Optimisation is the art of making something faster

- Desire: It must go too slow
- Benchmark: You must know how fast it goes
- Profile: You must know what to change

Fast XML Parsing with Haskell – Neil Mitchell
Fast XML Parsing with Haskell

Neil Mitchell

http://ndmitchell.com

@ndm_haskell

+ Christopher Done
System Optimisation

• Optimisation folk lore
  – 90% of the time is spent running 100 lines
  – Optimise those 100 lines and profit

Warning: After a few rounds of optimisation, your profile may be mostly flat
The Problem

• Parse XML to a DOM tree and query it for tags/attributes

```xml
<conference title="Haskell eXchange" year=2017>
  <talk author="Gabriel Gonzalez">
    Scrap your Bounds Checks with Liquid Haskell
  </talk>
  <talk author="Neil Mitchell">
    Fast XML parsing with Haskell
    <active/> <!-- remove this in 30 mins -->
  </talk>
</conference>
```
Existing Solutions

- xml – 100x-300x slower
- hexpat – 40x-100x slower
- xml-conduit – much slower
- tagsoup – SAX based
- XMLParser
- xmlhtml
- xml-pipe
- PugiXML: C++ library, fastest by a lot
  – Haskell binding segfaults 😞
PugiXML Tricks

• Extremely fast – faster than all others
  – 9x faster than libxml
  – 27x faster than msxml
  – Closest are asmxml (x86 only), rapidxml
  – “Parsing XML at the Speed of Light”

• Ignore the DOCTYPE stuff (no one cares)

• Does not validate

• In-place parsing
Our Tricks

• Ignore the DOCTYPE stuff (no one cares)
• Does not validate
• In-place parsing (even more so)
• Don’t expand entities e.g. &lt;
  – All returned strings are offsets into the source
  – In body text, only care about <, so memchr

• Hexml: Haskell friendly C library + wrapper
• Xeno: Pure Haskell alternative
# Haskell inner loops

<table>
<thead>
<tr>
<th>C</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security!!!!!</td>
<td>Security!</td>
</tr>
<tr>
<td>Painful allocation</td>
<td>Implicit allocation</td>
</tr>
<tr>
<td>Marshalling</td>
<td>INLINE and -O2</td>
</tr>
<tr>
<td>No abstractions</td>
<td>Many abstractions</td>
</tr>
<tr>
<td>Single lump</td>
<td></td>
</tr>
<tr>
<td>Less familiar</td>
<td></td>
</tr>
<tr>
<td>Verbose</td>
<td></td>
</tr>
<tr>
<td>Undefined behaviour</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td>Segfaults</td>
<td></td>
</tr>
</tbody>
</table>
Approach 1: C inner loops
Hxml

https://hackage.haskell.org/package/hxml
Hexml Memory

Document (C, block alloc)

Node

Attr

Text (Haskell, ByteString)

Points at substring

Allocated inside
typedef struct {
    int32_t start;
    int32_t length;
} str;

typedef struct {
    str name;  // tag name, e.g. <[foo]>
    str inner; // inner text, <foo>[bar]</foo>
    str outer; // outer text, [<foo>bar</foo>]
} node;
Hexml Interface (functions)

document* document_parse(const char* s, int slen);
char* document_error(const document* d);
void document_free(document* d);
node* document_node(const document* d);

attr* node_attributes(const document* d, const node* n, int* res);
attr* node_attribute(const document* d, const node* n, const char* s, int slen);
How did I get to that?

- I’ve written FFI bindings before, so know what is hard/slow, and avoided it!
  - Simple memory management (only document)
  - Functions are relatively big – where possible known structs are used
  - Use ByteString because it is FFI friendly (C ptr)

- Intuition and experience matters...
  - (My excuse for not using a simple example)
Wrapping Haskell (types)

data Str = Str {  
    strStart :: Int32,  
    strLength :: Int32  
}  

instance Storable Str where  
    sizeof _ = 8  
    alignment _ = alignment (0 :: Int64)  
    peek p = Str <$> peekByteOff p 0 <*> peekByteOff p 4  
    poke p (Str a b) = pokeByteOff p 0 a >>= pokeByteOff p 4 b
Wrapping Haskell (functions)

```c
data CDocument
data CNode

foreign import ccall document_parse :: CString -> CInt -> IO (Ptr CDocument)
foreign import ccall "&document_free" document_free :: FunPtr (Ptr CDocument -> IO ())
foreign import ccall unsafe document_node :: Ptr CDocument -> IO (Ptr CNode)
```
Wrapping Haskell (memory)

- Document is not on the Haskell API (pretend it’s a node)
- A node must know about the text of it, the document it is in, and the node itself

```haskell
data Node = Node
            BS.ByteString
            (ForeignPtr CD documento)
            (Ptr CNode)
```
Creating Node

parse :: BS.ByteString -> Node
parse src = unsafePerformIO $
  BS.unsafeUseAsCStringLen src $(str, len) -> do
  doc <- document_parse str (fromIntegral len)
  doc <- newForeignPtr document_free doc
  node <- document_node doc
  return $ Node src doc node
Using Node

```
attr* node_attributes(const document* d, const node* n, int* res);
node_attributes :: Ptr CDocument -> Ptr CNode -> Ptr CInt -> IO (Ptr CAttr)

attributes :: Node -> [Attribute]
attributes (Node src doc n) = unsafePerformIO $
   withForeignPtr doc $ \d ->
   alloca $ \count -> do
   res <- node_attributes d n count
   count <- fromIntegral <$> peek count
   return [attrPeek src doc $ plusPtr res $ i*szAttr
       | i <- [0..count-1]]
```
The big picture

• Define some simple functions types in C
  – Wrap them to Haskell almost mechanically

• Define some types in C
  – Wrap them to Haskell in a context specific way

• Wrap the functions into usable Haskell
  – Requires smarts to get them looking right
  – Requires insane attention to detail to not segfault

• Note we haven’t shown the C code!
Continuing onwards

• Testing can and should be in Haskell
  – Explicit test cases based on errors
  – Property based testing
  – Wrote a renderer, checked for idempotence
    – parse . render === id

• Debugging C by printf is super painful
  – I used Visual Studio for interactive debugging
  – Used American Fuzzy Lop for fuzzing (thanks Austin Seipp)
Results

• Fast! ~2x faster than PugiXML
• Simple! Nice clean interface
• Abstractable! hexml-lens puts lenses on top
• But ran into...
  – Undefined behaviour in C
  – Buffer read overruns in C
  – Incorrect memory usage in Haskell
• All removed with blood, sweat and tears
Approach 2: Haskell inner loops

Xeno

https://hackage.haskell.org/package/xeno

Christopher Done, now Marco Zocca
Approach

• Hexml: Think hard and be perfect
• Xeno: Follow this methodology
  – Watch memory allocations like a hawk
  – Start simple, benchmark
  – Add features, rebenchmark
  – Build from composable pieces
Simplest possible

```
parseTags :: ByteString -> Int -> ()  -- walk a document
parseTags str i
    | Just i <- findNext '<' str i
    , Just i <- findNext '>' str (i+1)
    = parseTags str (i+1)
    | otherwise = ()

findFirst :: Char -> ByteString -> Int -> Maybe Int
{-# INLINE findNext #-
findFirst c str offset = (+ offset) <$> 
    BS.elemIndex c (BS.drop offset str)
```
## Timing

<table>
<thead>
<tr>
<th>File</th>
<th>hexml</th>
<th>xeno</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>6.395 μs</td>
<td>2.630 μs</td>
</tr>
<tr>
<td>42KB</td>
<td>37.55 μs</td>
<td>7.814 μs</td>
</tr>
</tbody>
</table>

- Basically measuring C `memchr` function
  - Plus bounds checking!
- Shows Haskell is not adding huge overhead

[https://hackage.haskell.org/package/criterion](https://hackage.haskell.org/package/criterion)
Memory

<table>
<thead>
<tr>
<th>Case</th>
<th>Bytes</th>
<th>GCs</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>4kb parse</td>
<td>1,168</td>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>42kb parse</td>
<td>1,560</td>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>52kb parse</td>
<td>1,168</td>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>182kb parse</td>
<td>1,168</td>
<td>0</td>
<td>OK</td>
</tr>
</tbody>
</table>

- Memory usage is linear – not per <> pair
- Don’t we allocate a Just per <>?

https://hackage.haskell.org/package/weigh
Watching the Just

parseTags str i
    | Just i <- findNext '<' str i

{-# INLINE findNext #-}
findNext c str offset = (+ offset) <$>
    BS.elemIndex c (BS.drop offset str)

{-# INLINE elemIndex #-}
BS.elemIndex str x =
    let q = memchr str x
    in if q == nullPtr then Nothing else Just $( str - q)
Is ‘Just’ expensive?

• A single Just requires:
  – Heap check (comparison, one per function)
  – Alloc (addition)
  – Construction (memory writes)
  – Examination (memory reads, jump)
  – GC (expensive, one every so often)

• Not “expensive”, just not free
Incrementally add bits

• Parse comments, tags, attributes
• Return results

• At each step:
  – Benchmark (will slow down a bit)
  – Memory (should remain zero)

• Tricks
  – INLINE, -O2, alternative functions
Making it useful

parseTags

:: (s -> ByteString -> s)
-> ByteString -> Int -> s
-> Either XenoException s

parseTags fTag str l s

| Just i <- findNext '<' str l = case findNext '>' str (i+1) of
  Nothing -> Left $ XenoParseError "mismatched <"
  Just j -> parseTags fTag str (i+1) $ fTag s $ BS.substr (i+1) j
| otherwise = Right s

Xeno specialises to a Monad and uses impure exceptions. Does that make it go faster or slower?
SAX Parser

fold

:: (s -> ByteString -> s) -- ^ Open tag.
-> (s -> ByteString -> ByteString -> s) -- ^ Attribute.
-> (s -> ByteString -> s) -- ^ End of open tag.
-> (s -> ByteString -> s) -- ^ Text.
-> (s -> ByteString -> s) -- ^ Close tag.
-> s
-> ByteString
-> Either XenoException s
DOM Parser

• Can be built on top of the SAX parser
  – Beautiful abstraction in action

• Harder problem
  – Can’t aim for zero allocations
  – Need a smart compact data structure
  – Need ST, STURef, vector
## Xeno vs Hexml

<table>
<thead>
<tr>
<th>File Size</th>
<th>hexml-dom</th>
<th>xeno-sax</th>
<th>xeno-dom</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>6.123 μs</td>
<td>5.038 μs</td>
<td>10.35 μs</td>
</tr>
<tr>
<td>31KB</td>
<td>9.417 μs</td>
<td>2.875 μs</td>
<td>5.714 μs</td>
</tr>
<tr>
<td>211KB</td>
<td>256.3 μs</td>
<td>240.4 μs</td>
<td>514.2 μs</td>
</tr>
</tbody>
</table>
# Haskell inner loops

<table>
<thead>
<tr>
<th>C</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security!!!!!</td>
<td>Security!</td>
</tr>
<tr>
<td>Painful allocation</td>
<td>Implicit allocation</td>
</tr>
<tr>
<td>Marshalling</td>
<td>Many abstractions</td>
</tr>
<tr>
<td>No abstractions</td>
<td>INLINE and -O2</td>
</tr>
<tr>
<td>Single lump</td>
<td>GHC version tuning</td>
</tr>
<tr>
<td>Less familiar</td>
<td>Slower</td>
</tr>
<tr>
<td>Verbose</td>
<td>Ongoing compromise</td>
</tr>
<tr>
<td>Undefined behaviour</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td>Segfaults</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• C is up front design, Haskell is feedback
• Haskell can use better abstraction and security
• C is a lot harder than I remember
• Haskell FFI is exceptionally good

I personally prefer C inner loops to Haskell
DOM Storage

• Now onto a smart representation/algo
  – Haskell and C share the same ideas
  – C inner loops requires DOM storage also in C

• Needs to be compact
  – Store attributes and nodes in single alloc

• Easier to describe in C?
DOM Attributes

typedef struct
{
    int size; // number used
    int used; // number available, doubles
    attr* attrs; // dynamically allocated buffer
    attr* alloc; // what to call free on
} attr_buffer;

Buffer that doubles on reallocation
Plus fast path for special allocation
typedef struct {
    const char* body; // pointer to initial argument
    // not owned by us
    char* error_message;
    node_buffer nodes;
    attr_buffer attrs;
} document;

Nothing interesting
DOM Creation

typedef struct
{
    document document;
    attr attrs[1000];
    node nodes[500];
} buffer;

Alloc a buffer, point document.nodes at buffer.nodes
If resizing, just ignore the memory
1 allocation for 3 buffers
DOM Nodes

typedef struct {
    int size;
    int used_front; // front entries, stored for good
    int used_back; // back entries, stack based, copied into front
    node* nodes; // dynamically allocated buffer
    node* alloc; // what to call free on
} node_buffer;

Want all DOM children to be adjacent (compact)
What about nested children?
Copy to the end of the buffer, then commit
Resizing needs to copy too
static inline bool is(char c, char tag)
{
    return table[(unsigned char) c] & tag;
}

C is hard: [1/7]
if (get peek(d) != '=')
{
    set_error(d, "Expected = in attribute, but missing");
    return start_length(0, 0);
}
skip(d, 1);
C is hard: [3/7]

attributeBy (Node src doc n) str =
  unsafePerformIO $ withForeignPtr doc $ \d ->
  BS.unsafeUseAsCStringLen str $ \(bs, len) -> do
    r <- node_attributeBy d n bs $ fromIntegral len
    touchForeignPtr $ fst3 $ BS.toForeignPtr src
  return $ if r == nullPtr then Nothing
    else Just $ attrPeek src doc r

Use after free
C is hard: [4/7]

let src0 = src <> BS.singleton '\0'
...
return $ Node src0 doc node
C is hard: [5/7]

d->nodes.nodes[0].nodes = parse_content(d);

str content = parse_content(d);
d->nodes.nodes[0].nodes = content;
C is hard: [6/7]

if (peek_at(d, -3) == '-' &&
    peek_at(d, -2) == '---')
while (1 d->error_message == NULL)

if (d->error_message != NULL) return;
c = get(d);