Better, faster binary serialization

A case study in low level optimization in Haskell

Duncan Coutts

May 2015, ZuriHac —

Well-Typed
The Haskell Consultants

Point of this talk

Case study in optimising a Haskell library

Not supposed to be about how great this new library is, but necessarily lots of details about the problem and the library.

Illustrate a number of **tricks and techniques** that (hopefully) are **transferable.**

Binary serialisation

Binary serialisation use cases

Two different use cases for reading/writing binary data

- serialisation of Haskell values
- externally defined formats

The `binary` and `cereal` packages do both (**confusing!**)

This talk is about the **serialisation of Haskell values** case

The new library

Short term goals

- standalone library (`binary-serialise-cbor`)
- much better serialisation format
  - schema-free decoding
  - allow for versioning/migration
  - more compact
- significantly faster

Longer term goals

- replace the serialisation layer in the `binary` package
  - consolidate `binary` and `cereal` packages
  - have more packages provide instances

The new binary serialisation format

Called “CBOR”

- “Concise Binary Object Representation”, RFC 7049

Think **JSON** but in **binary**

- self-describing low level structure
- like JSON but proper low level types
  - sensible numbers (signed, unsigned and floating)
  - byte strings and Unicode strings
- compact encoded size ($\approx 1/2$ of `binary` and `cereal`)
- suited to Haskell’s tree-like data (unlike e.g. protobufs)

CBOR values

CBOR “diagnostic notation” is much like JSON

Examples

```
10
100
""
"a"
[]
[1, 2, 3]
[1, [2, 3]]
```

Encoding Haskell values

```
data T = C1 Int Int
       | C2 String
       | C3
```

We will encode constructors as CBOR arrays, with a constructor tag

Haskell value	CBOR notation
C1 3 4	[0, 3, 4]
C2 "hi"	[1, "hi"]
C3	[2]

Serialised CBOR

Conceptually, a serialised CBOR value **tree** is represented by a flat **sequence** of (binary) **tokens**

Examples

CBOR notation	Tokens (no official notation)
$[1, 2, 3]$	ListLen 3, Int 1, Int 2, Int 3
$[1, [2, 3]]$	ListLen 2, Int 1, ListLen 2, Int 2, Int 3

Each token has a binary encoding

Serialisation strategy

Never build an **intermediate** CBOR value structure

- straight from Haskell value to serialised CBOR tokens
- straight from serialised CBOR tokens to Haskell value

Write instances in terms of **flattening** to and from CBOR **tokens**

Use printer and parser combinators specialised to binary CBOR tokens

Serialisation strategy

Never build an **intermediate** CBOR value structure

- straight from Haskell value to serialised CBOR tokens
- straight from serialised CBOR tokens to Haskell value

Write instances in terms of **flattening** to and from CBOR **tokens**

Use printer and parser combinators specialised to binary CBOR tokens

Contrast

With `aeson` you convert to/from generic `Value` JSON structure

Intermediate structures

Intermediate structures have significant performance consequences

- intermediate structure often larger than final structure
- may be necessary to build entire intermediate structure before any conversion to final structure
- cost of allocating, converting and GCing large structure

Real life example, ide-backend package

Decoding large JSON (RPC) values with aeson caused huge memory spikes

Switched to binary package, memory spikes gone

Serialisation strategy example

```
data T = C1 Int Int
       | C2 String
       | C3
```

C1 3 4	[0, 3, 4]
C2 "hi"	[1, "hi"]
C3	[2]

Serialise and deserialise code would look something like

```
instance Serialise T where
  encode (C1 x y) = elistLen 3 <> eword 0 <> eint x <> eint y
  encode (C2 s)   = elistLen 2 <> eword 1 <> estring s
  encode C3       = elistLen 1 <> eword 2

  decode = do
    len ← dlistLen; tag ← dword
    case tag of
      0 → expect (len ≡ 3) ≫ C1 <$> dint <⋆> dint
      1 → expect (len ≡ 2) ≫ C2 <$> dstring
      2 → expect (len ≡ 1) ≫ pure C3
```

Serialisation performance

Existing `binary`/`cereal` approach

`binary` and `cereal` packages embody the traditional approach

- continuation-based builder monoid
- continuation-based binary parser monad
- both shallow embeddings
- generates code that directly manipulates buffers

Both packages look ok on **microbenchmarks** but less good on typical **big tree-like values**

- only marginally faster than `aeson` on encoding

Existing `binary`/`cereal` approach

`binary` package's Builder monoid

```
newtype Builder = Builder {
   runBuilder :: (Buffer → IO L.ByteString)
              → Buffer
              → IO L.ByteString
   }
data Buffer = Buffer !(ForeignPtr Word8) !Int !Int !Int
```

Continuation style, passing a Buffer around.

Why is it so slow?

Hard to figure out definitive answers

- Micro-examples are often OK (varies with GHC version)
- Real scale examples perform badly but are hard to analyse

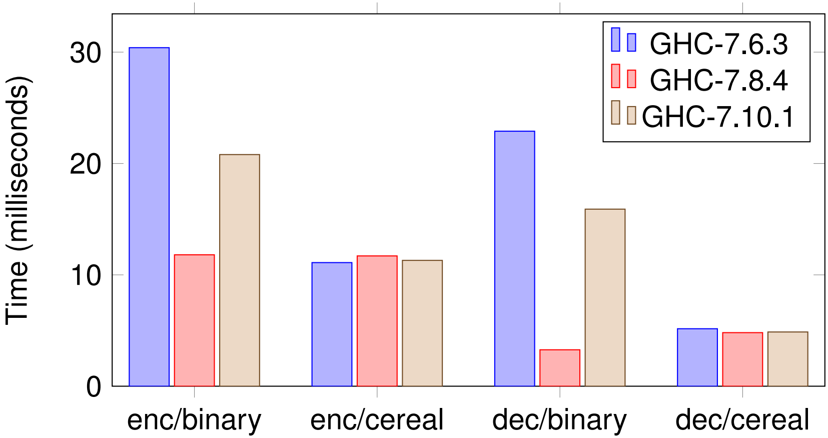
It generates a **lot** of code.

Writing an optimal code **generator** is harder than writing optimal code.

Every buffer check needs a resume continuation (which duplicates code).

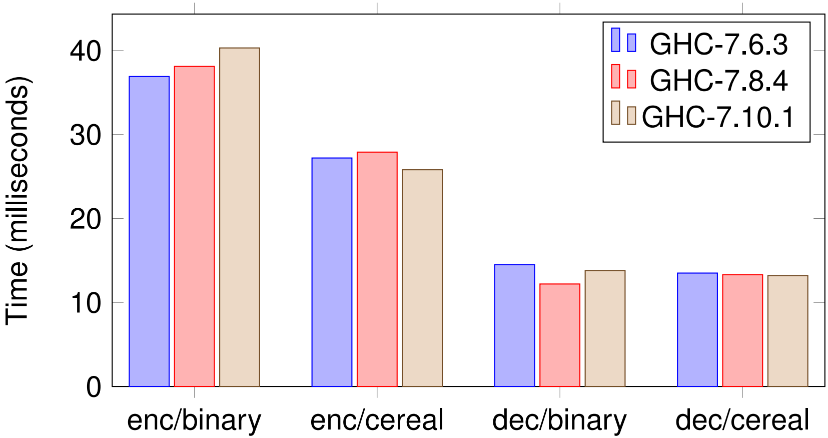
Allocates many `Buffer` and `PS` structures.

Trouble with micro-benchmarks



- Requires per-GHC tuning
- Either great core or obvious (fixable) nonsense

Trouble with macro-benchmarks



- Less GHC variation
- Not great core
- But too much code to pin down precisely the culprit

A crazy alternative idea

Simon Meier had a crazy idea:

use a **deep embedding** to separate **description** of token sequence from **encoding** of final binary data

- instances provide token sequence
- interpreter converts token sequence to binary blobs

Crazy because a deep embedding is an **intermediate structure**!

A crazy alternative idea

Simon Meier had a crazy idea:

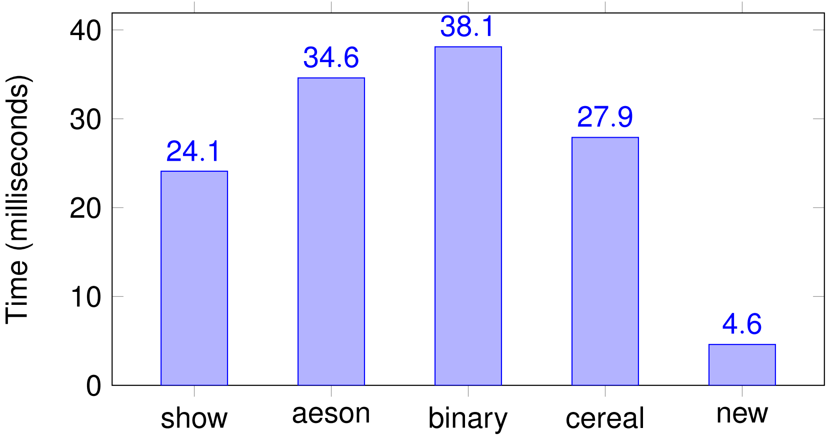
use a **deep embedding** to separate **description** of token sequence from **encoding** of final binary data

- instances provide token sequence
- interpreter converts token sequence to binary blobs

Crazy because a deep embedding is an **intermediate structure**!

But actually about **5x faster**!

Encoding shootout on macro-benchmark



- Most libs are slower than show! Embarrassing!
- New code 5x faster than show, 6x faster than cereal.

Describing the flattened token sequence

```
data Tokens = TkWord   !Word Tokens
            | ...    -- for each token
            | TkEnd

newtype Encoding = Encoding (Tokens → Tokens)
instance Monoid Encoding    -- standard d-list style

encodeWord :: Word → Encoding
encodeWord n = Encoding (λts → TkWord n ts)
```

Instances just provide these tokens

```
encode (C1 x y)  =  encodeListLen 3 <> encodeWord 0
                 <> encodeInt x <> encodeInt y
```

The generated code is nice and simple

Interpreter for token sequences

```
import qualified Data.ByteString.Builder                as B
import qualified Data.ByteString.Builder.Internal       as BI
import qualified Data.ByteString.Builder.Prim.Internal as PI

toBuilder :: Encoding → B.Builder
toBuilder = λ(Encoding vs0) → BI.builder (step (vs0 TkEnd))
  where
    step vs1 k (BI.BufferRange op0 ope0) =
      go k vs1 op0 ope0

    go k vs ! op
      | op `plusPtr` bound ⩽ ope0 = case vs of
        TkWord x vs′ → PI.runB wordMP x op >>= go vs′
        ...
```

We can go as low level as we need.
This uses `bytestring` Builder's low level API

The “compiler” approach to performance

Combinators using shallow embedding

- combinators defined separately
- have to be assembled at use sites
- have to be optimised to produce near-perfect code for every specific case

Two-stage programming

- input to GHC simplifier & rule engine
- plus ‘normal’ runtime code

Like writing a compiler but with much less control

The “interpreter” approach to performance

Pay the **cost** of the intermediate (lazy) structure
move **as much work as possible to the interpreter side**
optimise the interpreter as much as possible

Combinators using deep embedding

- combinators just provide data description
- almost nothing to optimise at use sites

Interpreter(s)

- a single chunk of code, not code to assemble code
- in workflow for optimising, can study **one** chunk of core
- bigger code complexity budget, can spend on more tricks
- can use more code, one interpreter but N use sites

Deserialisation

Serialisation is easy! What about deserialisation?

The deep embedding trick worked really well for serialisation and was **relatively straightforward** to implement

What about deserialisation? Can we use a similar approach?

Serialisation is easy! What about deserialisation?

The deep embedding trick worked really well for serialisation and was **relatively straightforward** to implement

What about deserialisation? Can we use a similar approach?

Would require a **description** of the binary token parser.

Mixed deep/shallow embedding

Can also use a mixed deep/shallow approach

- description containing little snippets of code specialised at the use site
- i.e. data constructors containing functions
- interpreter would call the compiled functions

Can provide some of the advantages of both approaches

- compiled code for low level bits that are use-site-specific and slow to interpret
- interpreted code for the “bigger things” like structure and control flow

A deep embedding for deserialisation

First go, start simple

```
data Decoder a = Decoder {
      runDecoder :: ∀r.(a → DecodeAction r) → DecodeAction r
   }
instance Monad Decoder    -- standard continuation monad

data DecodeAction a
    = ConsumeToken (TermToken → DecodeAction a)
    | Fail String
    | Done a

data TermToken =
      TkWord  ! Word
    | ...     -- for each token
```

Every consume token primitive uses a function (shallow embedding) for what to do next, but returning a new description (deep embedding)

Interpreter for the deep embedding

```
go (ConsumeToken k) !bs
    | BS.length bs ⩾ maxTokenSize    -- plenty of space
    = let hdr = BS.head bs in
      case decodeToken hdr bs of
        ResultToken       sz tok → go (k tok) (BS.drop sz bs)
        ResultTokenString sz len → ...
        ResultTokenBytes  sz len → ...
        ResultFail        msg    → ...
```

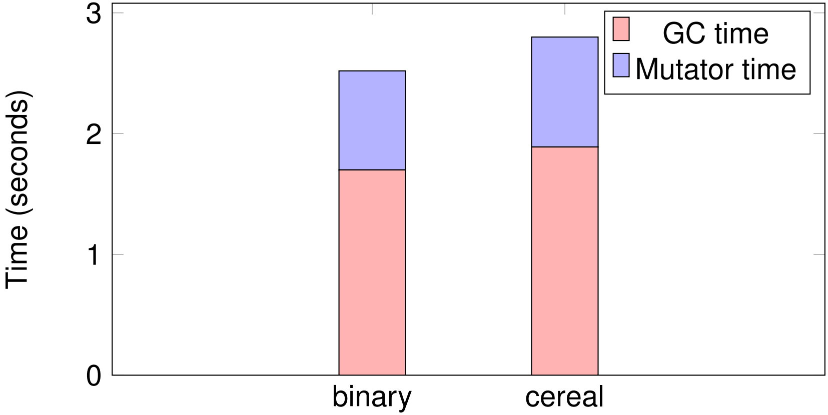
Fast path is fairly small

Costs

- calling continuation function
- allocating token constructors

Suspicions about allocations

Allocation in decoding

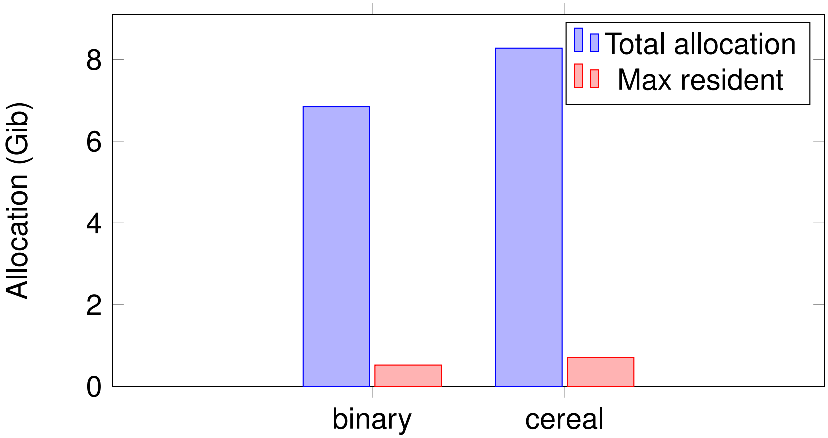


Larger example, deserialising to ∼0.5G in-mem data

GC cost is a major factor in deserialisation

Well-Typed

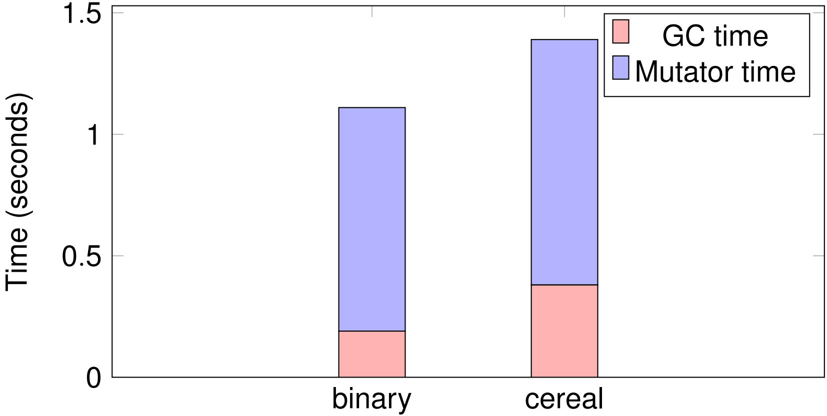
Allocation in decoding



Allocating **huge** amounts of garbage during deserialisation

So allocate less and go faster?

Allocation in decoding

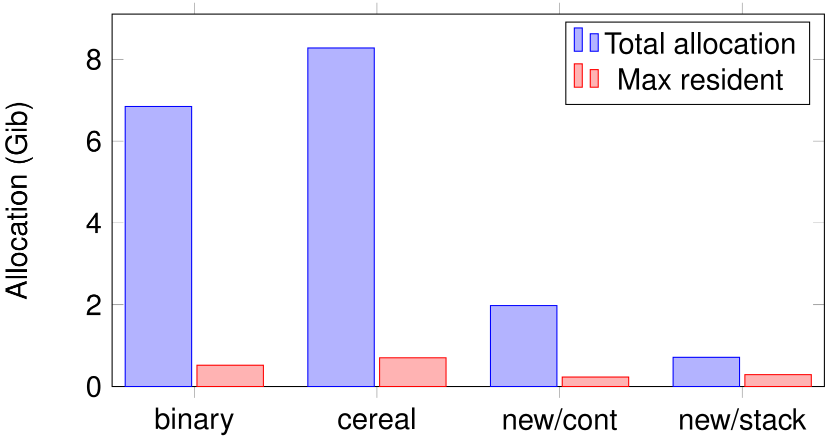


Same example, but not accumulating result in memory
(still deserialising everything)

Major GC cost appears to be repeatedly scanning the structure being accumulated

Well-Typed

Allocation in the interpreter style

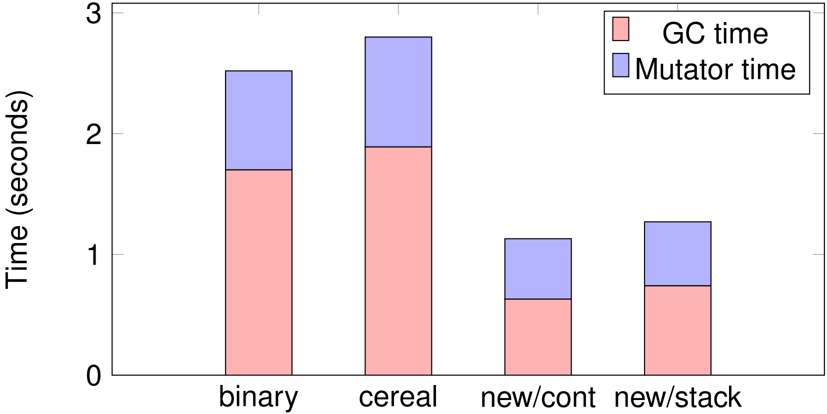


Two new interpreter style implementations

Yes we can radically reduce allocations

Will explain how . . .

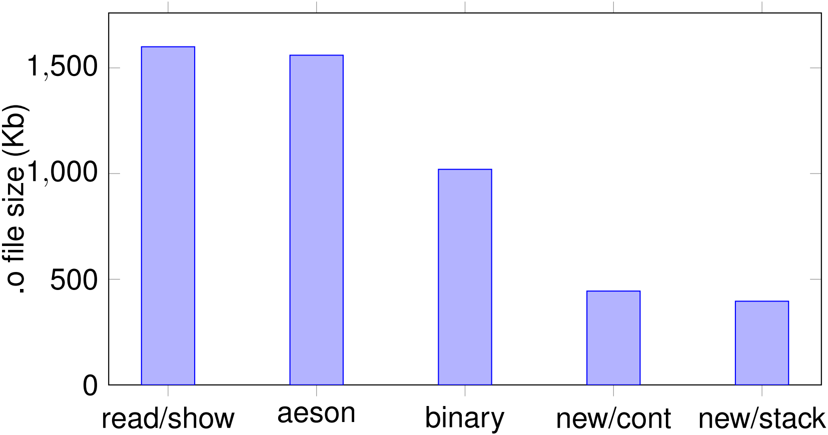
Allocation in the interpreter style



Even less allocation does not translate into even less GC time

Unclear why so far

Object code size



Much smaller object code for interpreter approaches

Reducing allocations

Spotting allocations

Where should we look for allocations?

How can we check if we're doing badly or well?

Spotting allocations

```
go (ConsumeToken k) ! bs
  | BS.length bs ⩾ maxTokenSize    -- plenty of space
  = let hdr = BS.head bs in
    case decodeToken hdr bs of
      ResultToken       sz tok → go (k tok) (BS.drop sz bs)
      ResultTokenString sz len → ...
      ResultTokenBytes  sz len → ...
      ResultFail        msg    → ...
```

Why is the tok token allocated but the ResultToken not?

How do we know?

Spotting allocations

```
go (ConsumeToken k) ! bs
  | BS.length bs ⩾ maxTokenSize    -- plenty of space
  = let hdr = BS.head bs in
    case decodeToken hdr bs of
      ResultToken       sz tok → go (k tok) (BS.drop sz bs)
      ResultTokenString sz len → ...
      ResultTokenBytes  sz len → ...
      ResultFail        msg    → ...
```

Why is the `tok` token allocated but the `ResultToken` not?

How do we know?

Either “just know” (e.g. spotting case-of-case opportunity)

Or look and verify . . .

Looking at allocations in STG code

STG is GHC core code in a special stylised low-level form.

Use GHC with `-ddump-stg -dsuppress-all`

In STG, every allocation is either a **let** or simple constructor application (and all **let**s of boxed types are an allocation)

Constructor allocation

```
let {
  v = TkWord! [x]
} in ...
```

Looking at allocations in STG code

STG is GHC core code in a special stylised low-level form.

Use GHC with `-ddump-stg -dsuppress-all`

In STG, every allocation is either a **let** or simple constructor application (and all **let**s of boxed types are an allocation)

Closure allocation

let $\{$
  k = λ_r srt:SRT:[] [x] -- explicit lambda args
    $\ldots$ -- closure body with captured vars in scope
$\}$ **in** ConsumeToken [k]

Eliminating the easy allocations

Relatively easy to eliminate allocation of most common tokens

```
data Decoder a
   = ConsumeToken       (TermToken → Decoder a)
   | ConsumeTokenWord (Word#         → Decoder a)
   | ...    -- others if necessary
   | Fail String
   | Done a
```

Provide special case for common tokens using unboxed types.

Eliminating closure allocations

```
decode = do
  t1 ← consumeToken
  t2 ← consumeToken
  return (TwoToks t1 t2)
```

For code like this we will see generated STG like

```
decode =
  λr srt:SRT:[] [k]
    let {clos1 = λr srt:SRT:[] [t1]
                   let {clos2 = λr srt:SRT:[] [t2]
                                  let {res = TwoToks ! [t1 t2];
                                  } in k res;
                   } in ConsumeToken [clos2];
    } in ConsumeToken [clos1];
```

When to look for closure allocations

When should we expect closure allocations?

- function or unevaluated value passed as arg
  - e.g. arg of data constructor
- where function/expression captures a var from the local environment

Note that a continuation argument is often captured

Continuation arg + data constructors containing functions gives lots of closures.

Can we do anything about all the closures?

Can we do anything about all the closures we allocate?

No general technique. It depends on application area.

Can we do anything about all the closures?

Can we do anything about all the closures we allocate?

No general technique. It depends on application area.

In this case yes! Use a stack!

Instead of

```
t1 ← consumeToken
t2 ← consumeToken
return (TwoToks t1 t2)
```

Values passed in closures

How about something like

```
pushToken
pushToken
alter (λt1 t2 → TwoToks t1 t2)
```

Values passed in interpreter state

A stack is not enough!

Just using a stack isn't enough to eliminate all the closure allocations

If we want to embed functions in constructors without allocation, the functions **must be closed terms**.

- such that they could be top level functions
- so must not capture vars from the environment

```
data Decoder a = Decoder {
    runDecoder :: ∀r.(a → DecodeAction r) → DecodeAction r
  }
```

Every Decoder we write like this takes a k contination arg

Must also eliminate the continuation argument

Deep embedding for composition

Need a different approach to composition

Rather than relying on continuation/d-list style to sort out the order of the primitive operations, include sequencing explicitly in the description

```
data Decoder (?) =
      Done
    | Sequence Decoder Decoder
    | PushToken Decoder
    | ...    -- other ops
```

This does allow **fully static** descriptions of stack programs

Deep embedding for composition

Can get core code like

```
d1 = PushToken d2
d2 = Sequence decodeSomethingElse d3
d3 = AlterStack f d4
d4 = Done
```

All constructors allocated at compile time

No allocation while traversing

Cyclic descriptions (loops) still possible

Following the stack approach

Types types types

The stack must be heterogeneous

Must keep track of the stack type

pushToken goes from ‘stack s’ to ‘stack Token on top of s’

Types types types

The stack must be heterogeneous

Must keep track of the stack type

pushToken goes from ‘stack s’ to ‘stack Token on top of s’

```
pushToken :: Decoder s (Token :*: s)
data a :*: b
```

Well-Typed

Types types types

The stack must be heterogeneous

Must keep track of the stack type

`pushToken` goes from 'stack `s`' to 'stack `Token` on top of `s`'

```
pushToken :: Decoder s (Token :*: s)
data a :*: b
```

Composition then has type

```
(>>>) :: Decoder a b → Decoder b c → Decoder a c
```

Types types types

Which fits the Category class

```haskell
class Category cat where
  id  :: cat a a
  (∘) :: cat b c → cat a b → cat a c

(>>>) = flip (∘)

instance Category Decoder where
  id  = Done
  (∘) = flip Sequence
```

Can also think of this as a **type-indexed monoid**

We'll use a type **indexed monad** in a sec . . .

Well-Typed

Chaining operations

So `Decoder` has to look like

$$
\begin{array}{l}
\textbf{data}\ \text{Decoder s s}'\ \textbf{where} \\
\quad \text{Done} \quad\quad\ :: \text{Decoder s s} \\
\quad \text{Sequence} :: \text{Decoder a b} \to \text{Decoder b c} \to \text{Decoder a c} \\
\quad \ldots
\end{array}
$$

For primitive `Decoder` ops we would prefer to chain directly, rather than use `Sequence` everywhere. Fewer indirections for the interpreter. Faster.

$$
\begin{array}{l}
\textbf{data}\ \text{Decoder s s}'\ \textbf{where} \\
\quad \ldots \\
\quad \text{PushToken} :: \text{Decoder (Token} :\!*\!: \text{s) s}' \to \text{Decoder s s}'
\end{array}
$$

But this doesn't match our `(>>>)` operator

Well-Typed

Chaining operations

$\mathsf{PushToken} :: \mathsf{Decoder}\ (\mathsf{Token} :\!*\!: \mathsf{s})\ \mathsf{s'} \to \mathsf{Decoder}\ \mathsf{s}\ \mathsf{s'}$

We can fit the type we want

$\mathsf{pushToken} :: \mathsf{Decoder}\ \mathsf{s}\ (\mathsf{Token} :\!*\!: \mathsf{s})$
$\mathsf{pushToken} = \mathsf{PushToken}\ \mathsf{Done}$

But now we don't get the chaining we wanted! Arggh!

Chaining operations

$$\mathsf{PushToken} :: \mathsf{Decoder}\ (\mathsf{Token} :\!*\!: \mathsf{s})\ \mathsf{s}' \rightarrow \mathsf{Decoder}\ \mathsf{s}\ \mathsf{s}'$$

We can fit the type we want

$$\begin{array}{l} \mathsf{pushToken} :: \mathsf{Decoder}\ \mathsf{s}\ (\mathsf{Token} :\!*\!: \mathsf{s}) \\ \mathsf{pushToken} = \mathsf{PushToken}\ \mathsf{Done} \end{array}$$

But now we don't get the chaining we wanted! Arggh!

But we can use GHC's rewrite rules!

```
RULES "Sequence/PushToken"
        ∀d d′.Sequence (PushToken d) d′
           =  PushToken (Sequence d d′)
RULES "Sequence/Done"
        ∀d.Sequence Done d = d
```

Chaining operations

```
RULES "Sequence/PushToken"
      ∀d d′.Sequence (PushToken d) d′
          =  PushToken (Sequence d d′)

RULES "Sequence/Done"
      ∀d.Sequence Done d = d
```

Will inline and rewrite

```
     pushToken
>>>  pushToken
>>>  d
```

to

```
PushToken (PushToken d)
```

Well-Typed

The alternative

Deep embedding of sequencing, and RULES for local improvement seems messy?

Alternative was the d-list approach

```
newtype Decoder s s′ =
  Decoder (∀s″.DecodeAction s″ s′ → DecodeAction s s′)
```

Inlining eliminates the continuation locally, but `k` still an argument to everything and so everything is a closure

Two approaches:

- potentially perfect but relies on inlining/specialising everywhere to work
- imperfect but optimise locally, no catastrophic optimisation failures

First approach particularly susceptible to problem when you try to abstract

Altering the stack

In Serialise instances we wanted something like:

```
    pushToken
>>> pushToken
>>> alter (λt1 t2 → TwoToks t1 t2)
```

Want a function/action that directly manipulates the stack

We use a type indexed monad

```
newtype StackEval st st′ a
```

The st, st′ args keep track of the shape of the stack

Return result is pushed back onto the stack

- means StackEval actions cannot grow the stack
- so no need for overflow checks

Altering the stack

So the real code looks like

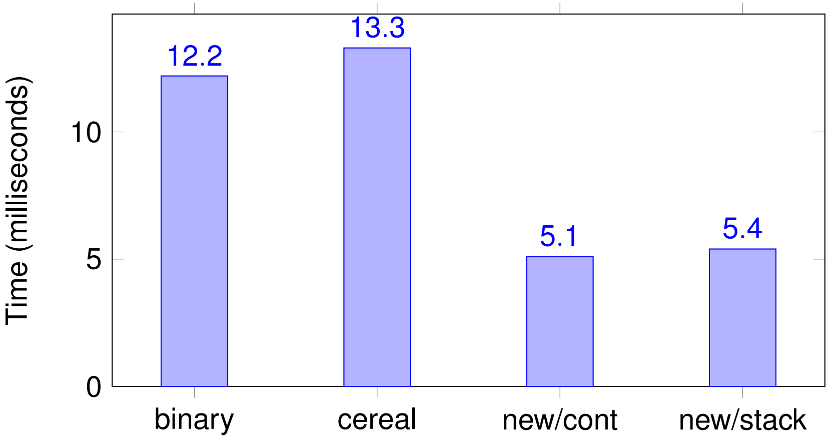
```
    pushToken
>>> pushToken
>>> alter (pop >>>= λt2 →
          pop >>>= λt1 →
          result (TwoToks t1 t2))
```

Note that it is a stack, so pop t2 then t1 !

Sadly no class in core libs for indexed monad,
nor syntactic support.

So does it work?

Decoding shootout on macro-benchmark



- Continuation and stack approach similar speed
- Both over 2x faster than binary.
- Stack version allocates much less, but this doesn't translate into less GC

Well-Typed

Summary

Take away lessons

- Deep embeddings can be ok for performance, or even excellent
- More predictable performance with interpreters
- Possibly easier to achieve reasonable performance?
- Much smaller object code sizes
- Allocations can be important. We can track them down.

Perils of calling unknown functions

Calling unknown functions

```
data Decoder s s′ where
  . . .
  AlterStack :: StackEvalFn a b → Decoder b s′ → Decoder a s′
```

Interface type between compiled code and interpreter

```
newtype StackEvalFn st st′
  = StackEvalFn
    (∀s.MutableByteArray# s →      -- unboxed
        MutableArray# s Any →      -- boxed
        Int# → Int# →              -- offsets
        State # s →
        (#State# s, Int#, Int ˆ# #))  -- updated offsets
```

Unpack all the Stack components and pass them individually

Avoid having to allocate a Stack constructor

Well-Typed

Calling unknown functions

```
newtype StackEvalFn st st′
    = StackEvalFn
      (∀s.MutableByteArray# s →      -- unboxed
          MutableArray# s Any →      -- boxed
          Int# → Int# →              -- offsets
          State # s →
          (#State# s, Int#, Int ˆ# #))  -- updated offsets
```

Unfortunately this is **terrible**

- slow
- involves hidden allocations

Calling unknown functions

Calling functions that are not statically known is tricky

- has to check for over and under saturation
- think of partial application and functions returning functions

GHC handles certain cases specially

P	1 GC pointer
PP	2 GC pointers
PV	1 GC pointer and a void
PPV	2 GC pointer2 and a void
…	…
N	unboxed word

But our pattern is 'PPNN'. GHC has to chain, PP with N with N.

Slooooo, and allocates.

GHC 7.10 with -O2 does better

Massive cheating

```
newtype StackEvalFn st st′
    = StackEvalFn
      (∀s.MutableByteArray# s →      -- unboxed plus offsets
          MutableArray# s Any →       -- boxed
          State # s →
          (#State# s, Int#, Int ˆ# #))   -- updated offsets
```

Just shove the two offsets in first two slots of the unboxed array.

Now the pattern is PPV which is special cased.

More details

Implementing the stack

Now the interpreter manages a stack, can we do fewer allocations?

Or are we just moving them around?

```
newtype Stack xs = Stack xs

data a :*: b = a :*: b

push :: x → Stack xs → Stack (x :*: xs)
push x (Stack xs) = Stack (x :*: xs)

pop :: Stack (x :*: xs) → (x, Stack xs)
pop (Stack (x :*: xs)) = (x, Stack xs)
```

Implementing the stack

Now the interpreter manages a stack, can we do fewer allocations?

Or are we just moving them around?

newtype Stack xs = Stack xs

data a :∗: b = a :∗: b

push :: x → Stack xs → Stack (x :∗: xs)
push x (Stack xs) = Stack (x :∗: xs)

pop :: Stack (x :∗: xs) → (x, Stack xs)
pop (Stack (x :∗: xs)) = (x, Stack xs)

Cheat!

Same typed interface (though in ST), but unsafe low level impl

Massive cheating

```
data Stack s xs = Stack
        (MutableArray# s Any) ! Int
pushBoxed :: x → Stack s xs → ST s (Stack s (x :∗: xs))
popBoxed  :: Stack s (x :∗: xs) → ST s (x, Stack s xs)
```

Just read/write the array, and adjust the stack pointer

Massive cheating

```
data Stack s xs = Stack
        (MutableArray# s Any) ! Int
pushBoxed :: x → Stack s xs → ST s (Stack s (x :∗: xs))
popBoxed  :: Stack s (x :∗: xs) → ST s (x, Stack s xs)
```

Just read/write the array, and adjust the stack pointer

Have to `unsafeCoerce#` between `Any` and `x` element types

```
pushBoxed x (Stack st# stptr#) =
  ST (λs →
    case writeArray# st# stptr# (unsafeCoerce# x) s of
      ...
```

Massive cheating

We can treat primitive types specially
keep them unboxed, save allocation

```
data Stack s xs = Stack
        (MutableByteArray# s) ! Int    -- for unboxed elements
        (MutableArray# s Any) ! Int    -- for boxed elements

data a :∗: b
data a :#: b    -- for keeping track of unboxed types

pushBoxed :: x       → Stack s xs → ST s (Stack s (x      :∗: xs))
pushWord  :: Word → Stack s xs → ST s (Stack s (Word :#: xs))
```

`Stack s (Word :#: s)` is not at all the same as

`Stack s (Word :∗: s)`

Well-typed interpreter

No type cheating in the interpreter

```
go_slow  :: Decoder s s′ → Stack r s → BS.ByteString
            → ST r (Either Err (Stack r s′))
go_slow d st bs = do
  res ← go_fast d st bs
  case res of
    FastDone _bs′ st′        → return (Right st′)
    SlowDecoder bs′ st′ d′ → . . .
            -- fix up and go back to fast path

go_fast  :: Decoder s s′ → Stack r s → BS.ByteString
           → ST r (SlowPath r s′)
```

Fast path / slow path split

- input chunk boundary issues
- handle stack growth

More cunning trickery

Done / Sequence control flow can be handled with recursion in fast path but this is relatively expensive

- many args to save to the stack
- tricky to eliminate allocation of fast-path's result

Use an explicit (typed) control stack!

More cunning trickery

Done / Sequence control flow can be handled with recursion in fast path but this is relatively expensive

- many args to save to the stack
- tricky to eliminate allocation of fast-path's result

Use an explicit (typed) control stack!

$$\begin{aligned} \text{go\_fast} :: {} & \text{Decoder s s}' \to \text{Stack r s} \\ & \to \text{BS.ByteString} \to \text{ST r (SlowPath r s}') \end{aligned}$$

becomes

$$\begin{aligned} \text{go\_fast} :: {} & \text{Decoder s s}' \to \text{Stack r s} \to \text{CStack r s}' \text{ s}'' \\ & \to \text{BS.ByteString} \to \text{ST r (SlowPath r s}'') \end{aligned}$$

More cunning trickery

Now the fast path function is only tail recursive

```
go_fast (Sequence d1 d2) ! st ! cst ! bs
    | CStack.spaceAvail cst
    = do cst′ ← CStack.push d2 cst
         go_fast d1 st cst′ bs
go_fast Done ! st ! cst ! bs = do
   view ← CStack.tryPop cst
   case view of
     CStack.Empty             → return (FastDone bs st)
     CStack.NonEmpty d′ cst′  → go_fast d′ st cst′ bs
```

We can implement the control stack efficiently (no allocation) with more cheating

More cunning trickery

Safe control stack implementation with GADTs

```
data CStack r a b where
  Empty ::                                     CStack r c c
  Push  :: Decoder b a → CStack r a c → CStack r b c

newStack :: ST r (CStack r e e)
newStack = return Empty

push :: Decoder b a → CStack r a c → ST r (CStack r b c)
push d cst = return (Push d cst)

tryPop :: CStack r a b → ST r (CStackView r a b)
tryPop Empty         = return Empty
tryPop (Push d cst′) = return (NonEmpty d cst′)
```

More cunning trickery

Fast control stack implementation with mutable arrays

```
data CStack s a b = CStack (MutableArray# s Any) Int#

data CStackView r a b where
  Empty    ::                                   CStackView r c c
  NonEmpty :: Decoder b a → CStack r a c → CStackView r b c

newStack :: ST r (CStack r e e)
push     :: Decoder b a → CStack r a c → ST r (CStack r b c)
tryPop   :: CStack r a b → ST r (CStackView s a b)
```

Essentially the same interface, type of elements tracked safely.

But must handle stack growth

```
spaceAvail :: CStack s a b → Bool
growSize   :: CStack s a b → ST s (CStack s a b)
```