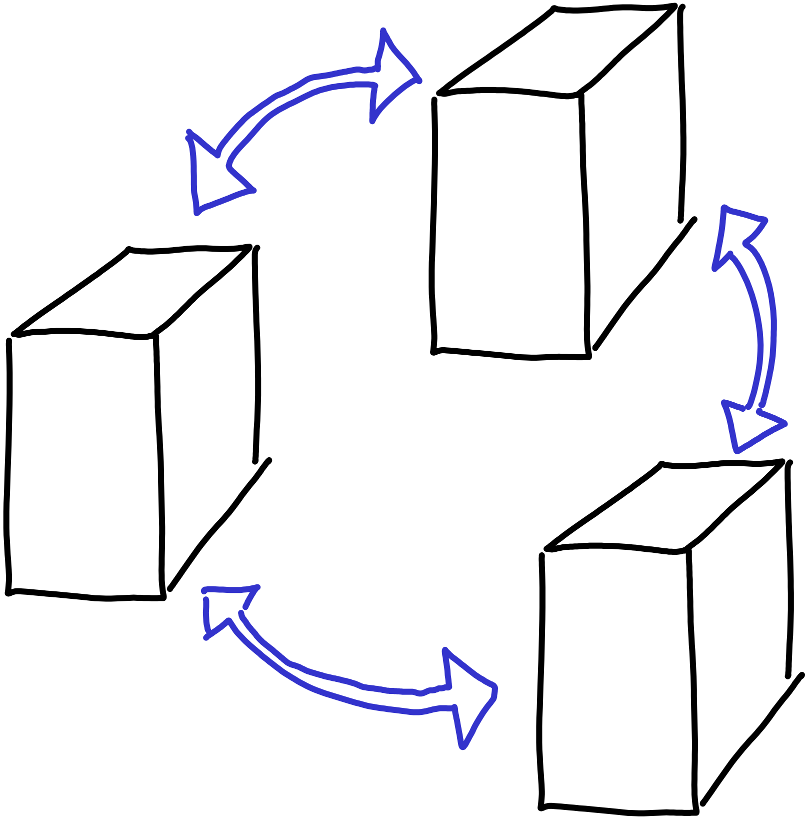
Efficient Communication and Collection with Compact Normal Forms

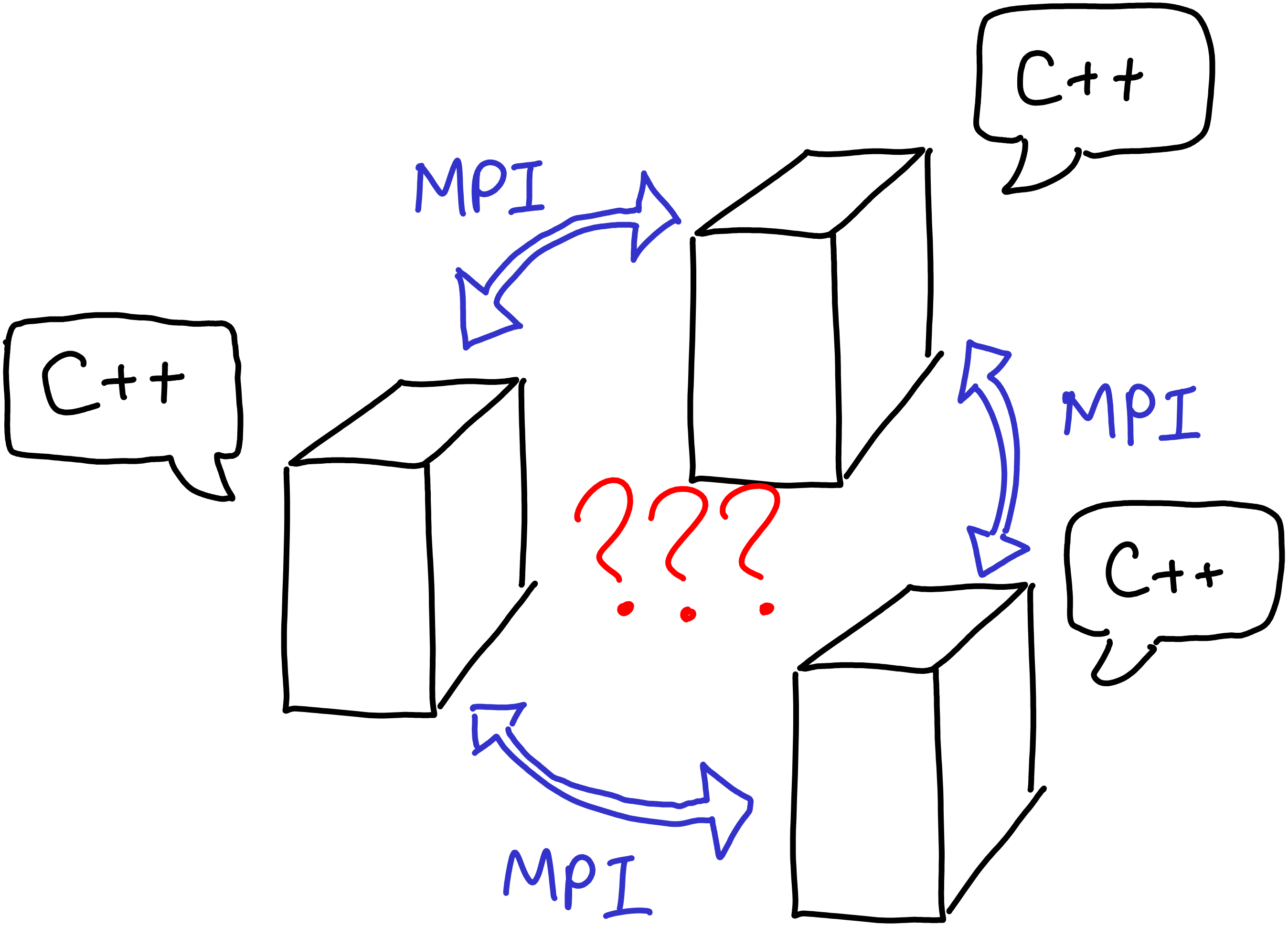
Edward Z. Yang, Giovanni Campagna, Ömer Agacan, Ahmed Al-Hassany, Abhishek Kulkarni, and Ryan Newton

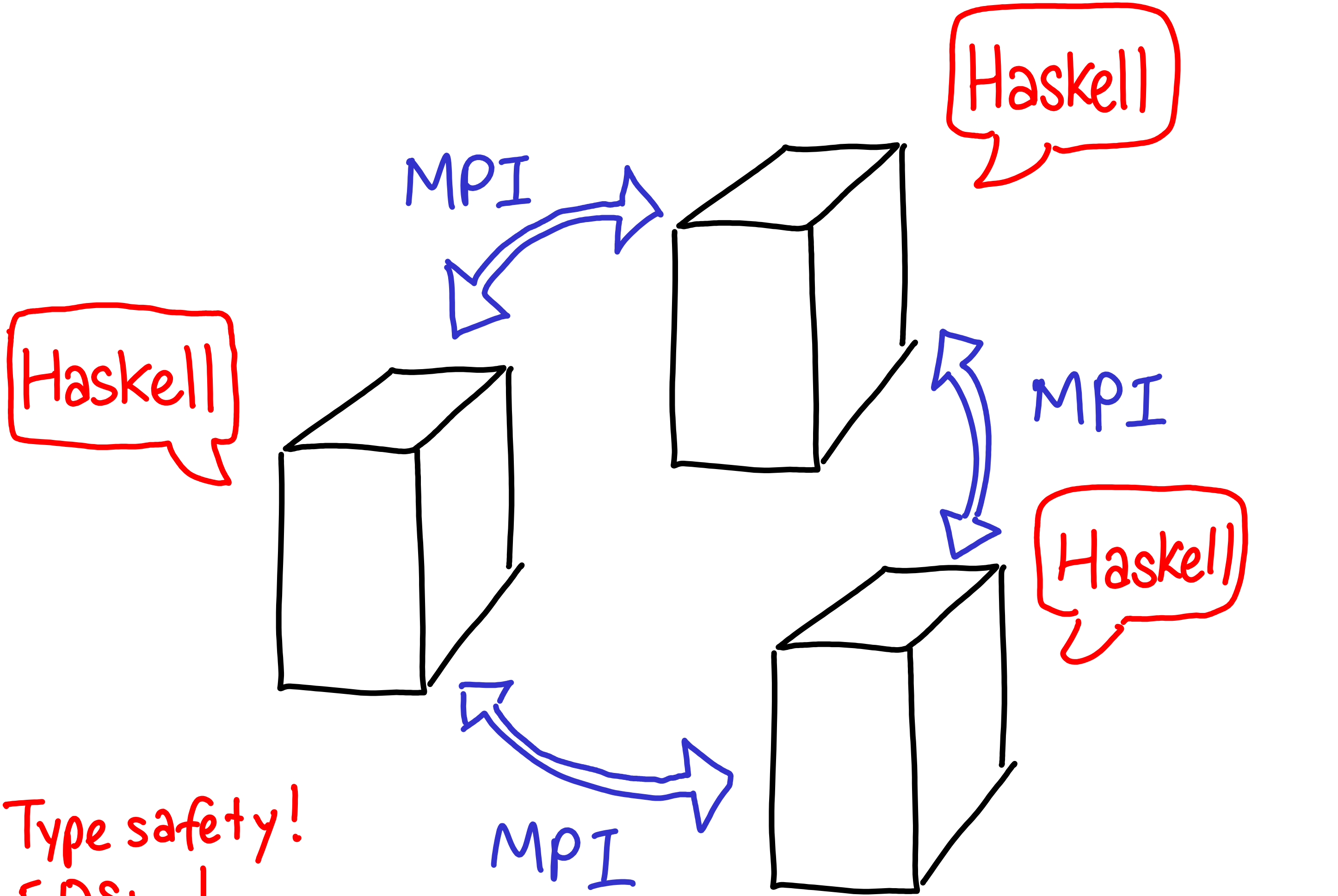
Where the story begins...

Here is a problem.



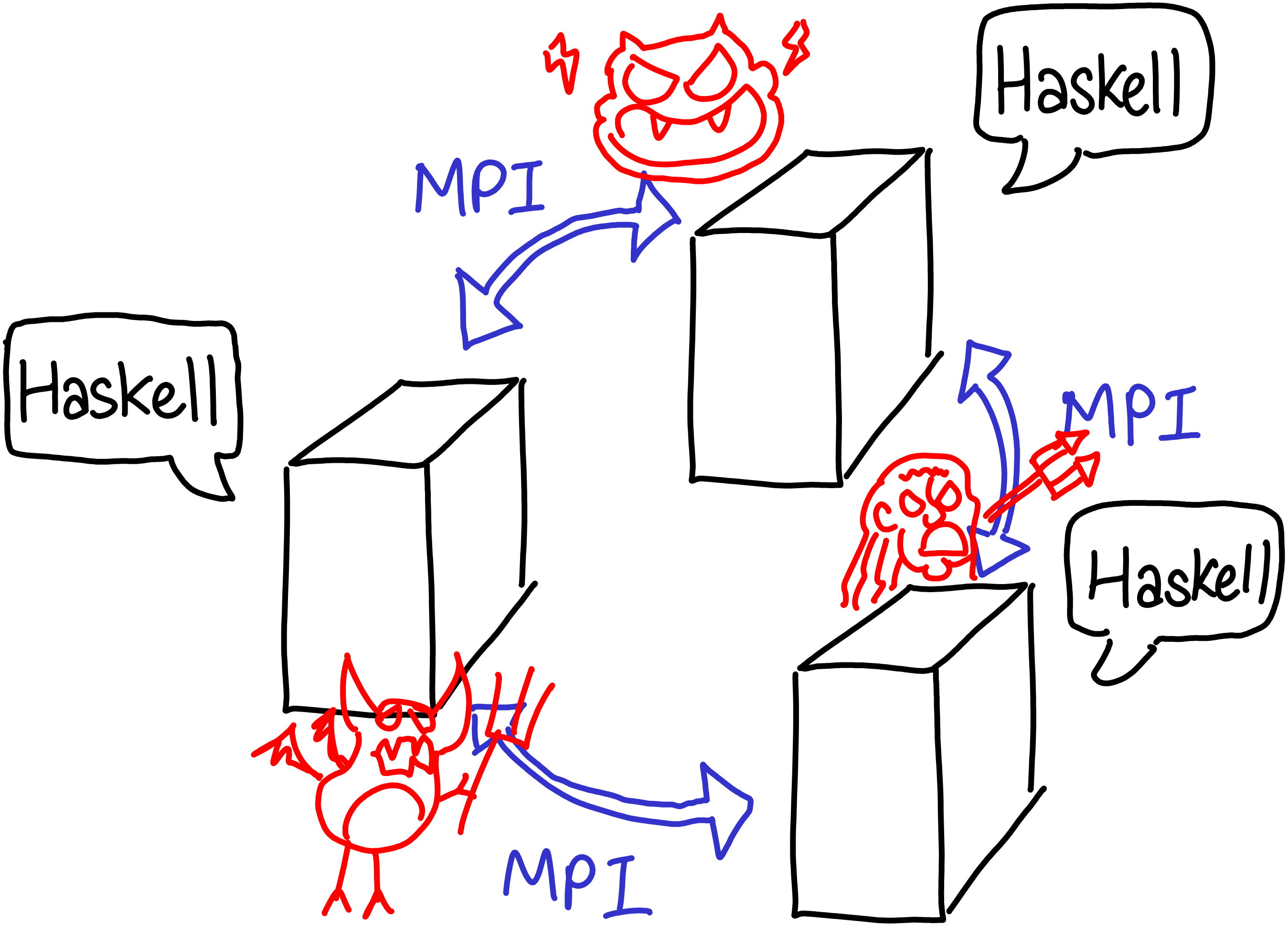
End of Moore's law, blah blah blah


C++
MPI
C++
???
MPI
C++
MPI

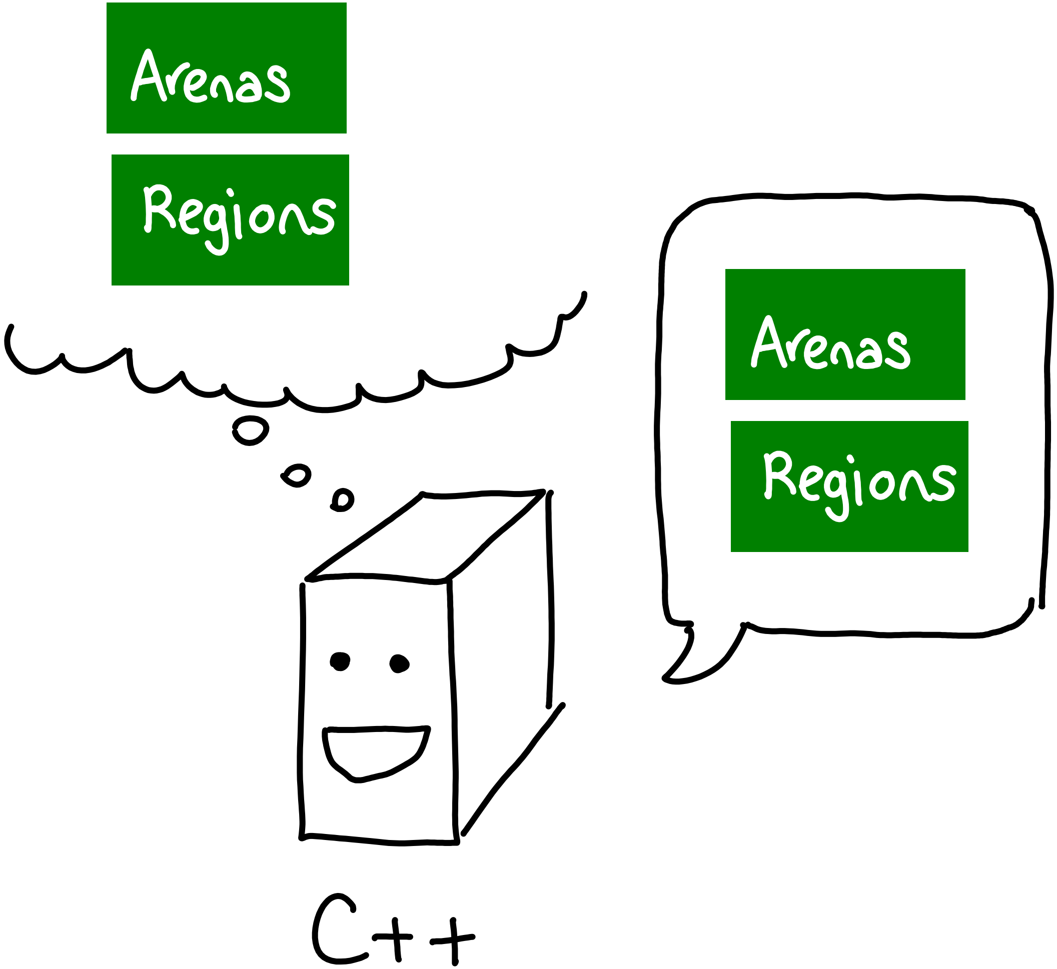


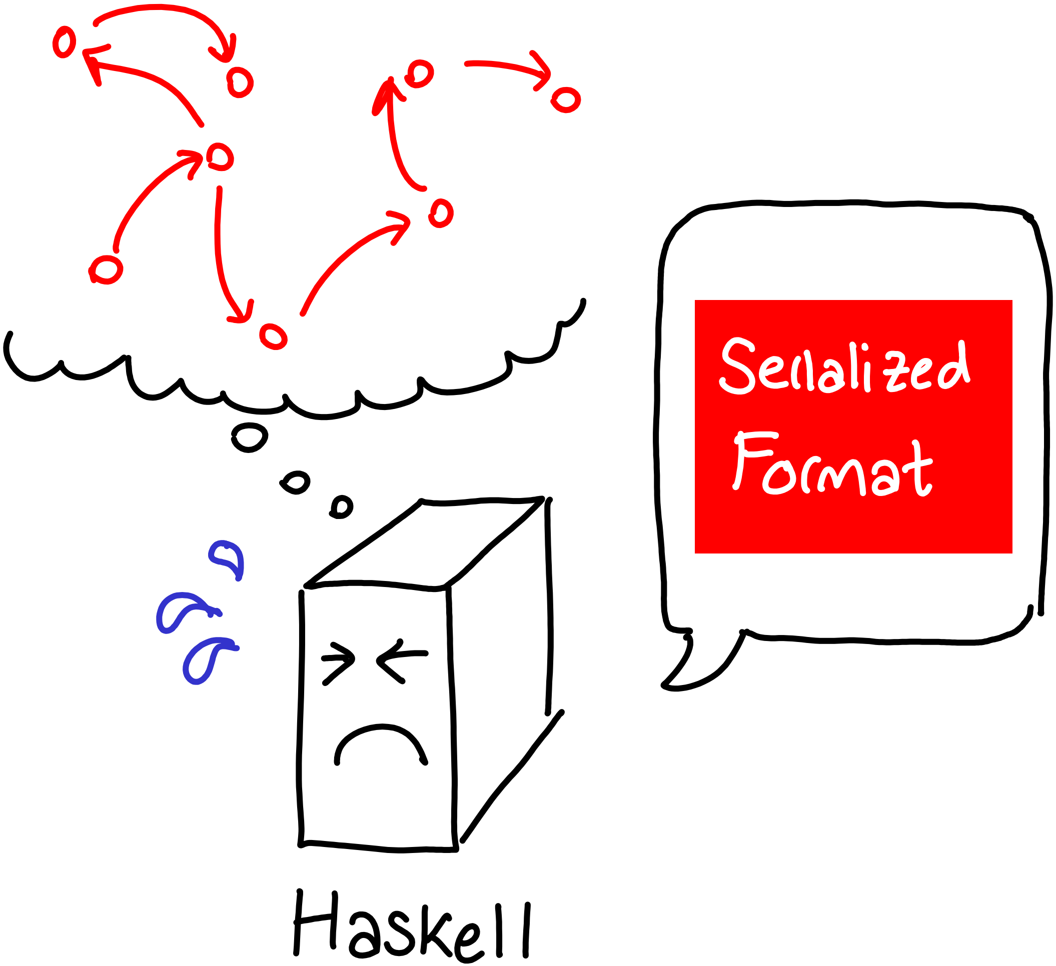
Type safety!
EDSLs!
Memory safety!

(or any other GC'd language)



serialization & deserialization


Arenas
Regions
Arenas
Regions
C++


Serialized Format
Haskell

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.

Later that summer...

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.

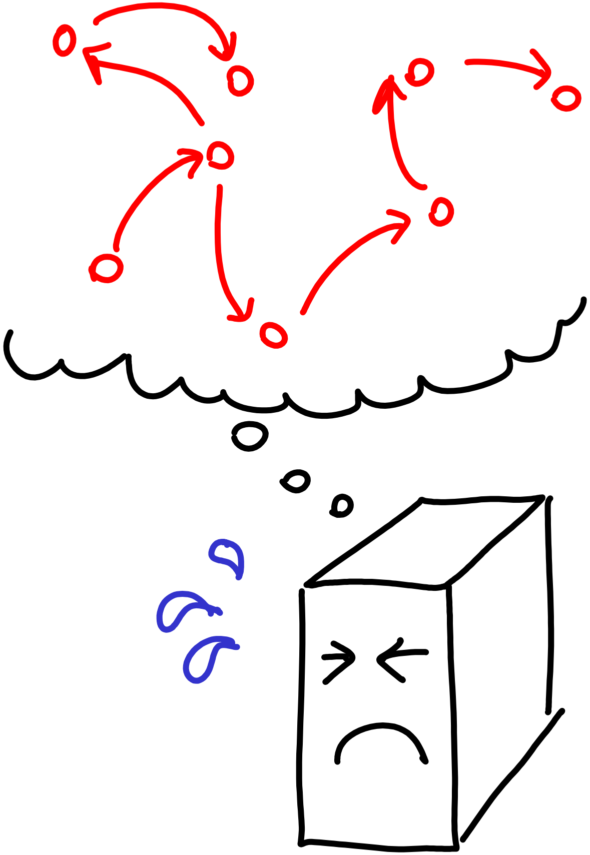
Compact Normal Forms Summary

1. In-memory representation = network representation

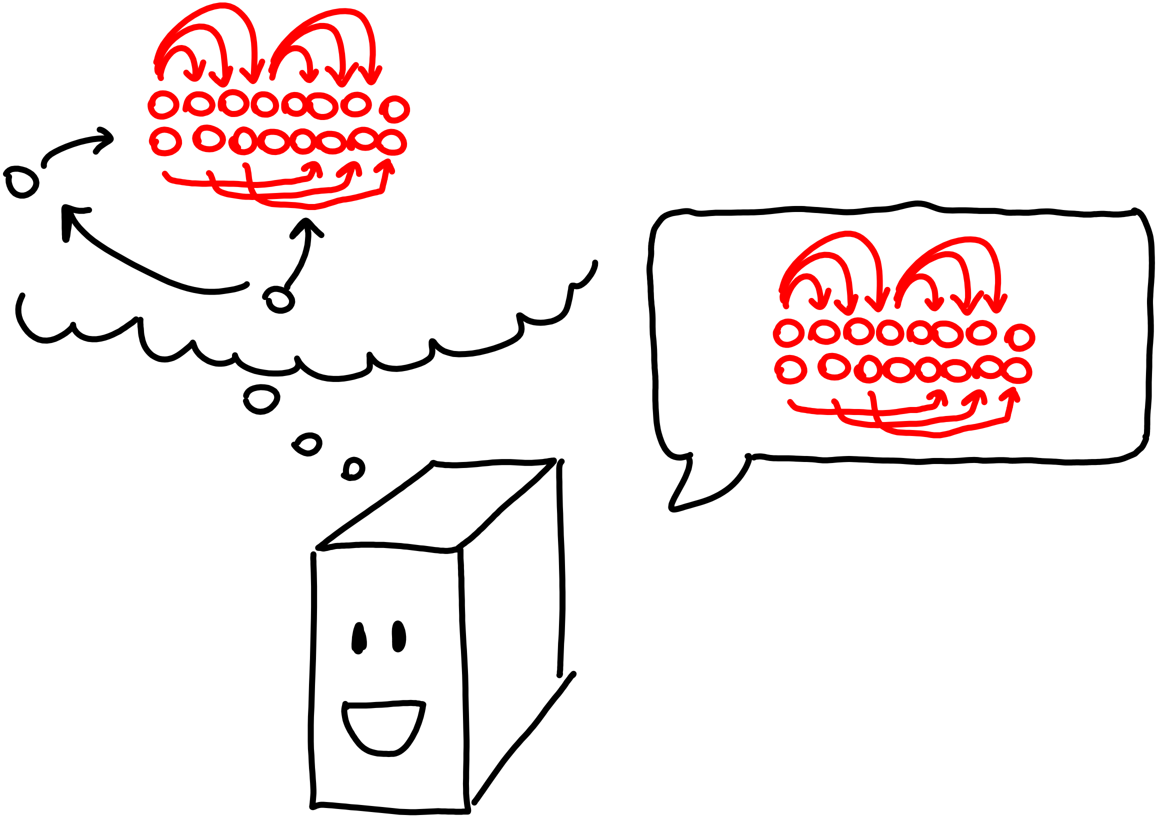
2. Divide heap into region per data structure; copy data into contiguous segments

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

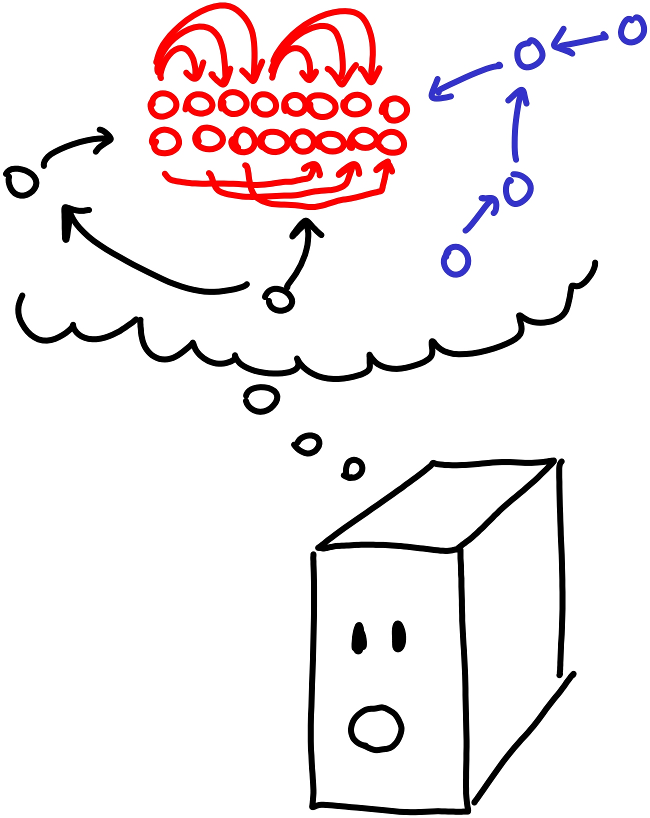
The use-case



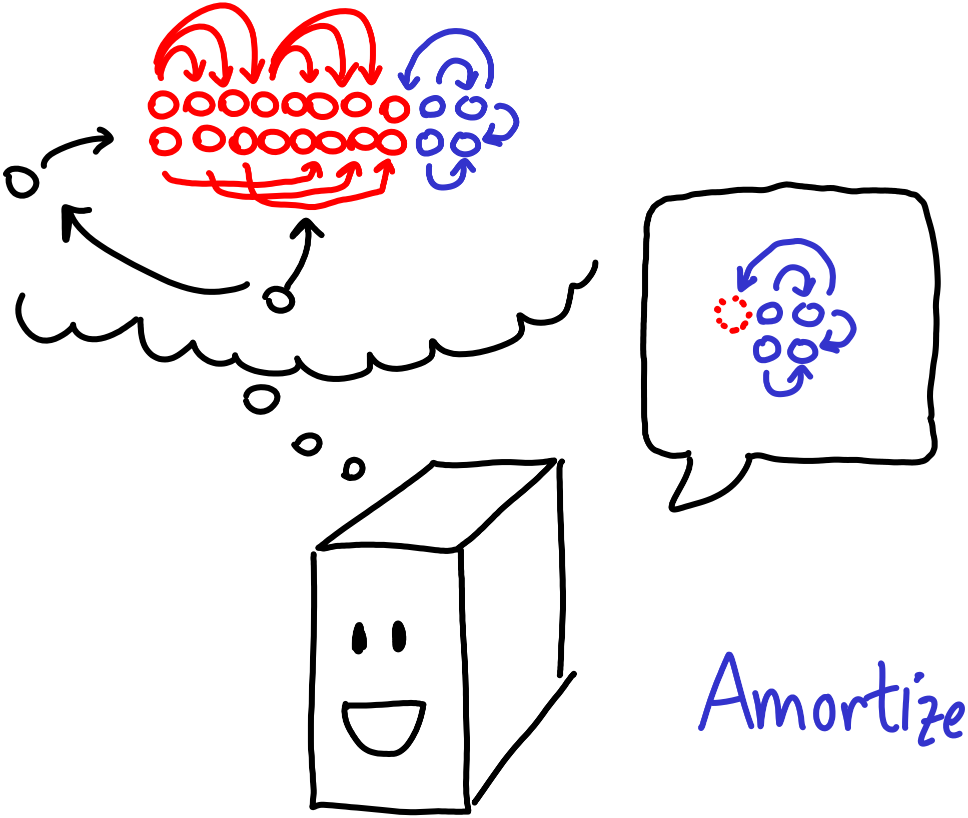
The use-case



The use-case



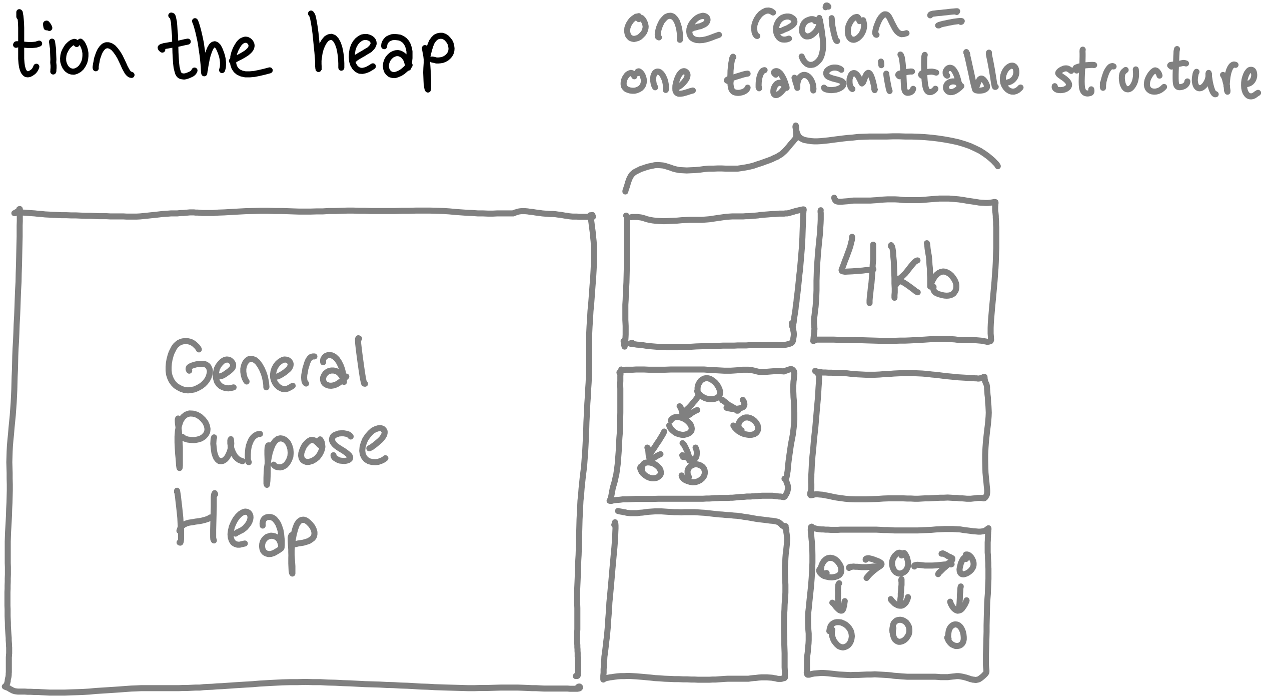
The use-case



Amortized copies

Old tricks for a new dog

Partition the heap

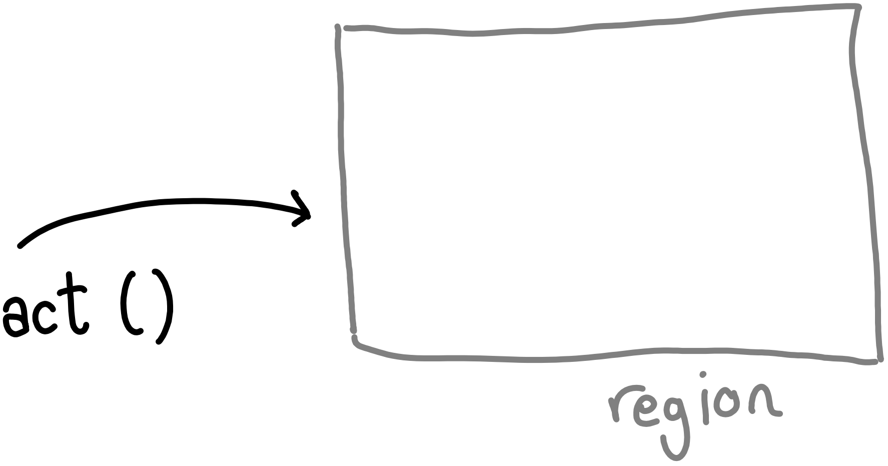


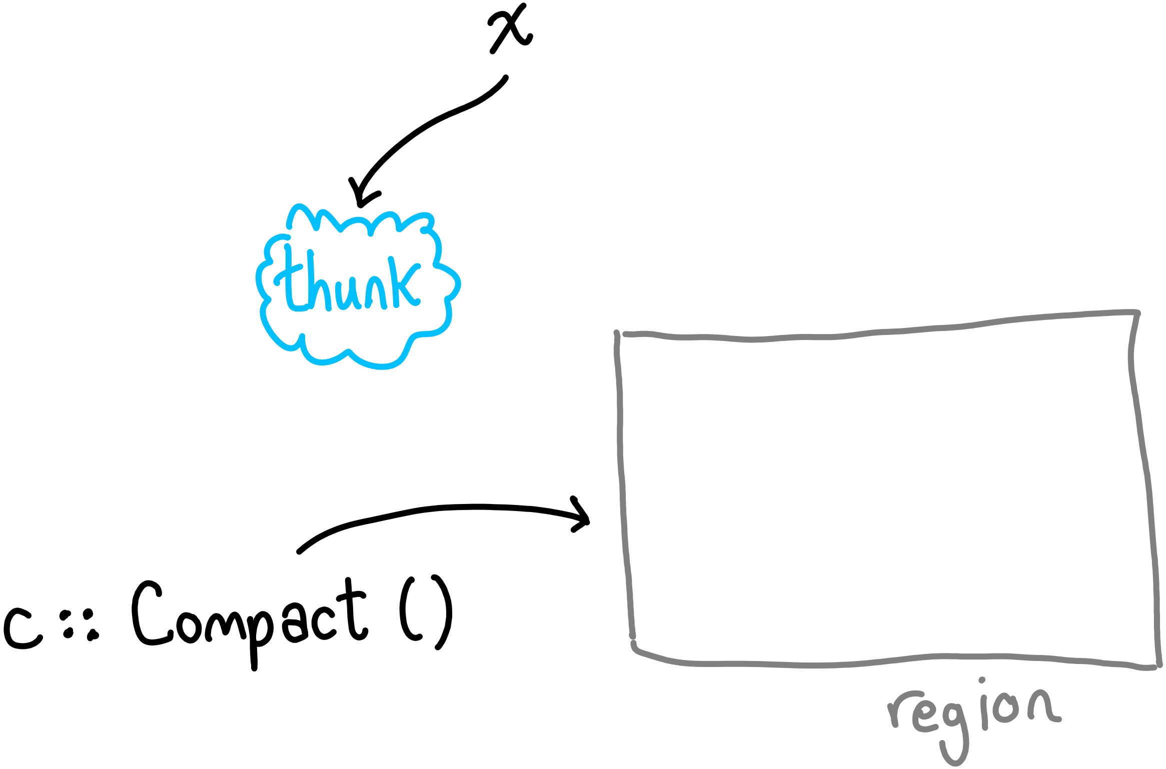
not essential;
could fail at runtime

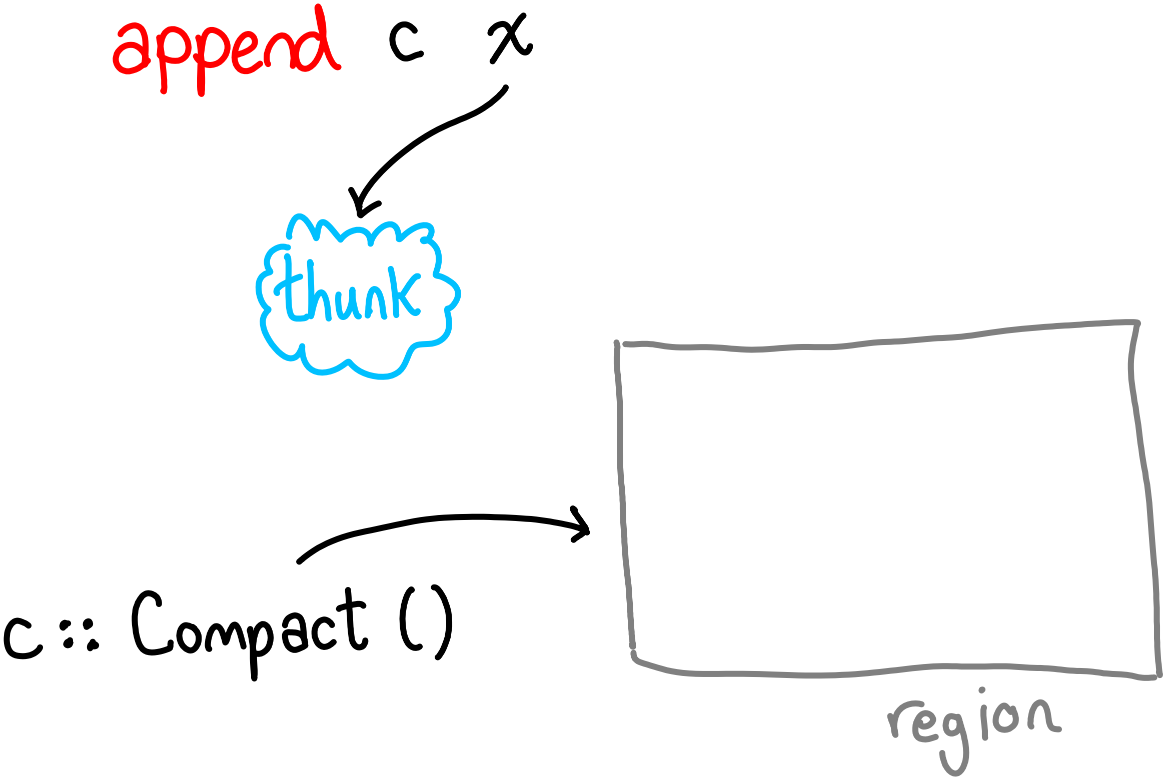
```
data Compact a

new    :: IO (Compact ())
append :: Compactable a =>
    Compact b -> a -> IO (Compact a)

get    :: Compact a -> a
```

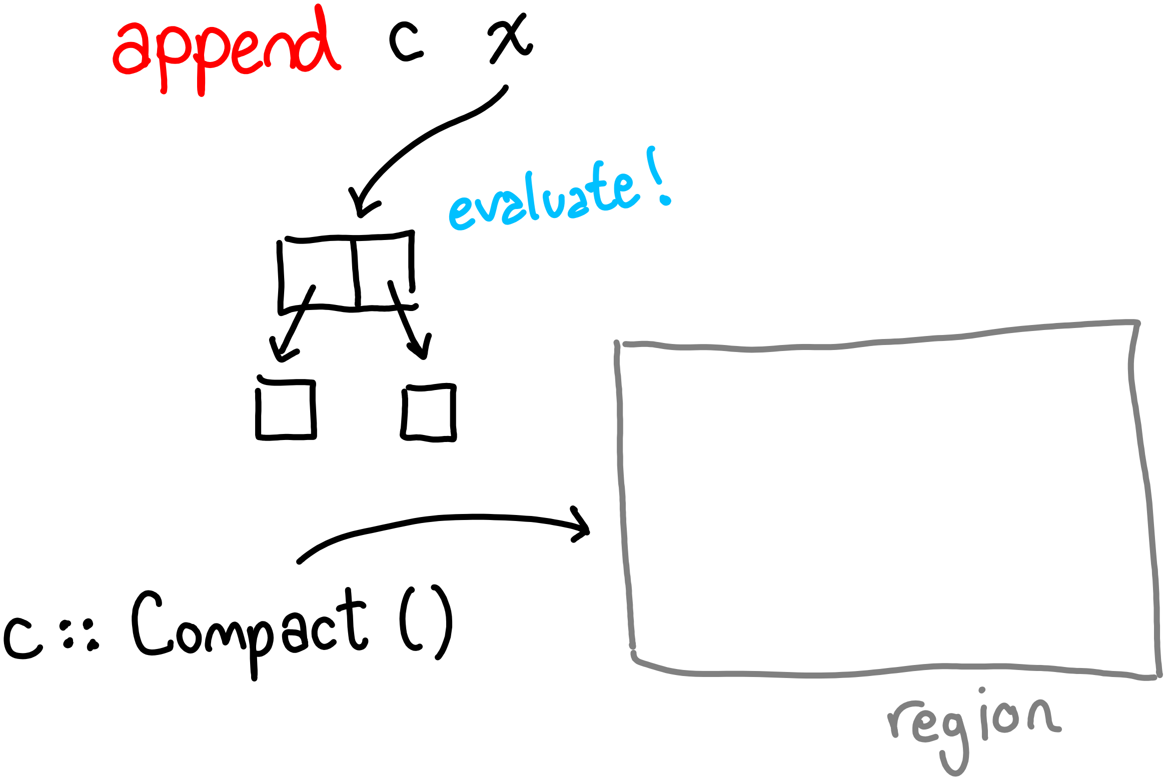
(IO to make it easier to control sharing) c ← new

c :: Compact ()

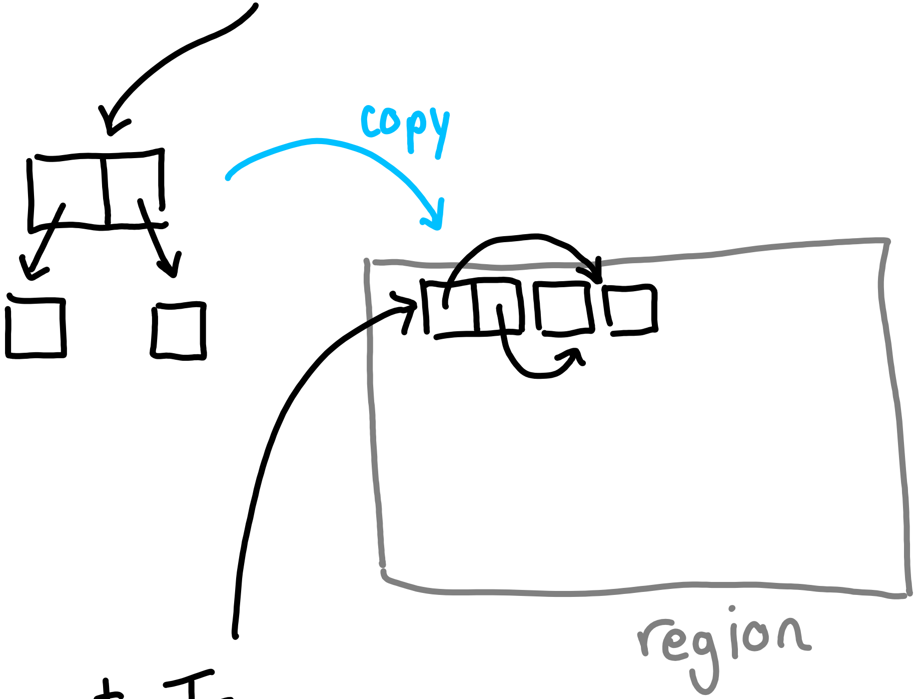



x
thunk
c :: Compact ()
region

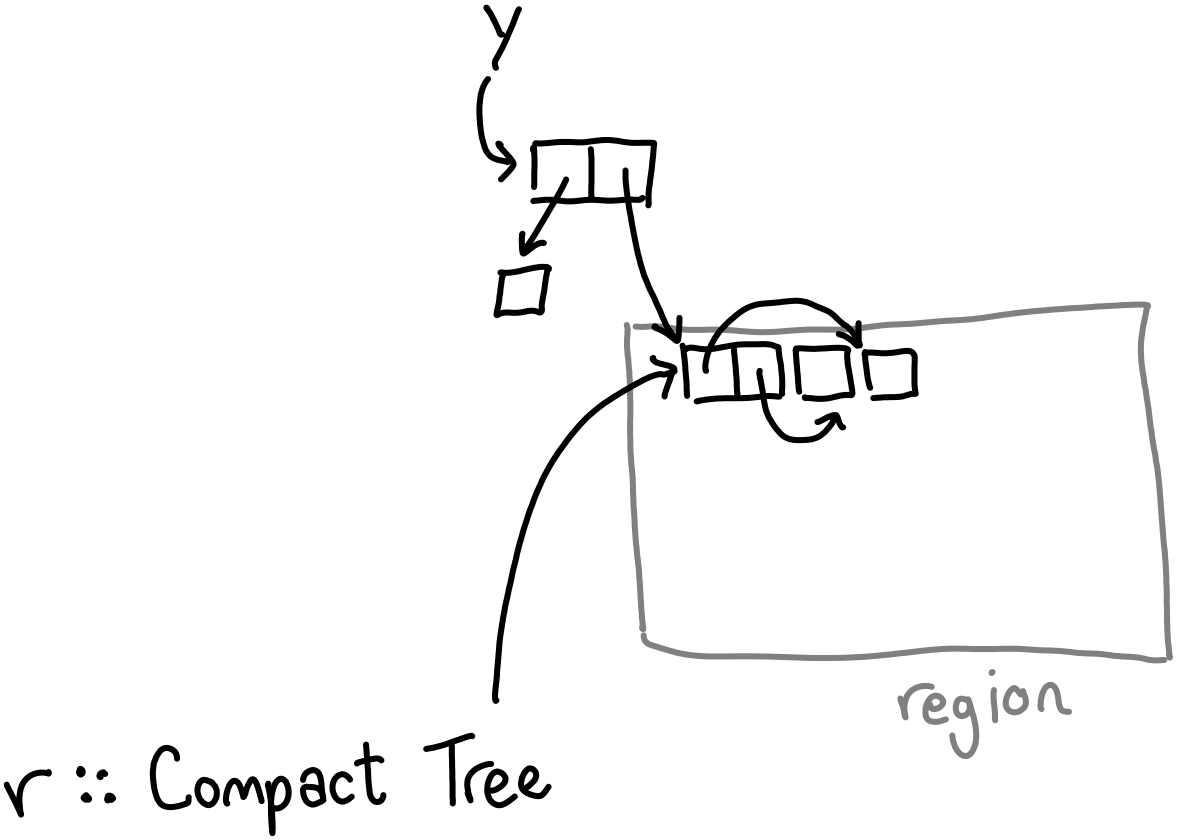

append c x
thunk
c :: Compact ()
region

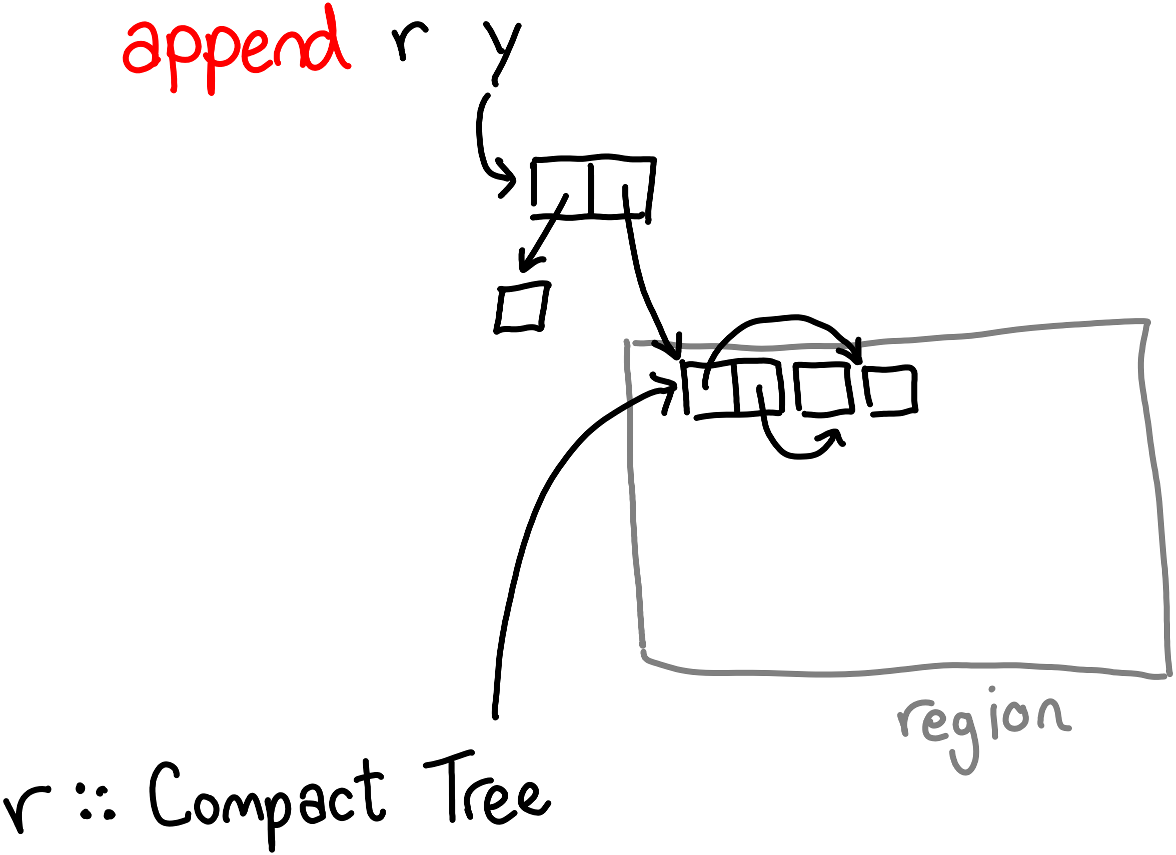

append c x
evaluate!
c :: Compact ()
region

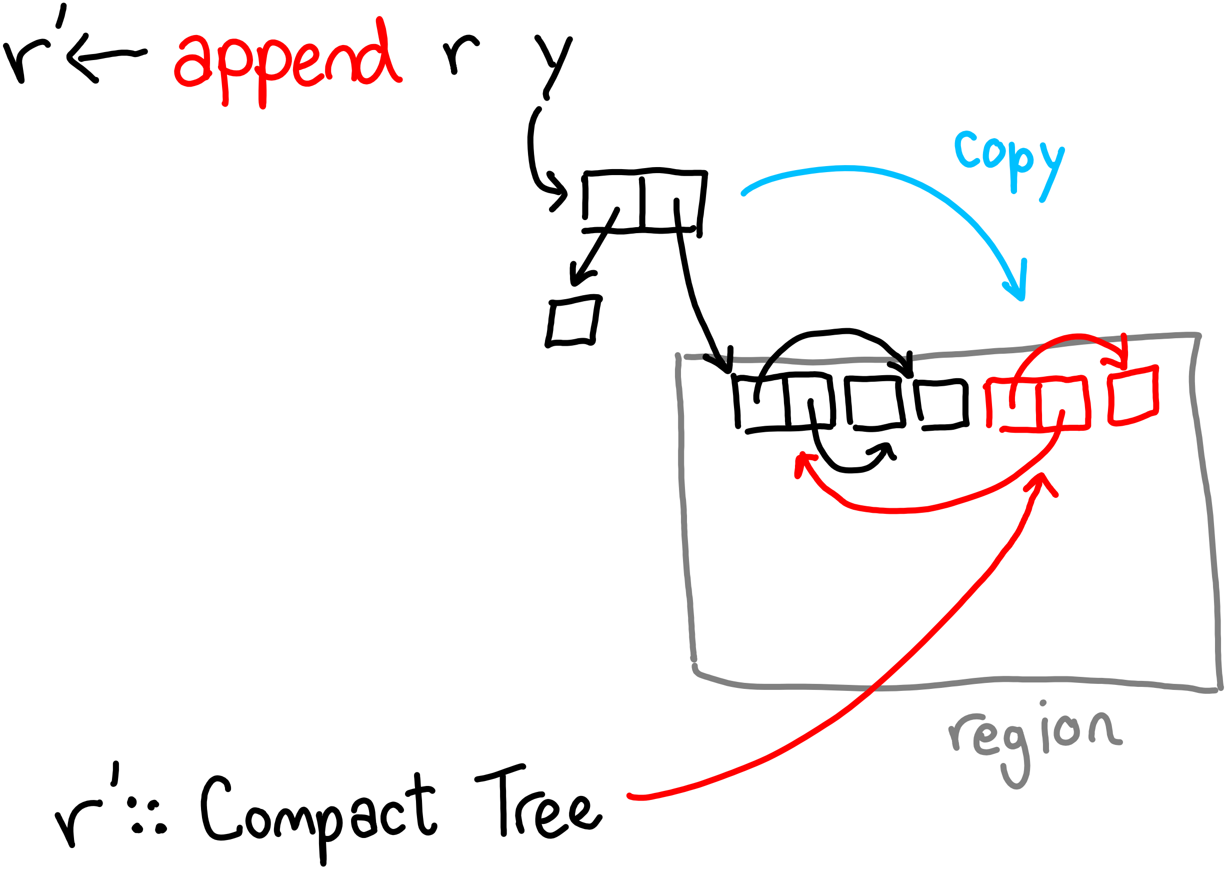
```
r ← append c x
```



```
r :: Compact Tree
```


y
region
r :: Compact Tree


append r y
region
r :: Compact Tree


r' ← append r y
copy
region
r' :: 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

```
append :: Compactable a =>
    Compact b -> a -> IO (Compact a)
```

restriction:
no mutable data

given an object

copy to destination heap

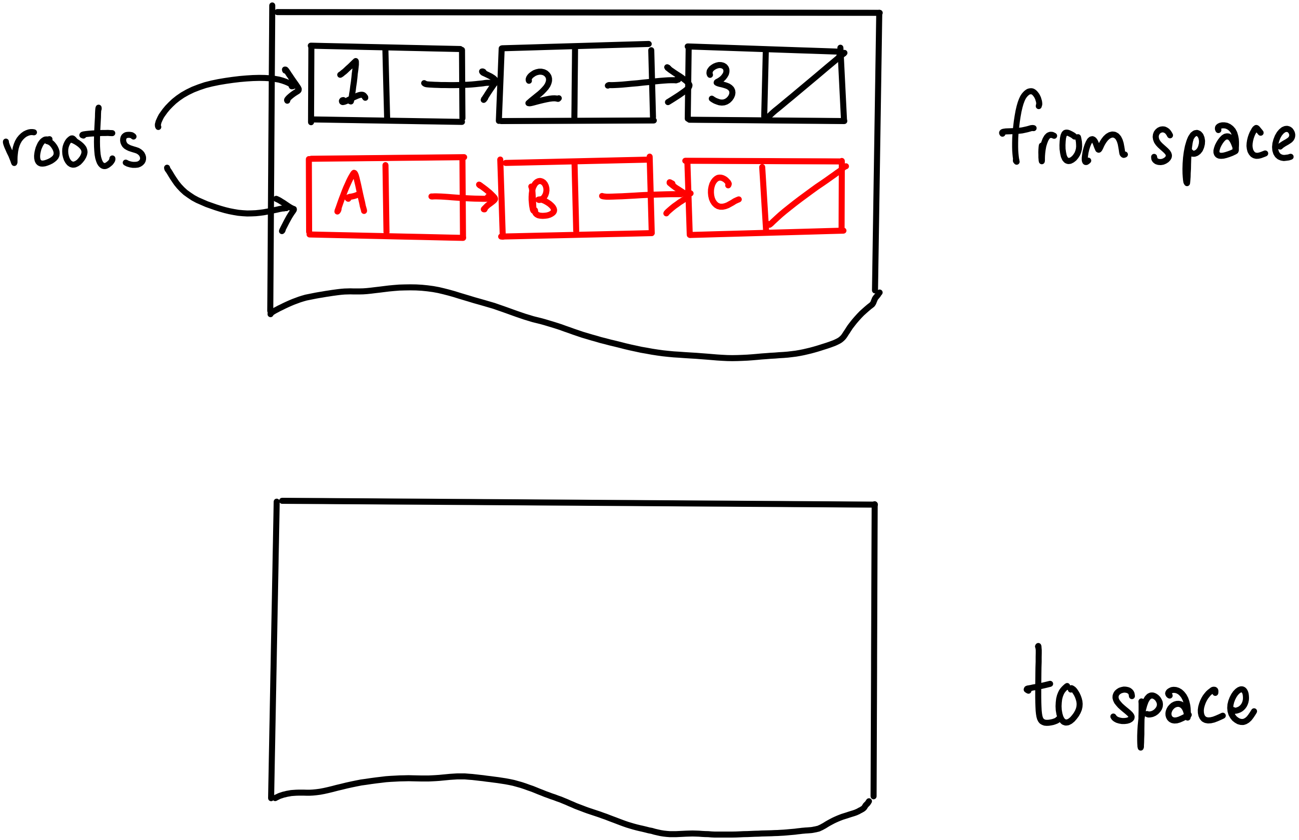
for each pointer field:

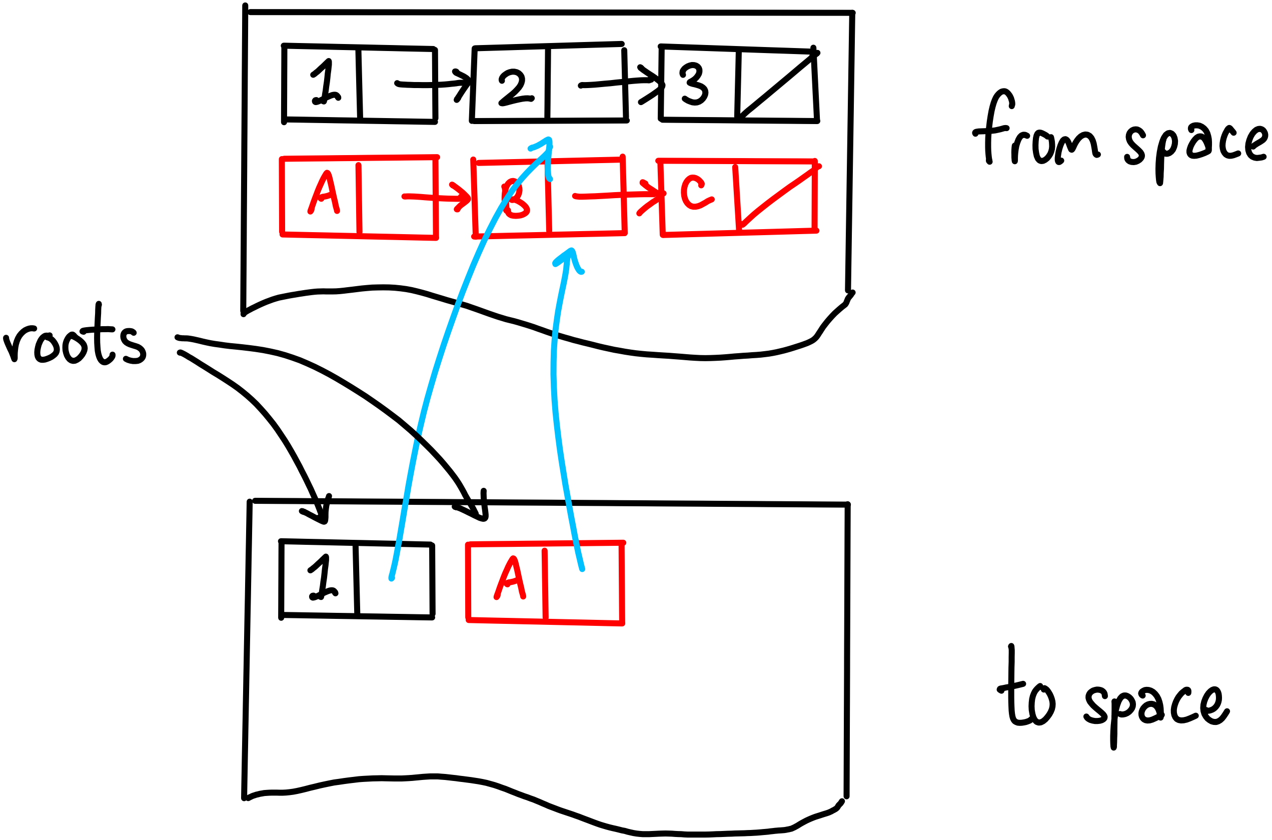
recursively process the object

evaluate object to normal-form first,
then recursive copy ensures internal pointers

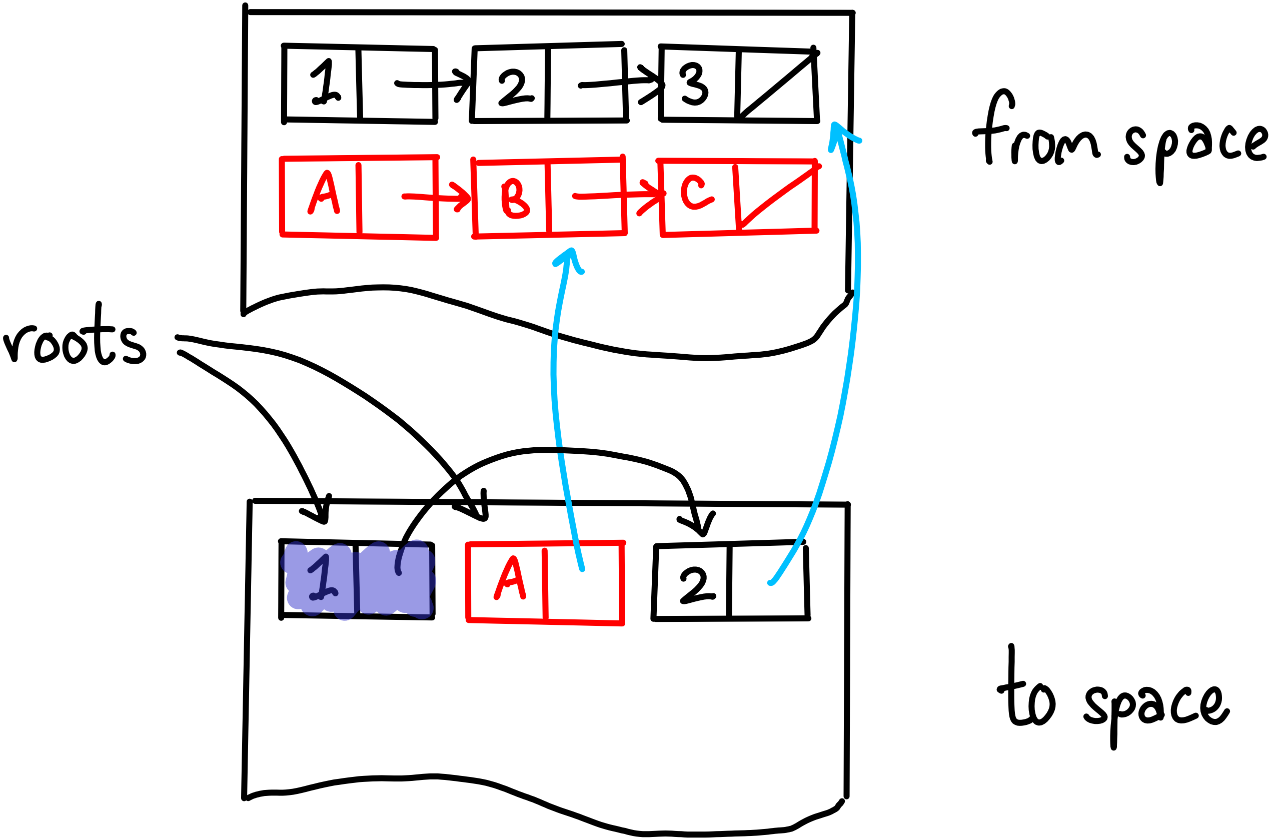
What about GC?

Cheney Garbage Collector

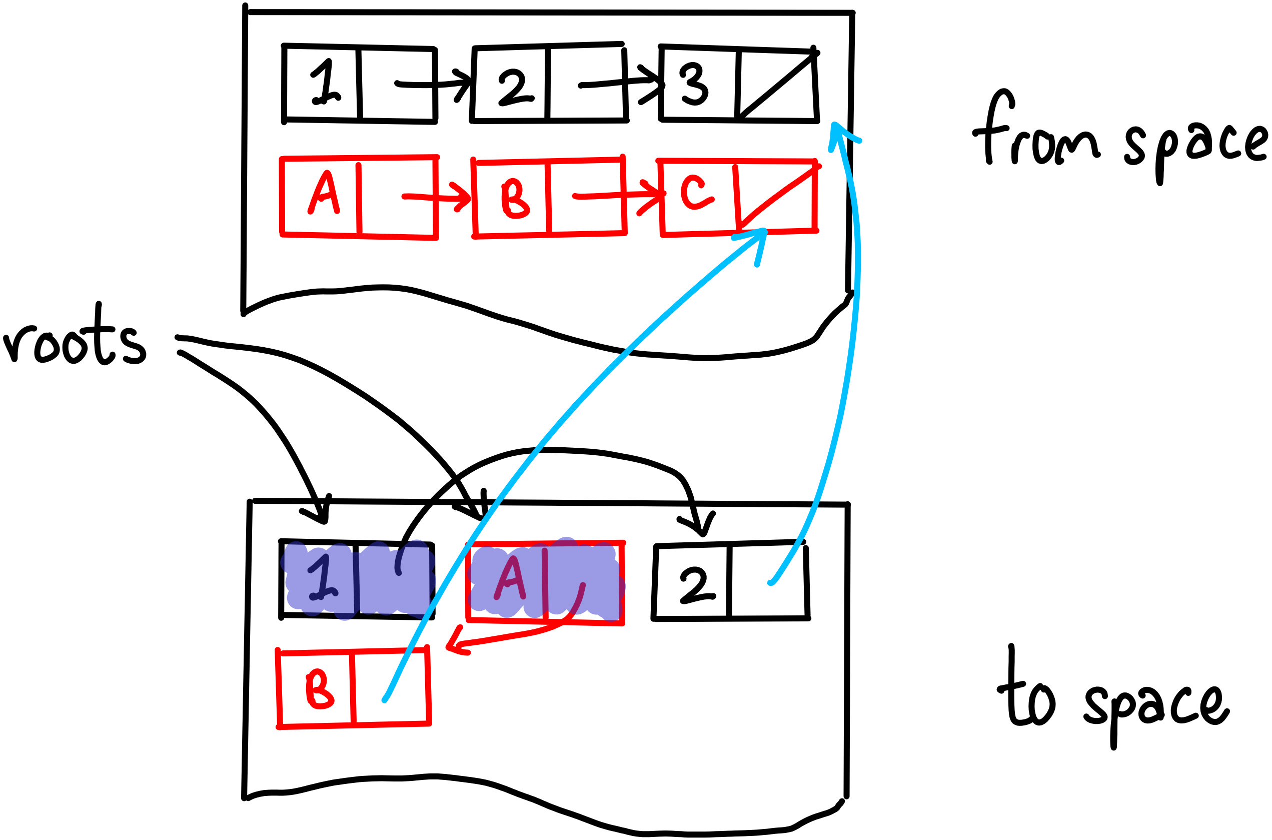

roots
1
2
3
A
B
C
from space
to space



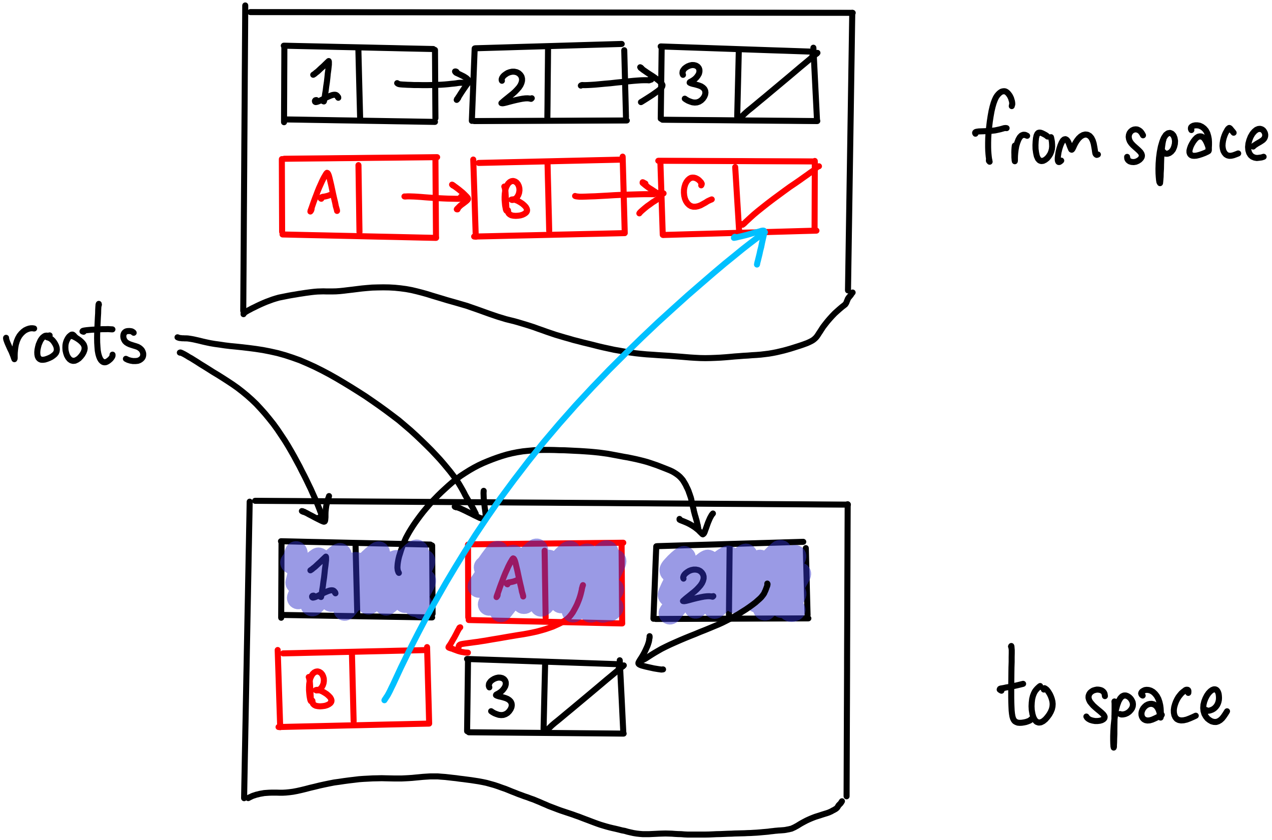
Evacuate the roots



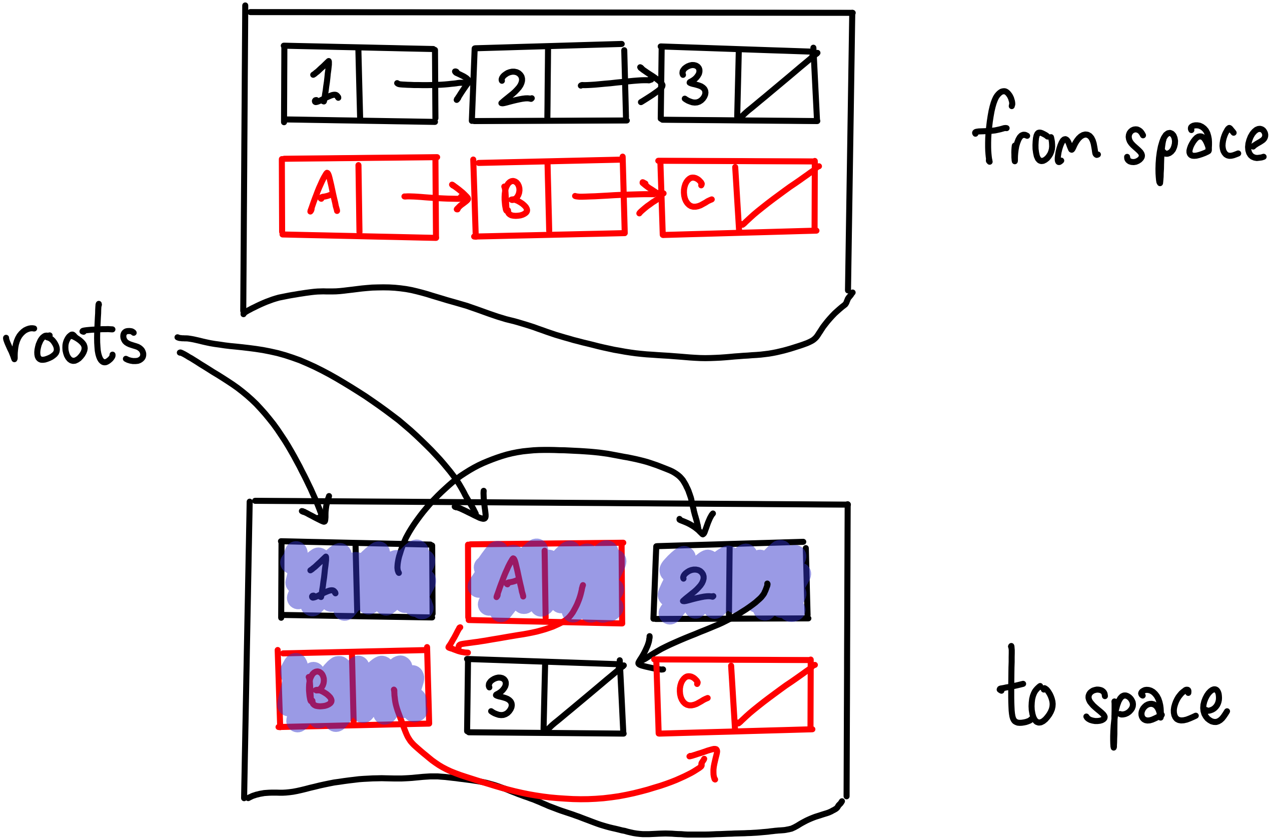
Process the to-do list breadth first



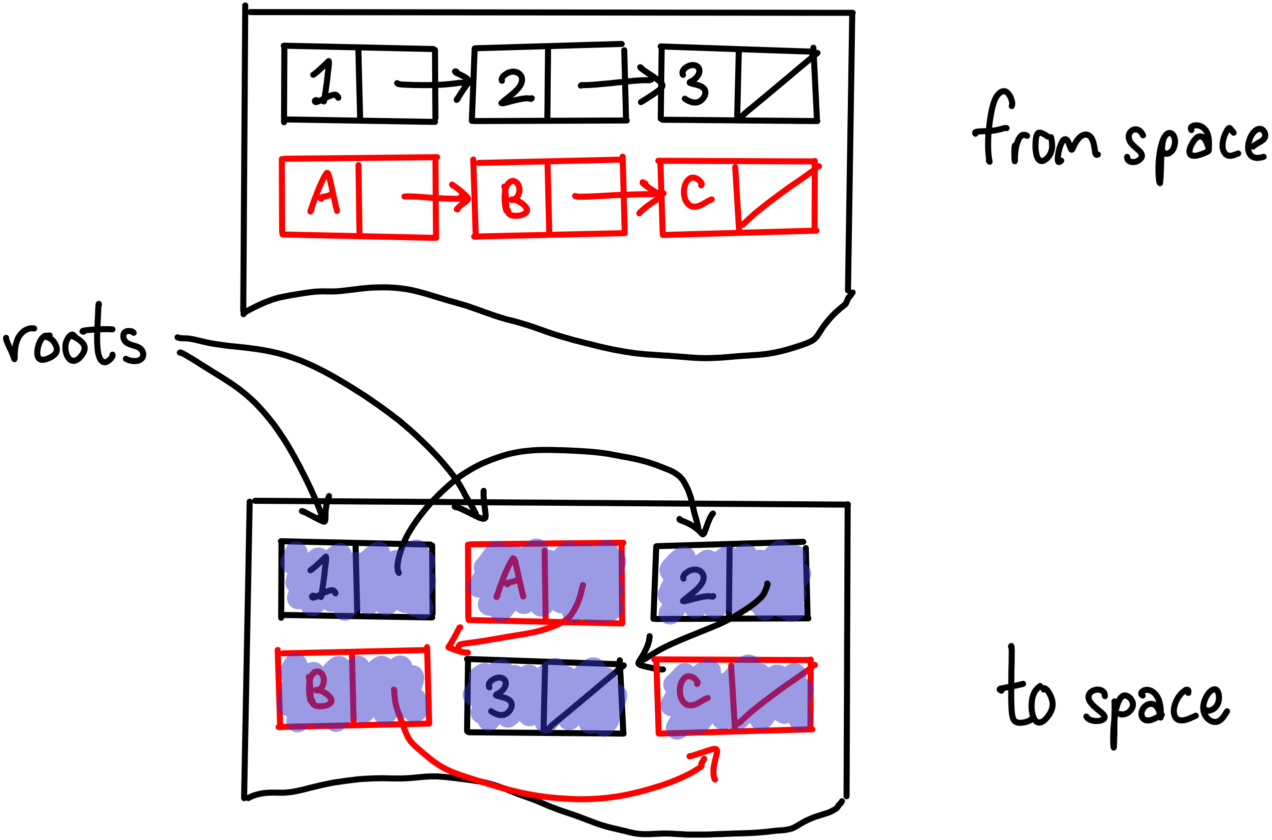
Process the to-do list breadth first



Process the to-do list breadth first



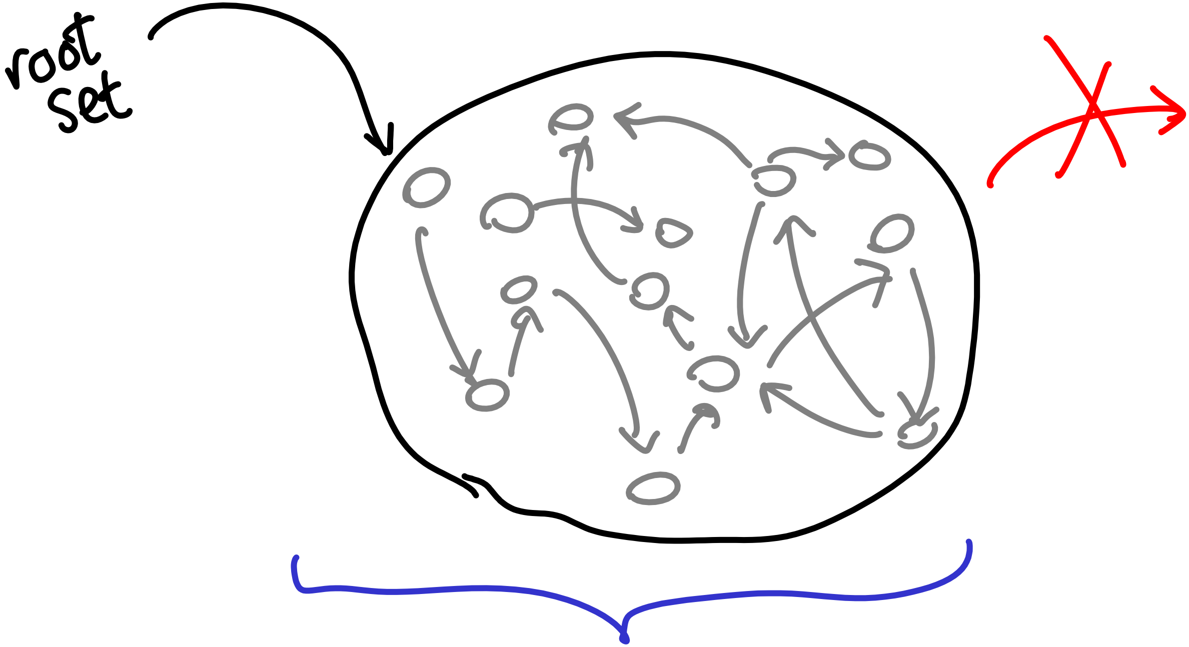
Process the to-do list breadth first



No longer contiguous!

So don't garbage collect it

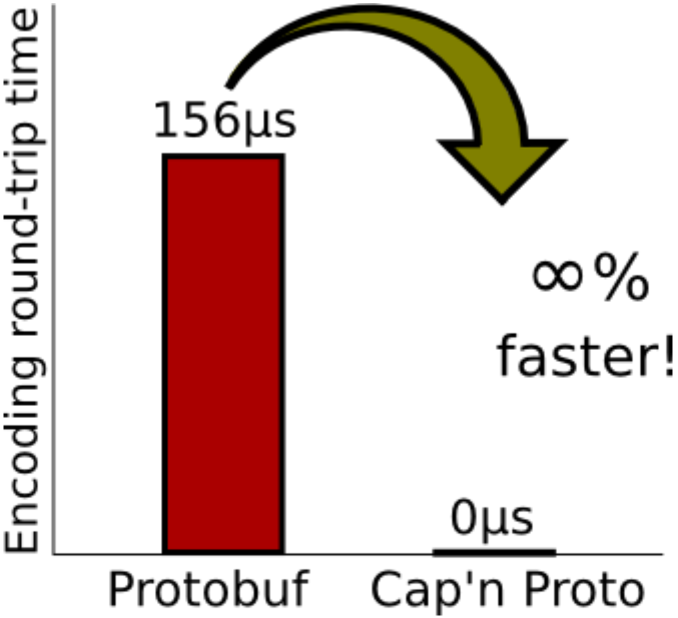
(Does waste space)



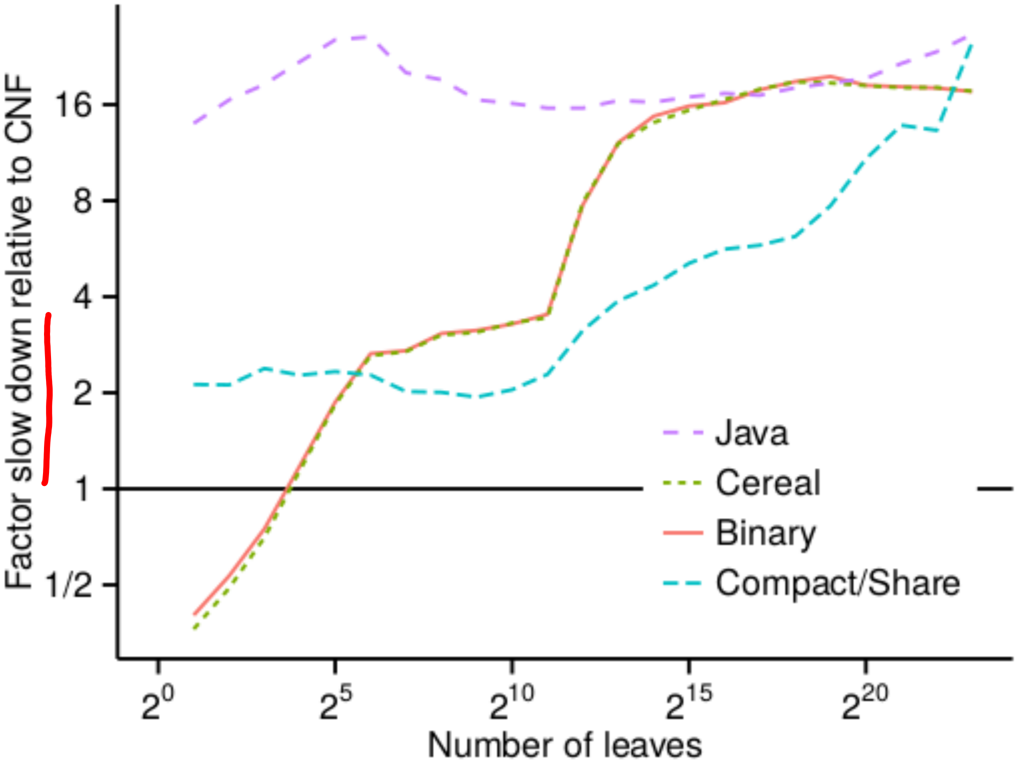
no outbound pointers means no live data!

don't want to trace

OK, but how fast is it?

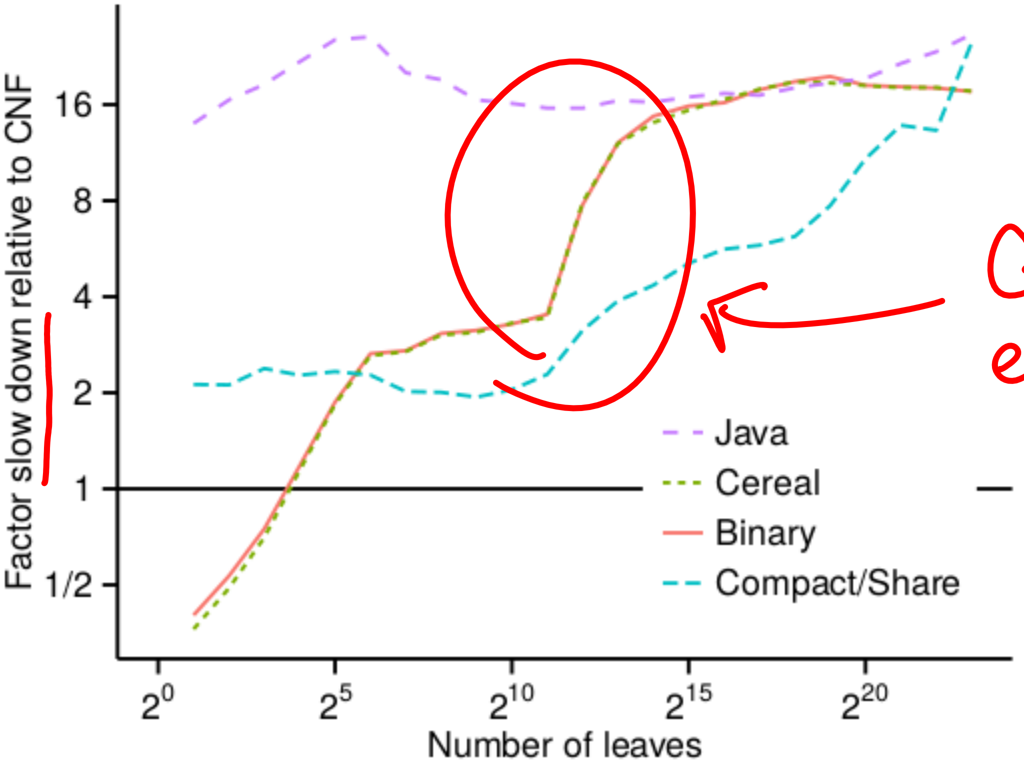

Encoding round-trip time
156µs
0µs
∞%
faster!
Protobuf
Cap'n Proto

Serialization benchmark (binary tree)



gc savings!

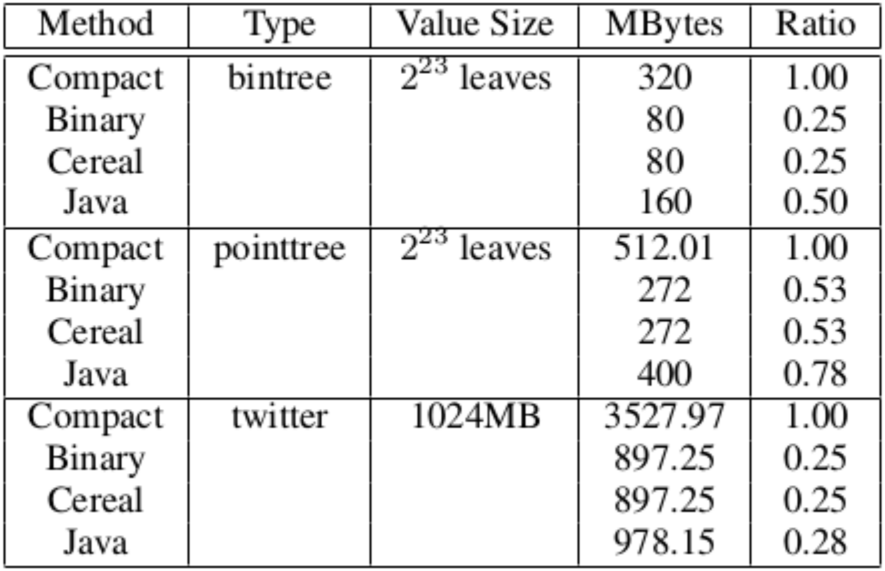
Serialization benchmark (binary tree)



GC effects

gc savings!

Size blowup!

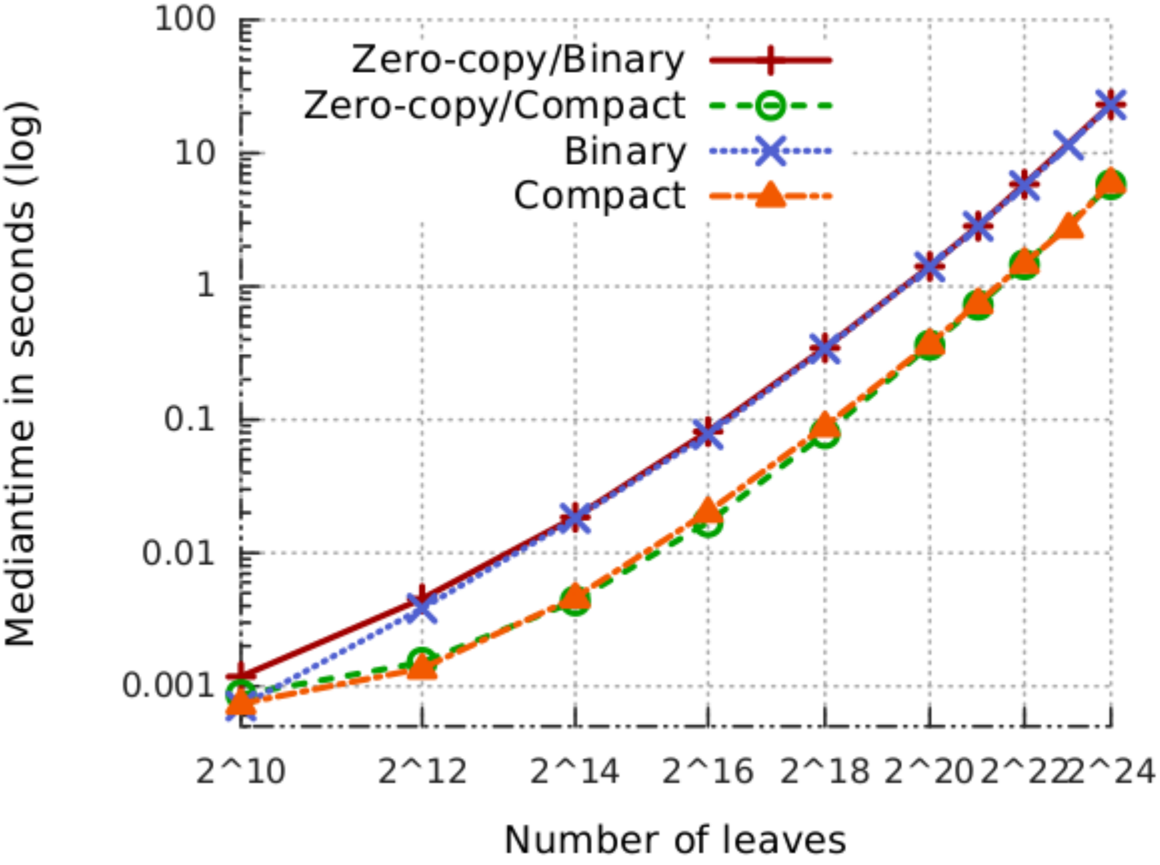


Method	Type	Value Size	MBytes	Ratio
Compact	bintree	2^{23} leaves	320	1.00
Binary			80	0.25
Cereal			80	0.25
Java			160	0.50
Compact	pointtree	2^{23} leaves	512.01	1.00
Binary			272	0.53
Cereal			272	0.53
Java			400	0.78
Compact	twitter	1024MB	3527.97	1.00
Binary			897.25	0.25
Cereal			897.25	0.25
Java			978.15	0.28

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

(NB: serializing took 7s!)

RDMA


Zero-copy/Binary
Zero-copy/Compact
Binary
Compact
Mediantime in seconds (log)
100
10
1
0.1
0.01
0.001
2^10
2^12
2^14
2^16
2^18
2^20
2^22
2^24
Number of leaves

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