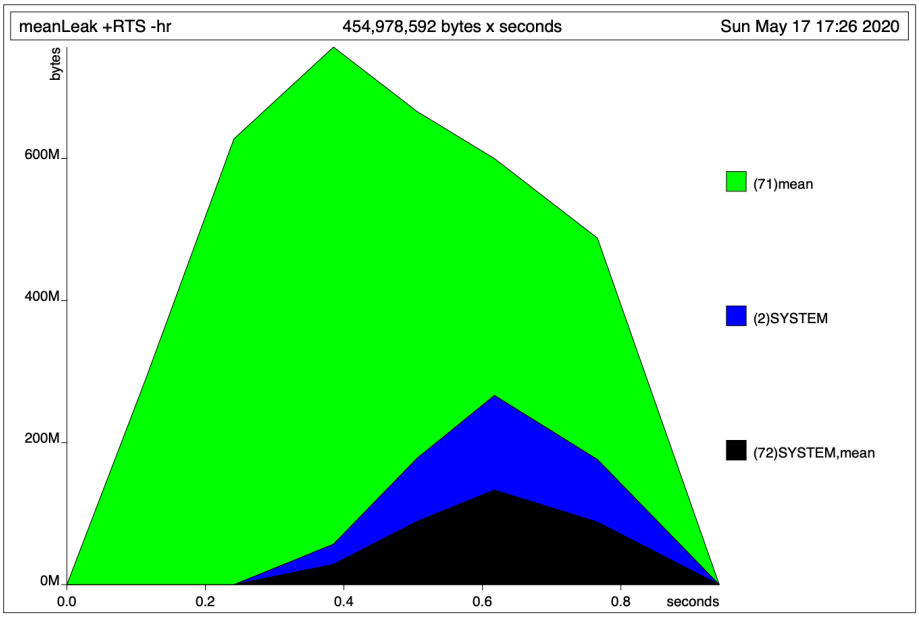
At no point during execution is there more than one suspended computation for addition, so the leak is fixed.

Example 3: Sharing causes leaks

The following example suffers from a space leak not due to laziness, but due to *sharing* [19]. Assuming `sum` and `length` do not leak, consider the following function:

```
mean :: [Int] -> Int
mean xs = sum xs `div` length xs
```

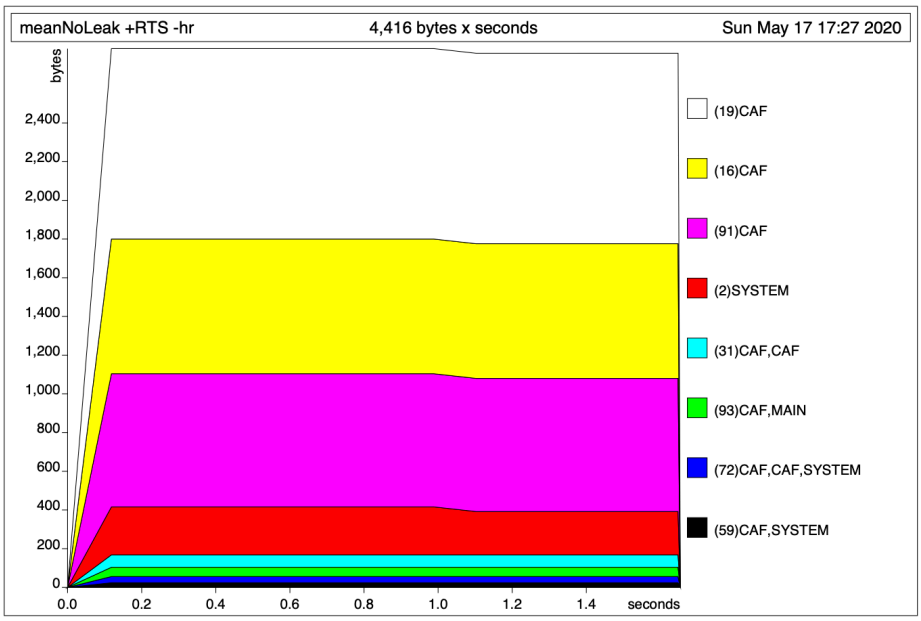
The `mean` function finds the arithmetic mean of a list of integers `xs` by dividing the sum of `xs` by its length, using the `div` function for integral division [19]. `xs` is *shared* between both the expressions `sum xs` and `length xs`, meaning that `xs` is only computed once and its result used twice, saving time. However, this sharing causes a space leak: the garbage collector cannot free `xs` as `sum` consumes it, because `length` also requires all of `xs`. Only after `sum` has finished with `xs` can `length` start consuming `xs`, while the collector frees elements of `xs` as they are consumed. This causes `mean` to use space linear with the length of `xs`, as shown in the retainer profile below:



One solution is to remove the sharing of `xs` by duplicating it in both expressions, allowing `xs` to be collected as it is consumed. This would fix the leak but at the cost of generating `xs` twice. Mitchel notes that the proper solution is to consume the length while consuming the sum [19]:

```
mean :: [Int] -> Int
mean xs = mean' 0 0 xs
  where
    mean' !s !l (x : xs) = mean' (s + x) (l + 1) xs
    mean' !s !l []       = div s l
```

This allows each element of the list `xs` to be created, used in `mean'`, and immediately garbage-collected without duplicating any work. As shown in the retainer profile below, `mean` now uses constant additional space, so the leak is fixed. More generally, this rewriting technique is called *tupling* [6].



Example 4: Fixing leaks through the compiler

While the leak in `mean` above can be fixed through clever refactoring, some functions cannot be written in Haskell without leaks—that is, without modifying the compiler. Consider the function `break`, which takes a list of characters and returns a tuple where the first element is all characters leading up to the newline and the second element is all elements following the newline [10]:

```
break (x:xs) = if x == '\n' then ([], xs)
               else (x : (fst b), (snd b))
               where b = break xs
```

The line `where b = break xs` declares the variable `b` visible in the scope of `break` and binds it to the expression `break xs`. Now consider the function `surprise`, defined in terms of `break`, which takes a string `s` and inserts the string `"surprise"` after the first newline in `s` [10]:

```
surprise xs = let b = break xs
              in (fst b) ++ "\nsurprise\n" ++ (snd b)
```

`surprise` uses space linear with the size of `xs`, despite the fact that in principle space usage should be constant [19]. Hughes showed that with a sequential evaluator—one which evaluates expressions iteratively—`surprise` cannot be written without leaks. To fix this class of leaks, Wadler modified the garbage collector to collect in *parallel* with evaluation [29]. See Wadler for details.

Further characterization of space leaks

There are multiple conflicting characterizations of space leaks [19, 22, 26]. Moalla's characterization is perhaps the most complete, and he finds that space leaks have one of three root causes [20]:

- **Degree of evaluation**: Lazy evaluation can save space and time by delaying, potentially forever, memory- or time-intensive computations. However, lazy evaluation can allocate a large amount of space towards thunks. Adding strictness often reduces memory usage by minimizing thunk buildup, but sometimes strictness can actually increase space usage—for example, when forcing evaluation of a large data structure of which only part was needed. Despite these downsides, the consensus is that lazy evaluation's benefits still warrant its inclusion in Haskell [12, 14, 17].
- **Degree of sharing**: In Haskell, variables occurring more than once in the scope of an expression are *shared*. In the expression `head xs ++ last xs`, for example, the list variable `xs` occurs twice, so to avoid computing it twice Haskell uses the same reference to `xs` in both occurrences. Sharing, when combined with tail recursion and garbage collection, allows such an expression to run in constant space [9]. However, sharing can cause data to be retained for longer than it normally would, as the garbage collector must wait until all references to that data have been dropped before collecting it. Moalla notes that explicit unsharing causes some data to be computed twice, but may allow for constant space usage [20].
- **Algorithmic design**: a space efficient algorithm as one which either minimizes memory usage through judicious use of sparse data structures or avoids additional memory altogether with constant-space algorithms [20]. An example of a space-inefficient algorithm is any function which uses the Haskell function `foldr`, which generalizes the pattern of recursion seen in `sum` previously of applying a binary operator over a list with right-associativity [13]. Due to the way lists are constructed in Haskell, `foldr` ensures that no reductions can take place until the entire list is traversed, making space usage proportional to the length of the input list.[5]

Additionally, Haskell is vulnerable to the following specific forms of space leaks:

- **Memory Leaks**: they are possible in Haskell when calling out to C code from a Haskell program wherein memory is allocated but not freed. Due to Haskell's being garbage collected and the rarity of needing to making foreign function calls to C, this is a rare problem [26].
- **Strong reference leak**: occur when a reference is kept to data that will not actually be used. This rarely happens due to laziness, which ensures that for the most part data is only evaluated when it is used. Mutable

[5] Unless the function provided to `foldr` yields another cons cells, e.g., `f = foldr (:) []`.

references like `IORef` can introduce strong reference leaks, as the garbage collector might be unable to infer whether some data is still needed in future computations [26].

Some problem domains, such as Functional Reactive programming and stream processing, have libraries which systematically avoid space leaks through their design [17]. All major Haskell stream processing libraries for Haskell, for example, ensure that memory is not retained longer than needed [17].

Finding and fixing space leaks

Tools and methods for finding and fixing space leaks, while sufficient, have much room for improvement because, as Moalla notes, heap profiling tools are effective enough at finding space leaks [20]. While effective, the process can be involved, and in particular, heap profiling tools can burden the programmer with too much irrelevant information [3, 5, 7, 20, 21]. Mitchell notes that more advanced tools, which can detect code with possible space leaks, have not had mainstream adoption [19]. Creating minimal test cases can help mitigate this problem of too much code to look at, although creating such test cases is something of a dark art, especially in complex codebases [21]. Besides heap profiling, there are few other approaches for locating space leaks.[6] The most prominent is the Glasgow Haskell Compiler (GHC), which performs sophisticated *absence* and *demand analysis* for transforming lazy code into strict code, which as a side effect avoids some space leaks [7, 25].

Once a space leak is fixed it can be difficult to write a regression test for, despite such tests being essential to prevent reintroducing the bug [30]. A recent approach by Marlow deals with this problem with *weak pointers* [7].[7]

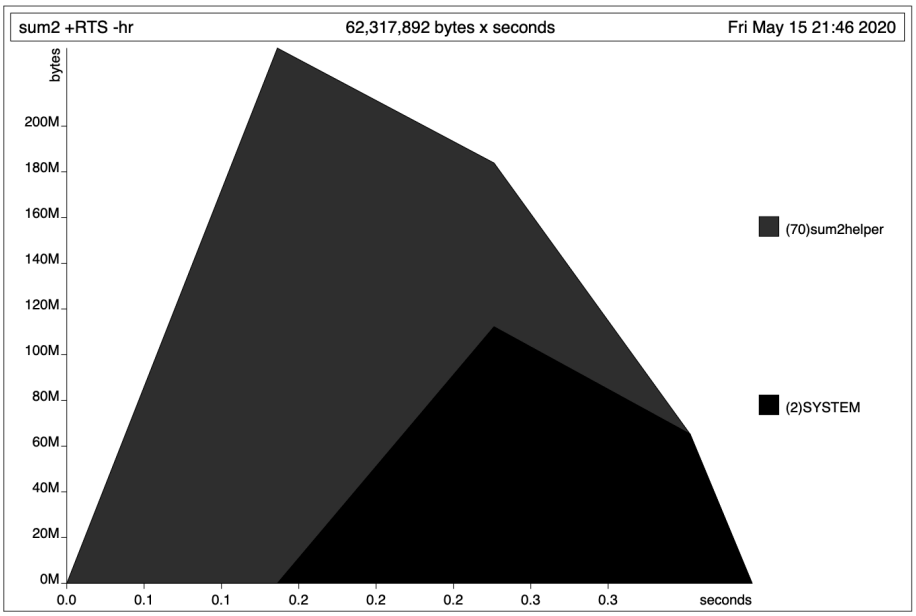
Memory profiling for finding leaks

Memory leaks can be easily discovered, as they only occur when a program exits with more heap memory than it started with; users need only compare the two statistics [19]. A space leak, by comparison, will eventually free all memory it allocates, just later than expected [19]. To catch space leaks one must collect memory statistics at runtime and inspect them later in search of memory spikes during execution. Haskell's tooling accounts for this by enabling a Haskell program to dump runtime memory statistics to a file, which can be inspected after the program exits or in real time [1, 2, 22]. These files can be converted to graphs using the `hp2ps` tool for visualizing memory usage [1].

[6] The tool `seqaid` describes itself as able to diagnose and fix space leaks. Unfortunately, it is incomplete as of time of writing [23, 24]

[7] For more information on Haskell's implementation of weak pointers, see the official documentation at [27]

For example, the graph below is the heap profile of our leaky `sum2` function from before, as it appears in the expression `print (sum2 [1..5000000])`. The graph shows that in the first 0.2 seconds of execution, the function `sum2helper`, depicted in gray, allocated roughly 240 MB of memory—thunks for `+` computations—after which memory usage drops gradually as thunks are evaluated and replaced by their result.



Profiling options in the Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) supports creating a variety of heap profiles, the most popular of which are as follows:

- A **retainer profile** classifies heap data by who has access to it [1, 22]. This can help determine why certain data is being retained, as a retainer profile will provide a list of functions (called *retainers*) maintaining a reference to some data. Retainer profiles are particularly effective at discovering space leaks induced by too much sharing, because they naturally highlight which components have access to (i.e., are sharing) a piece of data [21].
- A **producer profile** classifies data by who produces it [22]. This can help discover space leaks isolated to a single function, such as the previous leaky `sum2` example. If the function producing the data is not the source of the leak, however, then the leak may stem from another component retaining access to the data. Thus, sometimes producer profiles must be used in conjunction with retainer profiles [1].

- A **biographical profile** classifies data by what stage in its lifetime it is in, where data can be in four stages during execution [1, 22]:
  - *Lag* if it has been allocated but not used.
  - *Use* if it has been used for the first but not the last time.
  - *Drag* if it has had its last use but is still in memory (it has not yet been garbage-collected).
  - *Void* if it will never be used.

  Biographical profiles help describe a program's overall space-efficiency. A program with perfect space-efficiency, for example, has no drag, void, or lag in any of its data at any point during execution [22]. While this is usually unachievable, biographical profiling can still be used to great effect. For example, in a case study on `nhc`, the space-efficient Haskell compiler, Röjemo and Runciman were able to halve its memory usage through the aid of biographical profiling [22].

Heap profiling tools, despite their benefits, can be difficult to implement especially in a lazy functional language [20]. Modern optimizing compilers such as GHC perform significant transformations on source code, which can make it hard to relate generated code back to the original source code for implementers [20]. Additional aggressive optimizations like array fusion [4], absence analysis, and demand analysis [25] further complicate this [20].

Effects of program optimization on space leaks

In addition to profiling tools, GHC's extensive optimizations tend to preemptively fix space leaks [7]. In particular, GHC employs *strictness and absence analysis*, which transforms some call-by-name parameters into call-by-value, so they are evaluated eagerly during function calls [25]. These call-by-value arguments will thus avoid having thunks created for them, which systemically avoids them leaking space [14]. In Marlow's case, for instance, out of 17 space leaks he fixed in a program, only 7 of them appeared when optimizations were enabled [7]. Besides plugging leaks, these optimizations improve performance in general. In a benchmark of roughly 60 programs, Sergey et al. found that GHC's strictness and absence analysis caused a 16.9% mean reduction in allocations and a 0.4% *increase* in runtime [25].

While strictness analysis can inadvertently eliminate space leaks and thus change the space complexity of function(s), other techniques tend to affect space performance by a constant factor. For example, *thunk recycling* saves space by making thunk generation use the memory from existing thunks instead of allocating new memory [28], and *think lifting* increases memory efficiency of function calls whose arguments are nested function calls [10]. One final technique, *deforestation*, can lead to more dramatic space savings and is discussed in the next section.

Deforestation: removing intermediate lists

Intermediate lists are extremely common in Haskell. They enable a compositional programming style where lists are passed from one function to the next, with each function consuming the previous intermediate list and creating a new one [8]. One such example is the function `all`, which takes a predicate function `p` and a list `xs`, and returns whether `p x` is true for all `x` in `xs`:

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

`all` does this by first mapping the function `p` over the list `xs`, like so:

```
and (map p [x1, x2, ..., xn]) => and [p x1, p x2, ..., p xn]
```

After this, the resulting list is evaluated by `and`, which inspects each element of the list from left to right until it finds a `False` value. If this happens, `and` returns `False`. If `and` consumes the entire list without encountering `False`, then it returns `True`.

The use of intermediate lists here is inefficient in time and space: an imperative language could avoid the use of intermediate lists by iteratively mutating the input list, which requires constant additional space. Additionally, the construction of `all` prevents the input list `xs` from being collected until `map p xs` completes.

To address this, Gill et al. propose *deforestation*, a program transformation which removes most instances of intermediate lists. Deforestation enables `all` to use constant space by allowing `xs` in `all p xs` to be collected while it is being consumed [8]. In a Haskell program which solves the 10-queens problem, Gill et al. found deforestation to decrease runtime by about 3 times and heap memory usage by about 5 times [8]. However, it is important to note that deforestation only indirectly minimizes space leaks.

Weak Pointers for precisely locating leaks

Some authors, such as Marlow, contend that GHC's profiling facilities can be insufficient for finding space leaks in large codebases [7]. Consider the possibility of some data being kept in memory due to some function referencing it, and that function is referenced by yet another function, and so on. Heap profiles generally expose only one level in this *chain of references*, when what is needed is the entire chain [7]. Further, Marlow finds that space leaks are difficult to write regression tests for if the space leak changes a function's space usage by a constant factor [7]. Marlow found that *weak pointers* address both of these concerns.

A weak pointer is a reference to some data which is not considered by the garbage collector when determining if that data is reachable [27].[8] Therefore,

[8] The garbage collector only collects *unreachable* data.

weak pointers can be used to determine whether some data has been collected, and can be used to test for space leaks as follows [7, 27]:

1. Create a weak pointer to `v` by calling `mkWeakPtr v`.
2. Force the garbage collector to run by calling `System.Mem.performGC`.
3. Attempt to dereference the weak pointer by calling `deRefWeak`. If dereferencing fails, then `v` has been collected. Otherwise, `v` is still live, presumably because some expression holds a reference to `v`.

This approach can be used in automated regression testing to ensure that space faults do not reappear after being fixed [7].

Future work

Existing approaches to detect space leaks are either slow, ineffective, or both. Manual code inspection is intractably slow; supplementing manual inspection with additional information, such as heap profiles, is faster but still slow overall [3, 5, 7, 20, 21]. Automated approaches would be faster in theory, but no fully automated methods exist [20]. One possible method would be to statically analyze the suspect code to find its expected heap memory usage, profiling the running program to determine actual memory usage, and comparing expected memory usage to actual [11, 14]. If actual exceeds expected, then there may be a space leak.

Another way for finding leaks would be to place suspect code (perhaps even the entire program) in a space-bounded containers, and observing if the code within executes successfully under the container's bounds. Yang and Mazières implemented this idea in Haskell under the name *dynamic space limits* [31]. Their work enables programmers to create space-bounded containers, whose bounds can be varied at runtime [31]. If the code within exceeds the given bounds, then it is given an exception. Dynamic space limits could be used for automated regression testing of space leaks, and would likely be more robust than the aforementioned approach with weak pointers. Unfortunately, dynamic space limits are not yet integrated into GHC proper [31], and thus remains a non-option for most Haskell developers.

Finally, Latoza et al.'s work on *explicit programming strategies* could be used to document a more systematic approach to discovering and fixing space leaks, which might lower the bar of entry for those new to Haskell [15]. Diagnosing space leaks is a difficult process, and requires knowing the layout of Haskell's heap, evaluation semantics, and the difference between the myriad memory profiles available in GHC [3, 5, 7, 20, 21]. Packaging the programming task of "space leak identification and remediation" into an algorithmic, step-by-step guide could help guide Haskell programmers when locating and fixing space leaks.

Acknowledgments

Thank you to Prof. Mark Snyder for his stimulating discussions on the innards of Haskell, and his thoughtful advice on writing clear, persuasive prose. Thank you to Prof. Pearl Y. Wang for their advice on picking a project and defining a feasible scope. And thank you to all the professors who took time out of their schedules to help me find a research topic, including Prof. Thomas LaToza, Prof. Hakan Aydin, and Prof. Jonathan Bell.

References

[1] 7.4. Profiling memory usage: *https://downloads.haskell.org/ghc/latest/docs/html/users_guide/profiling.html#profiling-memory-usage*. Accessed: 2020-05-13.

[2] 7.5.4. Viewing a heap profile in real time: *https://downloads.haskell.org/ghc/latest/docs/html/users_guide/profiling.html#viewing-a-heap-profile-in-real-time*. Accessed: 2020-05-16.

[3] Anatomy of a thunk leak: 2011. *http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/*. Accessed: 2020-03-24.

[4] Chakravarty, M.M.T. and Keller, G. 2001. Functional Array Fusion. *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy, 2001), 205–216.

[5] Chasing a Space Leak in Shake: 2013. *https://neilmitchell.blogspot.com/2013/02/chasing-space-leak-in-shake.html*. Accessed: 2020-02-28.

[6] Chin, W.-N. 1993. Towards an automated tupling strategy. *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '93* (Copenhagen, Denmark, 1993), 119–132.

[7] Fixing 17 space leaks in GHCi, and keeping them fixed: 2018. *https://simonmar.github.io/posts/2018-06-20-Finding-fixing-space-leaks.html*. Accessed: 2020-02-28.

[8] Gill, A. et al. 1993. A short cut to deforestation. *Proceedings of the conference on Functional programming languages and computer architecture - FPCA '93* (Copenhagen, Denmark, 1993), 223–232.

[9] Gustavsson, J. and Sands, D. 2001. Possibilities and limitations of call-by-need space improvement. *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming - ICFP '01* (Florence, Italy, 2001), 265.

[10] Haydarlou, A.R. and Hartel, P.H. Thunk-lifting: Reducing heap usage in an implementation of a lazy functional language. 14.

[11] Hofmann, M. and Jost, S. 2003. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices*. 38, 1 (Jan. 2003),

185–197. DOI:https://doi.org/10.1145/640128.604148.

[12] Hudak, P. et al. 2007. A history of Haskell: Being lazy with class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III* (San Diego, California, 2007), 12–1–12–55.

[13] Hutton, G. 2016. *Programming in Haskell*. Cambridge University Press.

[14] Kulal, S. et al. *Space leaks exploration in Haskell*. Indian Institute of Technology, Bombay.

[15] LaToza, T.D. et al. 2020. Explicit programming strategies. *Empirical Software Engineering*. (Mar. 2020). DOI:https://doi.org/10.1007/s10664-020-09810-1.

[16] Lipovača, M. 2011. *Learn you a Haskell for great good! A beginner's guide*. No Starch Press.

[17] Liu, H. and Hudak, P. 2007. Plugging a Space Leak with an Arrow. *Electronic Notes in Theoretical Computer Science*. 193, (Nov. 2007), 29–45. DOI:https://doi.org/10.1016/j.entcs.2007.10.006.

[18] Marlow, S. 2013. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. O'Reilly.

[19] Mitchell, N. 2013. Leaking Space. *Queue*. 11, 9 (Sep. 2013), 10–23. DOI:https://doi.org/10.1145/2538031.2538488.

[20] Moalla, F.F. 2015. *Fresh Techniques for Memory Profiling of Lazy Functional Programs*. University of York.

[21] Pinpointing space leaks in big programs: 2011. *http://blog.ezyang.com/2011/06/pinpointing-space-leaks-in-big-programs/*. Accessed: 2020-02-28.

[22] Röjemo, N. and Runciman, C. 1996. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. *Proceedings of the first ACM SIGPLAN international conference on Functional programming - ICFP '96* (Philadelphia, Pennsylvania, United States, 1996), 34–41.

[23] Seqaid: Dynamic strictness control, including space leak repair: *https://hackage.haskell.org/package/seqaid*. Accessed: 2020-02-28.

[24] Seqaid : Space leak diagnostic and remedial tool: *https://fremissant.net/seqaid*. Accessed: 2020-05-15.

[25] Sergey, I. et al. 2017. Theory and Practice of Demand Analysis in Haskell.

[26] Space leak zoo: 2011. *http://blog.ezyang.com/2011/05/space-leak-zoo/*. Accessed: 2020-02-28.

[27] System.Mem.Weak - Weak Pointer: *https://hackage.haskell.org/package/base-4.8.1.0/docs/System-Mem-Weak.html*. Accessed: 2020-05-13.

[28] Takano, Y. and Iwasaki, H. 2015. Thunk recycling for lazy functional languages: Operational semantics and correctness. *Proceedings of the 30th annual ACM symposium on applied computing* (Salamanca, Spain, 2015), 2079–2086.

[29] Wadler, P. 1987. Fixing some space leaks with a garbage collector. *Software: Practice and Experience*. 17, 9 (Sep. 1987), 595–608. DOI:https://doi.org/10.1002/spe.4380170904.

[30] Without performance tests, we will have a bad time, forever: 2018. *https://www.fpcomplete.com/blog/without-performance-tests-we-will-have-a-bad-time-forever*. Accessed: 2020-02-28.

[31] Yang, E.Z. and Mazières, D. 2013. Dynamic space limits for Haskell. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14* (Edinburgh, United Kingdom, 2013), 588–598.