I accidentally the entire heap

Or: how I learned to stop worrying and love the profiler.

Trevor Caira

Bitbase LLC

June 26, 2013

Topic

Laziness and Space Leaks

- This is not a highly theoretical talk
- I will be giving practical advice on how to debug space leaks caused by lazy evaluation

Overview

- First I will briefly review how Haskell programs are evaluated by example
- Then I will go over a plausible example of debugging a program with space leaks

Topic

Whirlwind tour

- Principles governing Haskell evaluation are simple
- Still, a proper treatment is the subject of its own talk
- Consult the Haskell wiki article on Graph Reduction

Non-strictness

- Haskell values are non-strict
- Not evaluated until actually needed
- That is, only pattern matching or I/O primitives can cause evaluation

Evaluation Relationship

- All evaluation in a Haskell program is ultimately rooted at the `main` top-level binding
- Strictness is most usefully thought of as a relationship

```
case xs of
  [] -> True
  _  -> False
```

- This expression is strict in xs
  - But not in the head or tail of xs
  - Only the outer cons cell is evaluated
  - Still yields a value given an infinite list

Laziness

- Non-strictness in GHC is done with laziness + sharing
- Laziness means expressions are suspended in thunks
  - Thunk = eventual value or non-termination
- Sharing means expressions are evaluated once per name

```
square x = x * x
main = print $ square (fibs !! 10000)
```

- `fibs !! 10000` is only computed once

Simple Example

Let's consider the evaluation of

```
and (repeat False)
```

- Like any Haskell expression, it starts out life as a thunk
- When evaluated, we get:

```
and <thunk: repeat False>
```

- I represent thunks in angle brackets with the code that yields their value

Simple Example (cont'd)

To move forward, let's consult the definition of `and`:

```
and xs = case xs of
           [] -> True
           (h:t) -> h && and t
```

Given our evaluated value so far:

```
and <thunk: repeat False>
```

Substituting the definition of `and` yields us:

```
case <thunk: repeat False> of
  [] -> True
  (h:t) -> h && and t
```

Simple Example (cont'd)

This is a pattern match. We must evaluate our thunk. Consulting the definition of `repeat`:

```
repeat x = x : repeat x
```

Given our evaluation so far:

```
case <thunk: repeat False> of
  [] -> True
  (h:t) -> h && and t
```

Substituting the definition of `repeat` yields:

```
case <thunk: False> : <thunk: repeat False> of
  [] -> True
  (h:t) -> h && and t
```

Simple Example (cont'd)

Here, the second pattern matches:

```
(h:t) -> h && and t
```

Substituting the variables *h* and *t* yields:

```
<thunk: False> && <thunk: and <thunk: repeat False>>
```

Simple Example (cont'd)

Now we're ready to apply `(&&)`, given below:

```
x && y = case x of
           True -> y
           False -> False
```

to our value so far:

```
<thunk: False> && <thunk: and <thunk: repeat False>>
```

yielding:

```
case <thunk: False> of
  True -> <thunk: and <thunk: repeat False>>
  False -> False
```

Simple Example (cont'd)

Evaluating this thunk yields `False`, of course matching the second pattern.

- Note that second argument to `(&&)` is never evaluated.
- Since we never evaluate the tail of `repeat False`, the program terminates.

Wrapping Up Evaluation

This was only a simple example to motivate the feel of programming with laziness. Much more information is available in the Haskell report and Haskell wiki.

Topic

A Concrete Example

Let's explore the challenges of laziness with a concrete, believable example.

Problem Statement

- Task at hand: build and query an inverted index
- An inverted index maps content (words) to documents
- We'll build a record-level inverted index
- Spoiler alert: laziness will get in our way

Inverted Index

- Map of terms to documents they occur in
- `AND` query is the intersection of documents referenced by the terms in the query

Topic

Starting out

We'll make a first attempt at building an inverted index starting with an obvious, naive implementation.

Data Model

```
type Term = String
```

We model terms (and documents) with `String`.

```
type Index = Map Term (Set FilePath)
```

An index is simply a map of terms to the set of documents they occur in.

String

Recall the definition of `String`:

```
type String = [Char]
```

Simply a (lazy) linked list of characters.

Index Creation

```
indexDocument :: Index -> FilePath -> Term -> Index

createIndex :: [FilePath] -> IO Index
createIndex documents =
  foldM addDocument Map.empty documents
  where addDocument :: Index -> Term -> IO Index
        addDocument index document = do
          contents <- readFile document
          return (indexDocument index document contents)
```

- We implement construction of the index as a monadic fold over the documents.
- The accumulating function reads each document and adds it to the index.

Monadic Fold

Remember, `foldM` has the following type:

```
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
```

- Just like `foldl` except the function argument yields a monadic value.

Document Indexing

```
segmentTerms :: Term -> [Term]

indexDocument :: Index -> FilePath -> Term -> Index
indexDocument index docPath contents =
    Map.unionWith Set.union index .
    Map.fromList .
    map (\term -> (term, Set.singleton docPath)) .
    segmentTerms $ contents
```

The document is indexed by splitting the document into words with `segmentTerms` and inserting a pointer from each word in the document back to the document's path.

Processing the Documents

```
segmentTerms :: Term -> [Term]
segmentTerms contents =
    words . map toLower .
    filter (\c -> isSpace c || isAlpha c) $
    contents
```

We filter out non-alpha characters, normalize to lower case, and split the document into words.

Querying the Index

- Once we have the index, we can efficiently perform our query.
- This is accomplished by intersecting the sets of documents which contain each term:

```
queryIndex :: Index -> [Term] -> [FilePath]
queryIndex index query =
    Set.toList . intersections .
    mapMaybe lookupTerm $ query
    where lookupTerm term =
              Map.lookup (map toLower term) index
```

Compile and Run

If we run a simple query on a medium-sized corpus, we get. . .

Compile and Run (cont'd)

```
openFile: resource exhausted (Too many open files)
```

Culprit: readFile

```
readFile :: FilePath -> IO String
readFile name = openFile name ReadMode >>= hGetContents
```

Looks like hGetContents should be cleaning up our file handles, but…

Lazy I/O!

```
hGetContents :: Handle -> IO String
```

> *Computation* `hGetContents hdl` *returns the list of characters corresponding to the unread portion of the channel or file managed by hdl,* ***which is put into an intermediate state, semi-closed****. A semi-closed handle becomes closed . . .* ***once the entire contents of the handle has been read****.*

Lazy I/O! (cont'd)

- Lazy I/O means that evaluating pure code can have I/O side effects
  - Fully evaluating the (pure) list causes the file handle to be deallocated
- We accumulate a big pile of unevaluated Strings in our `foldM`
- This program leaks file handles!

Topic

Read strictly

- Let's just slurp in the whole file each time
- This way `hGetContents` will clean up after us
- No more semi-closed handles

How do we accomplish this?

```
readFile' :: FilePath -> IO String
readFile' path =
    do contents <- readFile path
       seq (length contents) (return contents)
```

This forces the entire file to be read before moving on to the next file.

Strictness with seq

- seq is our strictness primitive
- Evaluates its first argument and returns its second
- $seq \perp x = \perp$

Update our code

Replace the use of `readFile` with our new function in `addDocument`.

```
addDocument index document = do
  contents <- readFile' document
  return (indexDocument index document contents)
```

- Let's run it: `./Stage2 docs know between together`

Performance

- It doesn't crash!
- But it's slow...
- `RSIZE` in `top` is 2,316M
  - Input documents only total 28M
  - Something is amiss

Memory Debugging

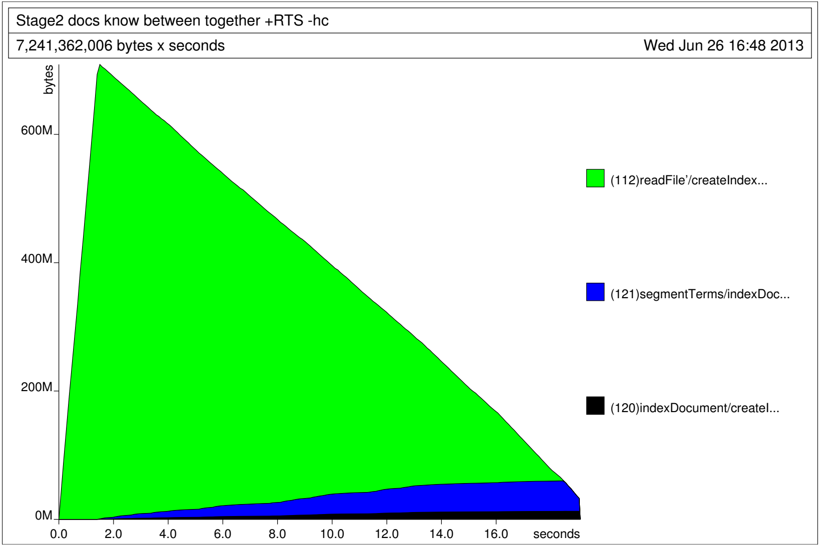
- GHC ships with fantastic memory profiling tools
- Make sure your libraries are compiled with profiling enabled
- Add this to your `$HOME/.cabal/config`:

```
library-profiling: True
```

Memory Debugging (cont'd)

- Compile with `-prof -fprof-auto`
- We want to see a timeline of allocations broken out by who allocated them
- This is given to us by the cost-centre heap profile
- Re-run with `+RTS -hc` to produce the heap profile output

Heap Profile



Heap Profile (cont'd)

- Each color corresponds to a different source of allocation
- Other annotation methods are available (e.g. `-hy` breaks out by type)
- Interpreting this graph requires reasoning about the program's course of execution

Heap Profile (cont'd)

- We can see that all of the documents are read in with `readFile` at the beginning, and are slowly deallocated as they are indexed
- We want to deallocate each file after it is read in before moving on to the next one

Strict folding

```
addDocument index document = do
  contents <- readFile' document
  return (indexDocument index document contents)
```

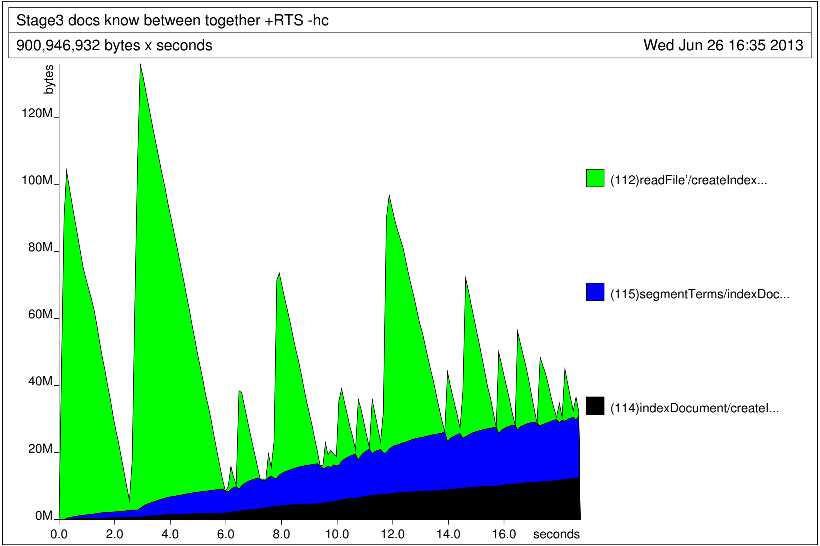
- `foldM` is strict in accumulator
  - (or at least as strict as >>=)
- We are using the strict `Map` variant
- But the `Map` in the accumulator is lazy

Strict folding (cont'd)

```
addDocument index document = do
  contents <- readFile' document
  let index' = indexDocument index document contents
  seq index' (return index')
```

- `return` is lazy
- We need to use the same strategy as with `readFile`
- Ensure the index is evaluated at each step

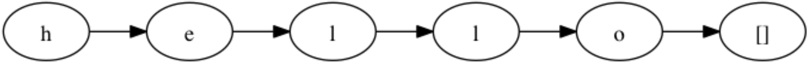
Heap Profile



Topic

Strings are bad

- Strings are extremely inefficient
- Linked list of characters
    - Each character is lazy
    - Each character has its own lazy cons cell



- Enter: `Data.Text`

Data.Text

An efficient packed Unicode text type.

- Written by Bryan O'Sullivan
- Widely used, highly optimized
- Dense UTF-16 array representation

Update our code

Let's update our representation with strict `Text`:

```
import Data.Text (Text)
```

```
type Term = Text
```

`text` also packages a strict, locale-sensitive `readFile`, obsoleting our strict `readFile` replacement. Let's update `addDocument`:

```
addDocument index document = do
  contents <- Text.readFile document
  let index' = indexDocument index document contents
  seq index' (return index')
```

Update our code (cont'd)

Now let's replace our `Data.Char` methods with their more efficient `Data.Text` equivalents:

```
segmentTerms :: Term -> [Term]
segmentTerms contents =
    Text.words . Text.toLower .
    Text.filter (\c -> isSpace c || isAlpha c) $
    contents
```

And in `queryIndex`:

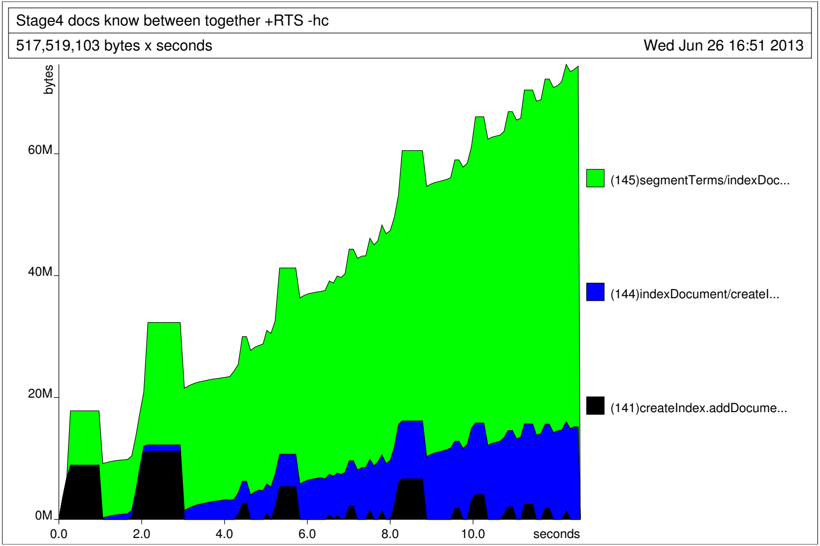
```
lookupTerm term = Map.lookup (Text.toLower term) index
```

Compile and run

Now our code should use a tiny fraction of the memory owing to the far more efficient text representation.

- Let's run it and find out:
  `./Stage4 docs know between together +RTS -hc`

Heap Profile



What's going on?

- This is reminiscent of our first heap profile
- It is a fraction of the size, but clearly asymptotically incorrect
- We didn't change the strictness of our program
    - It is building the index as it reads the files

Cracking the code

- Our biggest hint is in the cost centre that is leaking
- Note that it's not `readFile`...

Data.Text revisited

- `segmentTerms` is holding the references to the bulk of the heap
- Isn't `Text.words` breaking up the big block of text and letting the GC do its job?

`words`

- In fact, `words` as provided by `Data.Text` provides *views* onto the source array
- Rather than copying the entire string, the list of words share a reference to the same array
- This is done for efficiency
- But what if we want copying?

copy

They thought of that, too!

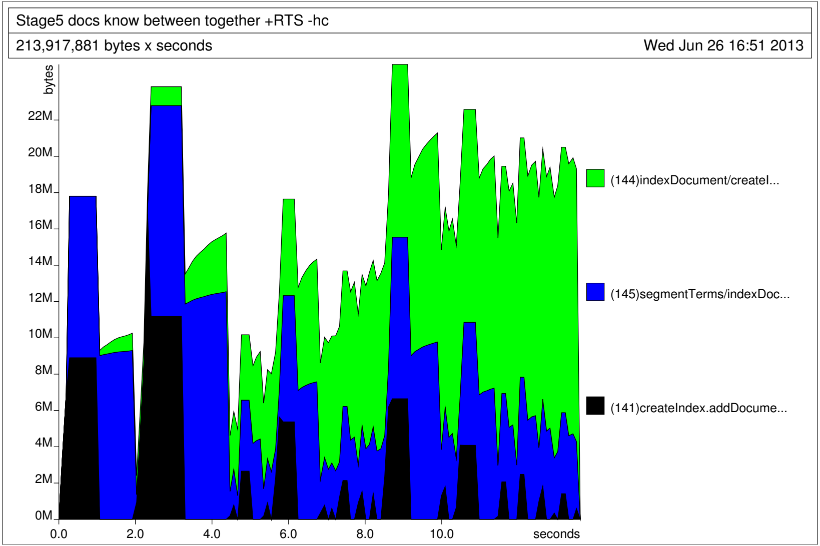
```
copy :: Text -> Text
```

> *O(n) Make a distinct copy of the given string, sharing no storage with the original string.*

Let's add it in:

```
segmentTerms :: Term -> [Term]
segmentTerms contents =
    map Text.copy .
    Text.words . Text.toLower .
    Text.filter (\c -> isSpace c || isAlpha c) $
    contents
```

Heap Profile



Conclusion

- Wonderful! An asymptotic improvement
- Our spikes are on the order of the size of individual documents
- We can't do much better than this asymptotically

Thank You!

- Brought to you by Bitbase
- We do Haskell consulting
- Slides are available at https://github.com/bitbasenyc/heap