Efficient Communication and Collection with Compact Normal Forms

Edward Z. Yang, Giovanni Campagna,
Ömer Agacan, Ahmed Al-Hassany, Abhishek Kulkarni, and Ryan Newton
Where the story begins...
Here is a problem.
End of Moore's law, blah blah blah
Haskell

Type safety!
EDSLs!
Memory safety!

(MPI) (or any other GC'd language)
serialization & deserialization
Arenas

Regions

C++
Haskell

Serialized Format
So, it would be really great if there was some **compact** form for Haskell data types
So, it would be really great if there was some compact form for Haskell data types.

Me: You'd have to rewrite GHC.
Parallel DSLs often compute on large data structures in normal form. A compact in-memory representation ... would be beneficial for cache performance and might reduce GC and serialization overheads.
Parallel DSLs often compute on large data structures in normal form. A compact in-memory representation ... would be beneficial for cache performance and might reduce GC and serialization overheads.

Me: Hmmm.
Constraint #1:

We want a **compact** representation of in-memory data...
Constraint #1: We want a **compact** representation of in-memory data...

Constraint #2: ...but we want to **reuse** our code for manipulating pointer data structures.
Constraint #1: We want the memory representation be contiguous...

Constraint #2: ...but we want to reuse our code for manipulating pointer data structures.
Ok, we can do this.
Compact Normal Forms

Summary

1. In-memory representation = network representation

2. Divide heap into region per data structure; copy data into contiguous segments

3. Enforce data in region has no outbound pointers and is in normal form (immutability)
The use-case
The use-case
The use-case
The use-case

Amortized copies
Old tricks for a new dog

Partition the heap

one region = one transmittable structure

General Purpose Heap

4KB
data Compact a

new :: IO (Compact ())

append :: Compactable a \Rightarrow
          Compact b \rightarrow a \rightarrow IO (Compact a)

get :: Compact a \rightarrow a

(IO to make it easier to control sharing)
c ← new

c :: Compact () → region
\( \chi \) :: Compact ()
append c x

{thunk}

c :: Compact ()

region
append c x

c:: Compact ()

region
$r \leftarrow \text{append } c \ x$

$r ::= \text{Compact Tree}$
r :: Compact Tree
append

r :: Compact Tree

region
\[ r' \leftarrow \text{append } r \ y \]

\[ r':: \text{Compact Tree} \]
Invariants for a network format

- No outbound pointers
  A pointer in a region points within the region.

- All objects are in normal form
Compaction

given an object

copy to destination heap

for each pointer field:

recursively process the object

append :: Compactable a ⇒ Compact b → a → IO (Compact a)

restriction: no mutable data

evaluate object to normal-form first, then recursive copy ensures internal pointers
What about GC?
Evacuate the roots

roots

from space

to space
Process the to-do list breadth first
Process the to-do list breadth first
Process the to-do list breadth first
Process the to-do list breadth first
from space

roots

No longer contiguous!
So don’t garbage collect it (Does waste space)

root set

don’t want to trace

no outbound pointers means no live data!
OK, but how fast is it?
Serialization benchmark (binary tree)

Factor slow down relative to CNF

Number of leaves

Java
Cereal
Binary
Compact/Share

gc savings!
Serialization benchmark (binary tree)

Factor slow down relative to CNF

Number of leaves

Java
Cereal
Binary
Compact/Share

GC effects
gc savings!
Size blowup!

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Value Size</th>
<th>MBytes</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact</td>
<td>bintree</td>
<td>$2^{23}$ leaves</td>
<td>320</td>
<td>1.00</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
<td>80</td>
<td>0.25</td>
</tr>
<tr>
<td>Cereal</td>
<td></td>
<td></td>
<td>80</td>
<td>0.25</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td>160</td>
<td>0.50</td>
</tr>
<tr>
<td>Compact</td>
<td>pointtree</td>
<td>$2^{23}$ leaves</td>
<td>512.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
<td>272</td>
<td>0.53</td>
</tr>
<tr>
<td>Cereal</td>
<td></td>
<td></td>
<td>272</td>
<td>0.53</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td>400</td>
<td>0.78</td>
</tr>
<tr>
<td>Compact</td>
<td>twitter</td>
<td>1024MB</td>
<td>3527.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
<td>897.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Cereal</td>
<td></td>
<td></td>
<td>897.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td>978.15</td>
<td>0.28</td>
</tr>
</tbody>
</table>

1Gbit: 240MB = 2s extra
10Gbit: 240MB = 0.2s extra

(NB: serializing took 7s!)
RDMA

The graph shows the mediantime in seconds (log) on the y-axis and the number of leaves (in powers of 2) on the x-axis. The graph compares different methods:

- Zero-copy/Binary
- Zero-copy/Compact
- Binary
- Compact
Block structured heap
+ Immutable data structures
+ Minor GC modifications

= Compact Normal Forms

ezyang.com/compact.html

Thank you!
Why is it in the IO monad?

- Doesn’t have to be: if you trust your optimizer to preserve sharing.
- Monad for sequencing and sharing
- API is referentially transparent