acmqueue

Leaking Space

Eliminating memory hogs

Neil Mitchell

A *space leak* occurs when a computer program uses more memory than necessary. 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. This article presents example space leaks and how to spot and eliminate them.

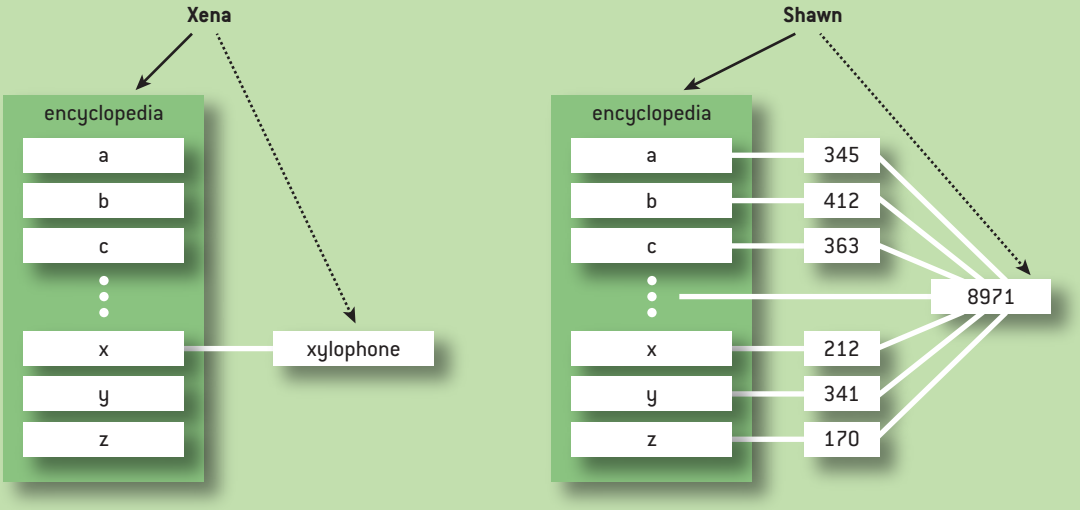
REAL-WORLD SPACE LEAKS

Let's first consider two "real-world" space leaks. Xena, the xylophone enthusiast, buys a 26-volume printed encyclopedia, but she only wants to read the article on xylophones. The encyclopedia occupies a lot of space on her bookshelf. Xena could throw away all but the X volume, reducing the shelf requirements, or she could cut out the xylophone article, leaving only a single piece of paper.

In this example, Xena is storing lots of information but is interested in only a small subset of it.

Xena's friend Shawn, the statistician, is curious about how many redundant pages Xena is storing. To determine the total number of pages in the encyclopedia Shawn buys a copy of the 26-volume encyclopedia, even though he is interested in only the number of pages per volume. Actually, Shawn doesn't need to know the sizes of 26 separate volumes but only the total size—information that could be written on the back of a stamp.

FIGURE 1 **The Information of Interest to Xena and Shawn**



In this example, Shawn is storing lots of information, and while each volume contains useful information, the result could be stored more compactly.

Figure 1 sketches the memory layout Xena and Shawn might represent if they were computer programs. In both cases a solid blue arrow points to the encyclopedia, representing the memory that is being retained. A dotted red arrow points to the information that is actually useful.

A space leak would occur if a program loaded the encyclopedia but did not immediately reduce it to the interesting part, resulting in the encyclopedia being kept in memory longer than necessary. Eliminating space leaks is about controlling when evaluation occurs, reducing the time between allocating memory and discarding it. Unsurprisingly, features that complicate evaluation order are particularly vulnerable to space leaks. The two examples this article focuses on are *lazy evaluation* (where evaluation of an expression is delayed until its value is needed) and *closures* (a function value combined with its environment). Both these features are found in lazy functional languages such as Haskell.

EXAMPLE 1: DELETE

How does lazy evaluation cause a space leak? Consider the following Haskell definition:

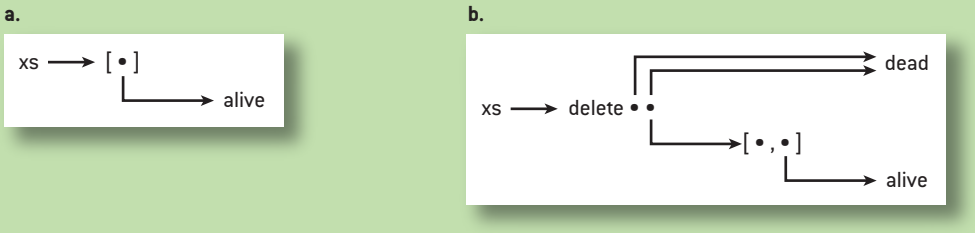
```
xs = delete dead [alive, dead]
```

This fragment creates a variable `xs` and a two-element list using the `[_,_]` notation, containing both `alive` and `dead`. Then the element `dead` is removed from the list using `delete`. A call to `length xs` returns 1, indicating there is only one element in `xs`. In the absence of lazy evaluation, the memory layout would look like figure 2a, where `xs` references a list containing `alive` as the only element; `dead` is not referenced and thus can be garbage-collected.

Haskell uses lazy evaluation (also known as call-by-need), however, so after `xs` is defined, the memory would look like figure 2b. Instead of pointing at a *value*, `xs` points at an *expression*, which may be replaced with an actual value later. There are still two paths from `xs` to `dead`; thus, `dead` cannot be garbage-collected, even though we know that it will never be used. The variable `dead` is part of a space leak because `delete` is being evaluated later than desired.

As previously mentioned, `length xs` will return 1, but as a consequence of computing the length, it will evaluate `delete`. The act of evaluating `length xs` reduces `xs` to a value, which eliminates

FIGURE 2 **Lazy Evaluation**



the space leak. A program using lists can end up with a space leak if it frequently adds and deletes elements but never uses the list to compute the length or look up values.

More generally, a space leak can occur when the memory contains an expression—where the expression grows regularly but where the evaluated value would not grow. Forcing evaluation usually solves such leaks; this makes evaluation of some variables *strict* instead of lazy.

FORCING EVALUATION

Eliminating space leaks often requires forcing the evaluation of an expression earlier than it would normally be evaluated. Before describing how to force evaluation, it is necessary to define how an expression is evaluated:

- An expression is in *normal form* if it cannot be evaluated further. For example, the list `[1,2]` is in normal form. Lists are constructed from `[]` (pronounced "nil") for the empty list, and `(:)` (pronounced "cons") to combine a head element to the tail of a list, so `[1,2]` can equivalently be written `1:2:[]`.
- An expression is in WHNF (weak head normal form) if the outermost part does not require further evaluation. For 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`. All values in normal form are by definition also in WHNF.

To force evaluation to WHNF, Haskell provides strictness annotations with the commonly used *bang patterns* extension.[3] You can define a function `output`, which prints "Output" to the console, followed by its argument:

```
output x = do
    print "Output"
    print x
```

Printing `x` evaluates `x` to normal form, so the function `output` will first print "Output", then evaluate `x` to normal form and print it. Adding an exclamation mark as a strictness annotation will force evaluation of `x` sooner:

```
output !x = ...
```

Now evaluating `output x` will first evaluate `x` to WHNF, then print "Output", then evaluate `x` to normal form and print it.

WHY LAZY?

Given that strictness avoids the space leak in Example 1—and (as shown later) several other space leaks—why not make all values strict? Certainly most languages have strict values, and even some variants of Haskell default to strict evaluation.[1] As with all language design decisions, lazy evaluation is a tradeoff—space leaks are a disadvantage, but there are also many advantages. Other articles discuss the advantages of lazy evaluation in depth,[2,6] but here, briefly, are a few reasons why it is a good choice:

- Defining new control structures in strict languages often requires macros or building them into the

compiler, while lazy evaluation allows such patterns to be expressed directly.
• Laziness allows variable bindings to be made without considering which bindings are evaluated in which code paths, a great simplification when combined with complex conditionals.
• Simulating laziness in a strict language is usually more difficult than forcing strictness in a lazy language, so laziness can be a better default.
• When combining functions, a strict language often requires the simplistic composition to take more memory than a lazy language, as will be demonstrated in Example 2.

EXAMPLE 2: SUM

Consider the following code:

```
sum [1..n]
```

In Haskell this expression creates a list containing the numbers 1 to *n*, then adds them up. In a strict language, this operation takes *O(n)* space: it would first generate a list of length *n*, then call `sum`. In a lazy language, however, the items in the list can be generated one at a time as they are needed by `sum`, resulting in *O(1)* space usage. Even if you replace `[1..n]` with numbers read from a file, the program can still run in *O(1)* space, as laziness automatically interleaves reading numbers from a file and computing the sum.

Unfortunately, this code when compiled with GHC (Glasgow Haskell Compiler) takes *O(n)* space as the result of a space leak, but when using the -O1 optimization flag takes *O(1)* space. More confusingly, for some definitions of `sum` the code takes *O(1)* at all optimization settings, and for other definitions the code always takes *O(n)*.

Why does the space leak arise? Consider the following definition of `sum`:

```
sum1 (x:xs) = x + sum1 xs
sum1 [] = 0
```

The first equation says that if the list has at least one item in it, bind the first item to `x` and the list containing the remaining items to `xs`. The sum is then defined recursively by adding the first element to the sum of the remaining elements. The second equation expresses the base case, and the sum of the empty list is 0. Let's consider evaluating `sum1 [1..n]` for some large value of `n`, which proceeds as follows:

```
sum1 [1..n]                 -- initial value
sum1 (1:[2..n])             -- sum1 requires the list
1 + sum1 [2..n]             -- sum1 reduces per the equation
1 + sum1 (2:[3..n])         -- + requires both its arguments
1 + (2 + sum1 [3..n])
```

You can trace the evaluation by looking at what the program will require next, working from the top left part of the expression. For example, initially `sum1` looks at the list to determine which expression to match: which one requires evaluating `[1..n]` to produce `1:[2..n]`. As evaluation

proceeds it builds up the term `1 + 2 + 3 + 4 ...`, taking *O(n)* space. While the program never has the whole list in memory at once, it instead has all the items of the list joined with "+" operations.

After the space leak is identified, strictness can be used to eliminate it. Given the expression `1 + 2`, it can be reduced to `3` immediately; and provided the program keeps performing the addition as the computation goes along, it will use only constant memory. Alas, with the definition of `sum1`, the expression is actually `1 + (2 + (3 ...`, meaning that `1` and `2` cannot be reduced. Fortunately, addition is associative, so `sum` can be redefined to build up `((1 + 2) + 3) ...`:

```
sum2 xs = sum2' 0 xs
    where
        sum2' a (x:xs) = sum2' (a+x) xs
        sum2' a [] = a
```

Defining `sum2` in terms of an auxiliary function `sum2'` takes an additional accumulator `a`, which is the value of all elements of the list processed so far. Tracing the evaluation looks more promising:

```
sum2 [1..n]
sum2' 0 [1..n]
sum2' 0 (1:[2..n])
sum2' (0+1) [2..n]
sum2' (0+1) (2:[3..n])
sum2' ((0+1)+2) [3..n]
```

Now literal numbers are applied to addition, but the space leak is still present. Fortunately, there is now a suitable target for a strictness annotation. You can define:

```
sum3 xs = sum3' 0 xs
    where
        sum3' !a (x:xs) = sum3' (a+x) xs
        sum3' !a [] = a
```

The strictness annotation on the accumulator argument `a` results in the accumulator being evaluated before the next element of the list is processed. Revisiting the trace

```
sum3 [1..n]
sum3' 0 [1..n]
sum3' 0 (1:[2..n])
sum3' (0+1) [2..n]
sum3' 1 [2..n]
sum3' 1 (2:[3..n])
sum3' (1+2) [3..n]
sum3' 3 [3..n]
```

shows that `sum3` takes *O(1)* space and does not have a space leak. The definition of `sum` in the standard Haskell libraries is equivalent to `sum2`; but with optimizations turned on, the compiler infers the strictness annotation, making it equivalent to `sum3`.

EXAMPLE 3: MEAN

Consider another example:

```
mean xs = sum xs `div` length xs
```

This function computes the `mean` of a list `xs` by taking the `sum` and dividing by the `length` (the backticks around `div` allow the use of a function as an infix operator). Assuming a space-leak-free definition of `sum`, how much space will `mean [1..n]` take?

Using lazy evaluation—namely, reducing the top left expression first—the answer is *O(n)*. Evaluating `sum xs` requires evaluating the entire list `xs`, but since that list is also used by `length xs`, `xs` must be retained in memory instead of being collected as it is produced.

In this example a smarter evaluation strategy could eliminate the space leak. If the program evaluated the first element of `xs`, then applied both `sum` and `length` to it, the function would take constant space. Another approach to computing `mean [1..n]` is to remove the sharing of the list:

```
sum [1..n] `div` length [1..n]
```

Here the list has been duplicated, and both arguments to `div` run in constant space, allowing the entire computation to run in constant space. Unfortunately, any work required to compute the lists will be duplicated.

The real solution is to take the pattern used for `sum3` and extend it so instead of accumulating just the sum, you also accumulate the length. The full definition is:

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

This accumulates the sum (`s`) and length (`l`) as local parameters, which are strict arguments to the helper function. The resulting definition has no space leak and runs in *O(1)*.

EXAMPLE 4: SPACE LEAKS AND THE GARBAGE COLLECTOR

The previous examples have inserted strictness annotations to eliminate space leaks. Not all space leaks can be removed by strictness annotations, however[5]; sometimes, special behavior is required from the garbage collector.[10] As an example, let's improve the impact of an academic paper by placing an exclamation mark at the end of the title:

```
improve xs = fst pair ++ "!" ++ snd pair
    where pair = firstLine xs
```

```
firstLine ('\n':ys) = ([], '\n':ys)
firstLine (y:ys) = (y:fst rest, snd rest)
    where rest = firstLine ys
firstLine [] = ([], [])
```

The `improve` function takes the source of the paper and produces a new paper. It splits the text into a variable `pair`, consisting of the first line and the remaining text, using the auxiliary function `firstLine`. The function then takes the first element of the pair using `fst`, and the second element using `snd`, and uses the string append operator "++" to insert an exclamation mark between them. The first equation of `firstLine` matches strings with a leading newline character and produces an empty first line, followed by the text. The second equation recursively calls `firstLine` with everything but the first character, then creates a result where the first character is at the front of the first line. The final equation ensures that the empty input produces empty outputs.

It should be possible for `improve` to run in *O(1)* space, producing an output character after examining each input character, and requiring only a small amount of memory. In the second equation of `firstLine`, after matching `y:ys` (i.e., consuming an input character), the program immediately produces `(y:_, _)`, making an output character available via lazy evaluation before making the recursive call. Unfortunately, using the obvious implementation techniques, this function requires space proportional to the first line of `xs`, so *O(`fst pair`)*.

To understand the space usage, consider the evaluation of `improve "abc..."`:

```
let rest4 = firstLine "..."
let rest3 = ('c':fst rest4, snd rest4)
let rest2 = ('b':fst rest3, snd rest3)
let rest1 = ('a':fst rest2, snd rest2)
'a':'b':'c':fst rest4 ++ "!" ++ snd rest1
```

In each step of `firstLine` a pair is produced whose second component is simply the second component of the recursive call. The result is both a linear chain of `snd` calls and all the characters being retained by references to the first component of each `rest` variable.

If the `snd` functions were forced, then this space leak would be eliminated to produce:

```
let rest4 = firstLine "..."
'a':'b':'c':fst rest4 ++ "!\n" ++ snd rest4
```

Unfortunately, there is nowhere to put a strictness annotation to perform the appropriate reduction. Although you want to force the evaluation of `snd`, you are also relying on the laziness of the pair in the recursive call of `firstLine` to achieve *O(1)* space. Fortunately, the garbage collector can solve this problem. The function `snd` is a selector—given a pair, it selects the second component. It does not compute any new values, does not allocate memory, and is cheap to compute. As such, the program can *evaluate* `snd` *during garbage collection*, which eliminates the space leak. The reduction of selector functions during garbage collection is now a standard feature of lazy functional languages,

automatically removing space leaks that would otherwise be impossible to eliminate.

EXAMPLE 5: SPACE LEAKS AND CLOSURES

All the examples so far have been in Haskell, but other garbage-collected languages are also susceptible to space leaks. While few languages are lazy by default, many support *closures*—a lambda expression or function, plus some variables bound in an environment. One popular language that makes extensive use of closures is JavaScript.

The following JavaScript code uses the Web Audio API[8] to retrieve an MP3 file and compute its duration:

```
function LoadAudio(mp3)
{
    // Load 'mp3' file into 'request.response'
    var request = new XMLHttpRequest();
    request.open('GET', mp3);
    request.responseType = 'arraybuffer';

    request.onreadystatechange = function(){
        if (request.readyState != 4) return;

        // Decode the audio data
        window.AudioContext = window.AudioContext || window.webkitAudioContext;
        var context = new AudioContext();
        context.decodeAudioData(request.response, function(audio){
            document.getElementById("status").onclick = function(){
                alert("MP3 is " + audio.duration + " seconds long");
            }
        });
    };
    request.send();
}
```

This function uses the `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 MP3's duration whenever a status button is clicked.

The implementation uses three local functions, two of which reference variables defined locally to `LoadAudio`. Those variables will be captured inside a closure when the local functions are referenced. As an example, the first function is assigned to `onreadystatechange` and captures the `request` variable defined three lines before.

After `LoadAudio` has run, 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—taking at least as much memory as the original MP3. The only thing ever accessed, however, is the `duration` field, which is a number, taking a mere eight bytes. The result is a space leak.

This space leak has many aspects in common with the lazy evaluation space leaks. The code references an expression `audio.duration`, which keeps alive a significant amount of memory, but when evaluated uses only a small amount of memory. As before, the solution is to force the evaluation sooner than necessary:

```
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.

JAVASCRIPT SELECTORS

While you can modify the code to eliminate the space leak, could the garbage collector have eliminated the space leak? The answer is yes, provided that `audio.duration` is cheap to compute, cannot change in the future, and will not cause any side effects. Since there are no other references to `audio`, the value to which `audio` refers cannot change; and since `audio.duration` is a read-only field, it was likely computed when the `audio` value was constructed. This optimization would be an instance of the selector evaluation from Example 4.

Unfortunately, the selector optimization is less applicable in JavaScript than in Haskell, because most values are mutable. As a small example, consider:

```
var constants = {pi : 3.142, fiveDigitPrimes : [10007,10009,10037,...]};
document.getElementById("fire").onclick = function(){
       alert(constants.pi);
};
```

This code defines a dictionary containing both `pi` (a number) and `fiveDigitPrimes` (a large array), then adds an event handler that uses `pi` only. If `constants` were immutable, then the garbage collector could reduce `constants.pi` and remove the reference to `constants`. Alas, the user can write `constants = {pi : 3}` to mutate `constants`, or `constants.pi = 3` to mutate the `pi` field, meaning evaluation in advance is unsafe.

While the difficulties of mutation mean that JavaScript does not reduce such functions in practice, it is not an insurmountable barrier. Consider a memory layout where you know which references are being used as read-only (i.e., `alert(constants.pi)`) and which are not (i.e., `constants.pi = 3`). This information can help determine which variables are used only as read-only and thus are guaranteed to be constant. If `constants` and `constants.pi` are both determined to be immutable, then the field lookup could be performed by the garbage collector, freeing both `constants` and `fiveDigitPrimes`.

In Haskell lazy evaluation is common (the default) and space leaks caused by selectors are

unavoidable, making the decision to apply selector optimization obvious. In languages such as JavaScript, adding code to solve fixable space leaks at the cost of making the normal code slower or more complex may not be a sensible tradeoff.

DETECTING SPACE LEAKS

The five examples of space leaks presented here provide some guidance as to where space leaks occur and how they can be fixed. All the examples, however, have consisted of only a handful of lines; for space leaks in big programs the challenge is often finding the code at fault. As Haskell is particularly vulnerable to space leaks, the compiler provides a number of built-in profiling tools to pinpoint the source of space leaks. Before looking at which tools are available, let's first consider which might be useful.

Space leaks are quite different from memory leaks—in particular, the garbage collector still knows about the memory referenced by the space leak and will usually free that memory before the program terminates. Assume a definition of `sum` contains a space leak; as soon as `sum` produces a result, the garbage collector will free any intermediate space leak. 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, to see what is unexpectedly retained. This technique is not applicable to space leaks.

Instead, it is often useful to examine the memory at intermediate points throughout the execution, looking for spikes in the memory usage. Capturing the entire memory at frequent intervals is likely to require too much disk space, so one solution is to record summary statistics at regular intervals, such as how much memory is allocated by each function.

HASKELL TOOLS

The Haskell compiler provides several profiling modes that generate plots summarizing memory usage. To generate a profile, first compile the program with the following flags:

```
ghc --make Main.hs -prof -fprof-auto -fprof-cafs -rtsopts
```

These flags are:

- `ghc --make Main.hs`. Compile the file `Main.hs` into an executable, as normal.
- `-prof -fprof-auto -fprof-caf`. Turn on profiling in the executable and make sure it is able to record information about top-level definitions.
- `-rtsopts`. Allow the resulting executable to accept profiling options.

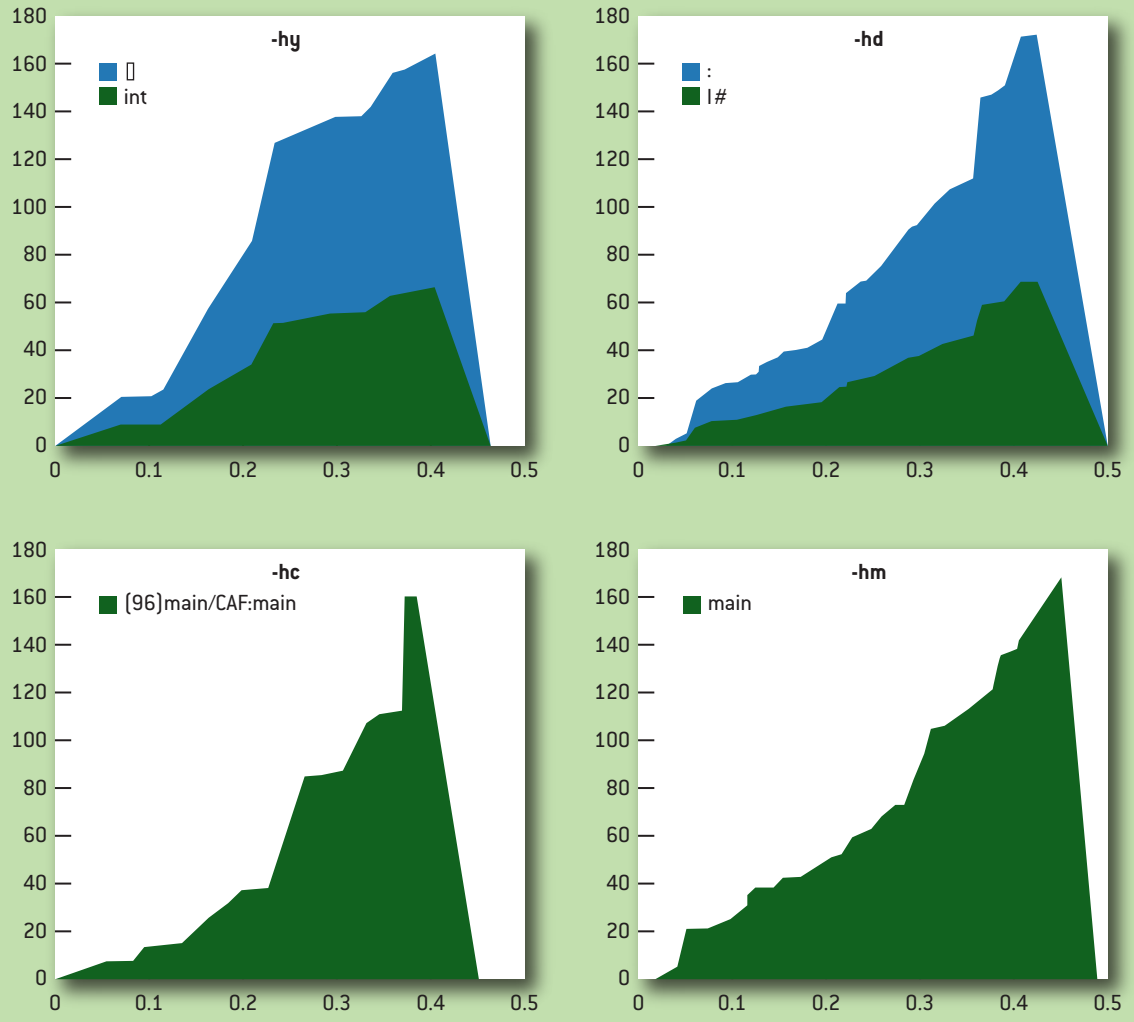
The resulting program can run as normal, but with additional flags it can also generate profile information:

```
main +RTS -xt -hy
hp2ps -c main.hp
```

Using the `mean` example presented earlier produces the first plot shown in figure 3. The X axis is time in seconds and the Y axis is memory in MB. The first command runs the resulting `main`

FIGURE 3

Profiles for the Mean Example with Different Memory Groupings



executable with some flags to the runtime system (anything after `+RTS`). The `-xt` flag includes any stack in the profile output (this author believes `-xt` should be on by default), and `-hy` generates a report summarized by type. The first command generates a file `main.hp`, and the second command turns that into a PostScript file `main.ps` (in color, due to the `-c` flag). In the plots shown I also passed `-i0.01` to sample the memory more frequently, which is usually necessary only when trying quick-running toy examples.

Haskell has a number of profiling modes, and the simplest approach is to try them all and see which produces the most useful information. The four standard types of profiles, shown in figure 3, are:

- `-hy`. Summarizes the memory by type. The example has some lists (`[]`) and numbers (`Int`). This summary answers the question of *what* is in the memory.

• `-hd`. Summarizes by description, showing a more refined version of `-hy`. In the example there is a close correspondence to `-hy`, with all `Int` entries matching `I#` (which is the internal constructor of `Int`) and lists matching `(:)`. Any group below a threshold is hidden; otherwise, there would likely be a single `[]` denoting the end of the list.
• `-hc`. Summarizes by cost center, a named area of the source code automatically inserted on all top-level definitions. It can also be manually inserted with an annotation in the code. In figure 3 `main` has been attributed all the memory, probably a result of optimization inlining `mean` inside of it. This summary answers the question of *where* the memory was created.
• `-hm`. Summarizes by module, which is a more granular version of a cost center.

From a combination of these plots you can see that the function `main` in the module `Main` allocates a large list of numbers. It allocates the list over 0.4 seconds, then quickly consumes the list over 0.1 seconds. This memory usage describes what would be expected from the original definition of `mean`.

For larger programs the plot will often contain a lot of memory usage that is expected—and not relevant to the space leak. To simplify the plot you can filter by any of the four types: for example, passing `-hc -hy[]` will generate a plot grouped by cost center but only for memory where the type is a list.

As seen in the `sum` example, compiling with different optimization settings may cause space leaks to appear or disappear, and, sadly, compiling for profiling can have similar effects (although this is relatively rare). As a fallback, any Haskell executable can be run using `+RTS -hT`, which produces a plot summarized by type without compiling for profiling. This causes fewer changes to the behavior of the program.

Before using the profiling tools, read the Profiling section of the GHC manual, which covers several additional flavors of profiling. For a better idea of how the profiling tools can be applied to large programs and how to interpret the results, I recommend the following two "tales from the trenches" from Edward Yang and myself:
• http://blog.ezyang.com/2011/06/pinpointing-space-leaks-in-big-programs/
• http://neilmitchell.blogspot.com/2013/02/chasing-space-leak-in-shake.html

JAVASCRIPT TOOLS

One tool Haskell lacks is the ability to pause execution at a certain point and explore the memory. This feature is available in some JavaScript implementations, including in Chrome as the heap profiler.

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. You can summarize by the type of object, see statistics about how much memory is consumed and referenced by a certain value, and filter by name. A feature particularly useful for diagnosing space leaks is the ability to see what references are keeping a value alive. The two JavaScript space leaks in this article produce heap snapshots that easily pinpoint the problem.

ARE SPACE LEAKS INEVITABLE?

Garbage collection frees programmers from the monotony of manually managing memory, making it easier for languages to include advanced features such as lazy evaluation or closures. These advanced features lead to more complex memory layout, making it harder to predict what memory

looks like, potentially leading to space leaks.

Compilers for lazy functional languages have been dealing with space leaks for more than 30 years and have developed a number of strategies to help. There have been changes to compilation techniques and modifications to the garbage collector and profilers to pinpoint space leaks when they do occur. Some of these strategies may be applicable to other languages. Despite all the improvements, space leaks remain a thorn in the side of lazy evaluation, producing a significant disadvantage to weigh against the benefits.

While space leaks are worrisome, they are not fatal, and they can be detected and eliminated. The presence of lazy evaluation has not stopped Haskell from being used successfully in many projects (you can find many examples in the conference proceedings of the Commercial Users of Functional Programming). While there is no obvious silver bullet for space leaks, there are three approaches that could help:

• Some complex problem domains have libraries that eliminate a large class of space leaks by design. One example is Functional Reactive Programming, which is used to build interactive applications such as user interfaces and sound synthesizers. By changing how the library is defined you can both guarantee certain temporal properties and eliminate a common source of space leaks.[7] Another example is stream processing, which is used heavily in Web servers to consume streams (e.g., a JavaScript file) and produce new streams (e.g., a minimized JavaScript file) without keeping the whole stream in memory. Several competing stream libraries are available for Haskell. All of them ensure that memory is retained no longer than necessary and that the results are streamed to the user as soon as possible.

• Space leaks are often detected relatively late in the development process, sometimes years after the code was written and deployed, and often only in response to user complaints of high memory usage. If space leaks could be detected earlier—ideally, as soon as they are introduced—they would be easier to fix and would never reach end users. Certain types of advanced profiling information can detect suspicious memory patterns,[9] and some experimental tools can annotate expected heap usage,[4] but nothing has reached mainstream use. The Haskell compiler does partition memory in such a way that some space leaks are detected—the `sum` example above fails with a message about stack overflow for lists of length 508146 and above, but the other examples in this article use all available memory before failing.

• The tools for pinpointing space leaks are powerful but certainly not perfect. An interactive viewer can explore existing plots,[11] but users are still required to specify how the memory is grouped before running the program. It would be much easier if all four groupings could be captured at once. A feature missing from Haskell programs is the ability to take a snapshot of the memory to examine later, which would be even more powerful if combined with the ability to take a snapshot when memory exceeded a certain threshold. Pinpointing space leaks is a skill that takes practice and perseverance. Better tools could significantly simplify the process.

REFERENCES

1. Augustsson, L. 2011. Pragmatic Haskell. Presentation at the CUFP (Commercial Users of Functional Programming) Conference; http://www.youtube.com/watch?v=hgOzYZDrXL0.
2. Augustsson, L. 2011. More points for lazy evaluation. Things that Amuse Me; http://augustss.blogspot.co.uk/2011/05/more-points-for-lazy-evaluation-in.html.

3. Glasgow Haskell Compiler Team. 2013. The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3; http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html.
4. Hofmann, M., Jost, S. 2003. Static prediction of heap space usage for first-order functional programs. *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*: 185-197.
5. Hughes, J. 1983. The design and implementation of programming languages. Ph.D. thesis. Oxford University.
6. Hughes, J. 1989. Why functional programming matters. *Computer Journal* 32(2): 98-107.
7. Liu, H., Hudak, P. 2007. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science* 193: 29-45.
8. Rogers, C. 2012. Web Audio API; http://www.w3.org/TR/2012/WD-webaudio-20120802/.
9. Röjemo, N., Runciman, C. 1996. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming (ICFP)*: 34-41.
10. Wadler, P. 1987. Fixing some space leaks with a garbage collector. *Software: Practice and Experience* 17(9): 595-608.
11. Yang, E. 2013. hp/D3.js; http://heap.ezyang.com/.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

NEIL MITCHELL is a Haskell programmer who works for Standard Chartered Bank. He obtained his Ph.D. in functional programming from the University of York in 2008. His open source projects include a search engine (Hoogle), a code suggestion tool (HLint), and a library for writing build systems (Shake).