Fast XML Parsing with Haskell

Neil Mitchell

http://ndmitchell.com

+ Christopher Done

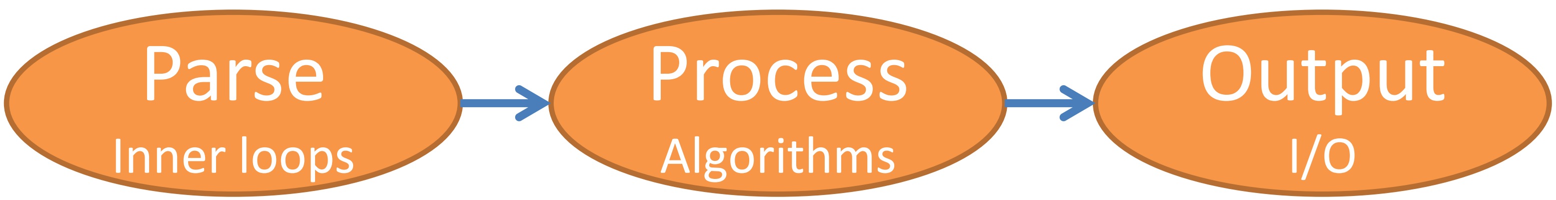
CONTENT DISCLAIMER

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

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

Haskell inner loops

C	Haskell
Security!!!!!	Security!
Painful allocation	Implicit allocation
Marshalling	INLINE and -O2
No abstractions	Many abstractions
Single lump	
Less familiar	
Verbose	
Undefined behaviour	
Portability	
Segfaults	

The Problem

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

```xml
<conference title="Haskell eXchange" year=2017>
  <talk author="Julie Moronuki">
    A Monoid for All Seasons
  </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. &
  - All returned strings are offsets into the source
  - In body text, only care about >, so memchr

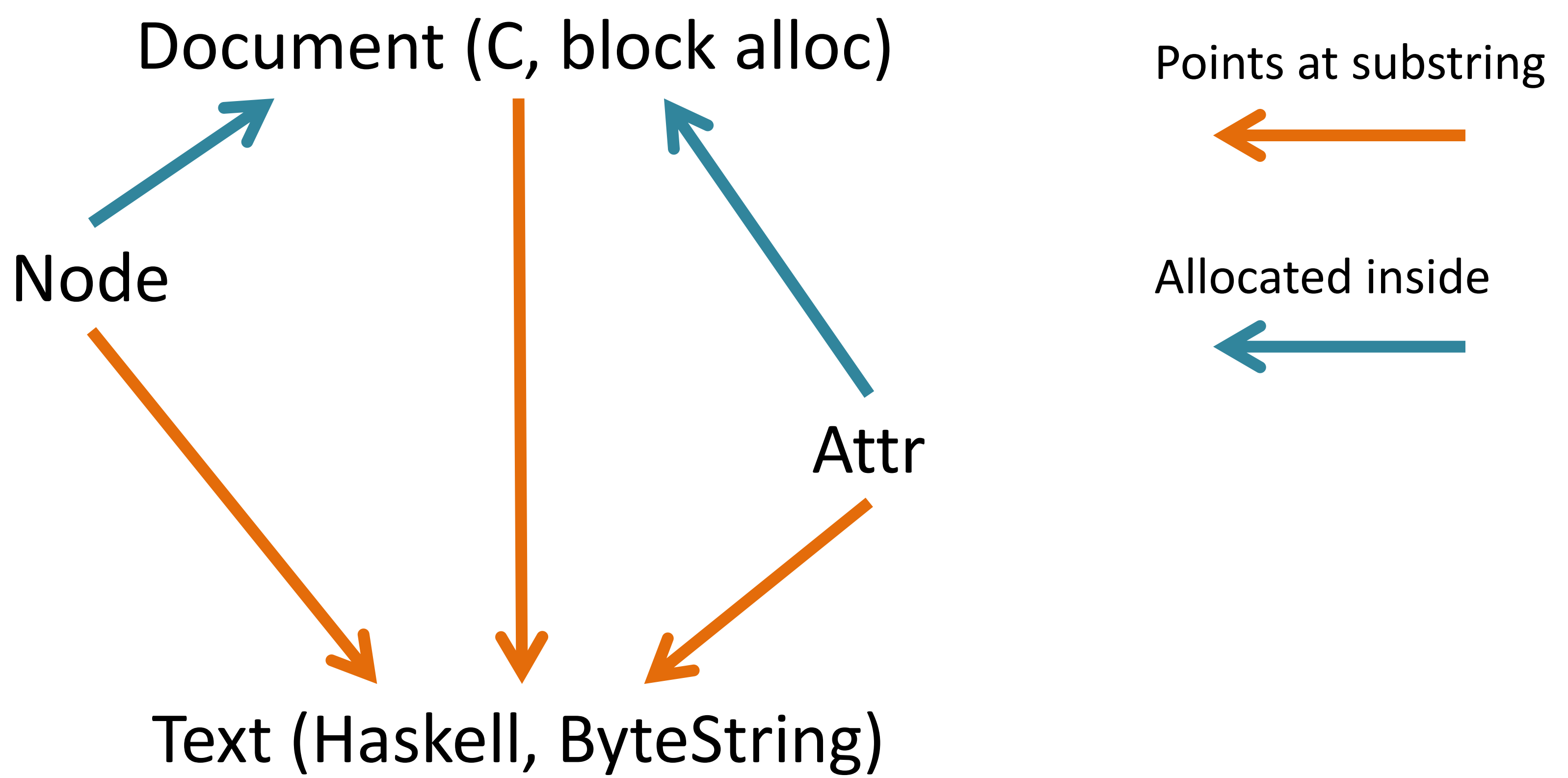
- Hexml: Haskell friendly C library + wrapper
- Xeno: Pure Haskell alternative

C

Approach 1: C inner loops
Hexml

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

Hexml Memory



C

Hexml Interface (types)

```
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;
```

C

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);
```

C

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)

C

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
```

```
typedef struct
{
    int32_t start;
    int32_t length;
} str;
```

Wrapping Haskell (functions)

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

```
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)
```

C

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

```
data Node = Node
    BS.ByteString
    (ForeignPtr CDocument)
    (Ptr CNode)
```

C

Creating Node

```
parse :: BS.ByteString -> 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
```

C

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]]
```

C

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
- Debugging C by printf is super painful
  - I used Visual Studio for interactive debugging
  - Used American Fuzzy Lop for fuzzing

Results

- Fast! ~2x faster than PugiXML
- Simple! Nice clean interface
- Abstractable! hexml-lens has lenses on top
- But ran into…
  - Undefined behaviour in C
  - Buffer 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 = ()

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

Timing

File	hexml	xeno
4KB	6.395 μs	2.630 μs
42KB	37.55 μs	7.814 μs

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

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

Memory

Case	Bytes	GCs	Check
4kb parse	1,168	0	OK
42kb parse	1,560	0	OK
52kb parse	1,168	0	OK
182kb parse	1,168	0	OK

- 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 I s
    | Just i <- findNext '<' str I = 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

File	hexml-dom	xeno-sax	xeno-dom
4KB	6.123 μs	5.038 μs	10.35 μs
31KB	9.417 μs	2.875 μs	5.714 μs
211KB	256.3 μs	240.4 μs	514.2 μs

Haskell inner loops

C	Haskell
Security!!!!!	Security!
Painful allocation	Implicit allocation
Marshalling	Many abstractions
No abstractions	INLINE and -O2
Single lump	GHC version tuning
Less familiar	Slower
Verbose	Ongoing compromise
Undefined behaviour	
Portability	
Segfaults	

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

DOM Document

```
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

C is hard: [1/7]

```
static inline bool is(char c, char tag)
{
    return table[(unsigned char) c] & tag;
}
```

Portability

Out of bounds read

C is hard: [2/7]

```
if (get peek(d) != '=')
{
    set_error(d, "Expected = in attribute, but missing");
    return start_length(0, 0);
}
skip(d, 1);
```

Incorrect result

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
```

Use after free

C is hard: [5/7]

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

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

Undefined behaviour

Unportable

Use after free

C is hard: [6/7]

```
if (peek_at(d, -3) == '-' &&
    peek_at(d, -2) == '-')
```

Incorrect result

C is hard: [7/7]

```
while (1 d->error_message == NULL)


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

Out of bounds read