How I learned to love immutable data

Matthew Brecknell

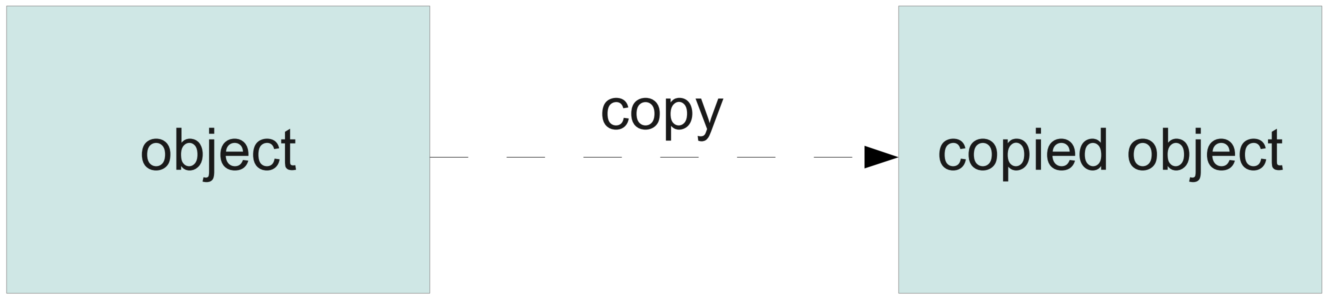
`m.brck.nl/bfpg5`

Q: Why immutable data?

Q: How efficient?

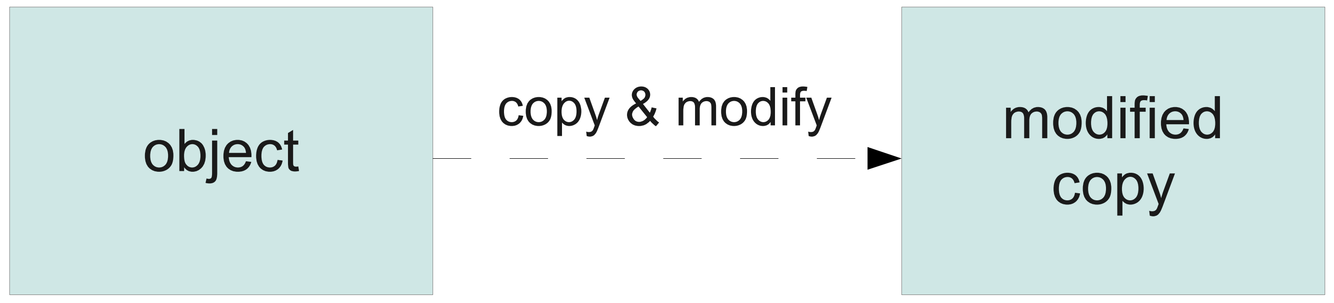
Q: What about implementation?

Lesson #1



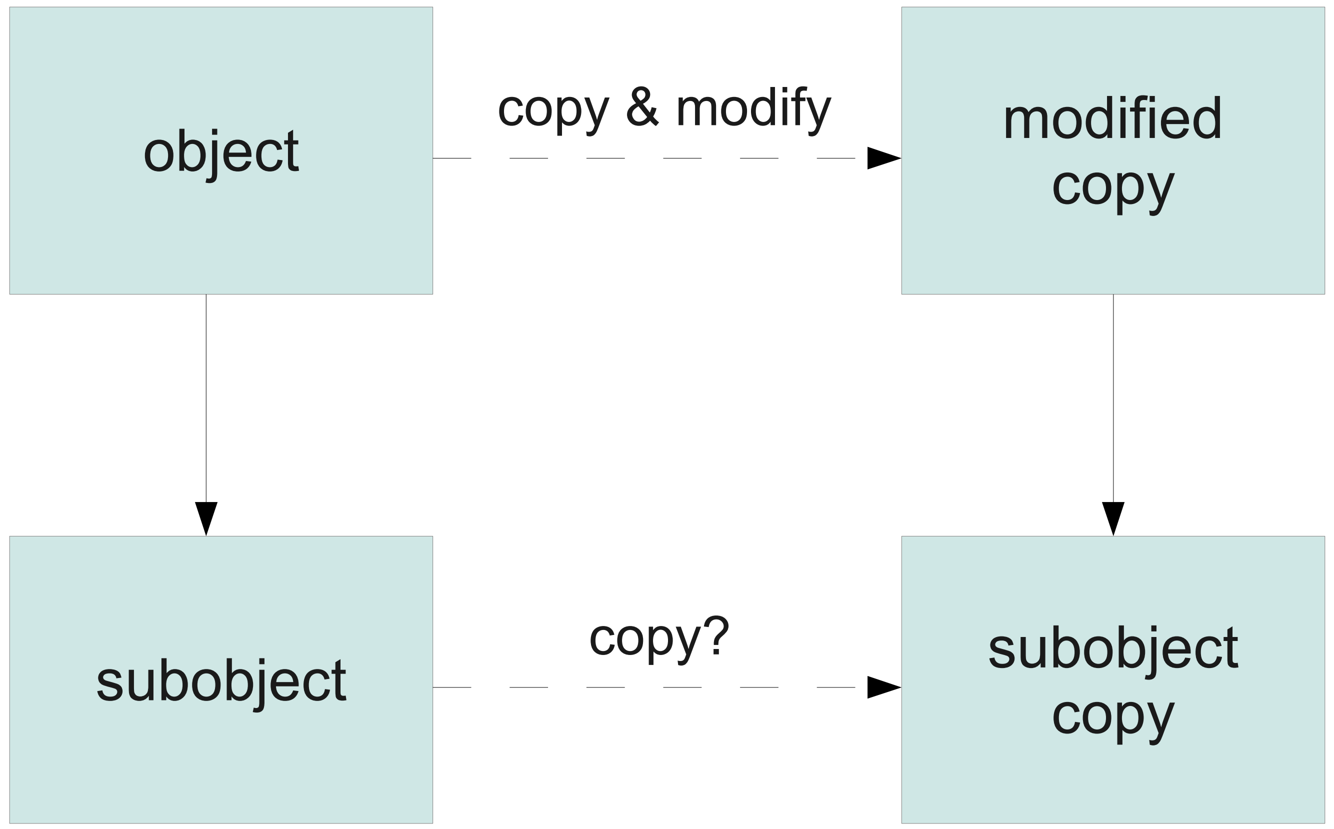
Q: Why copy?

A: To modify one, but not the other.



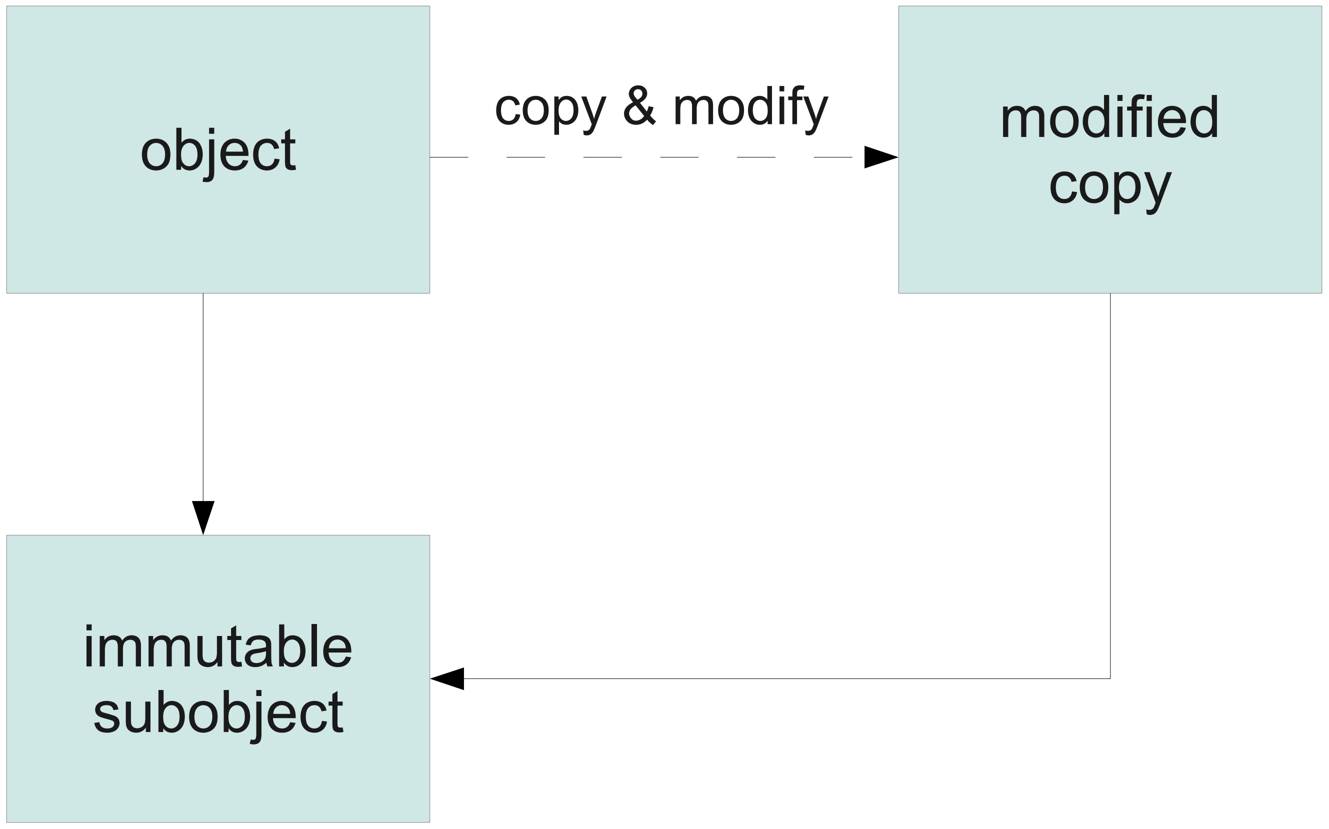
Q: Why copy?

A: To modify one, but not the other.


object
copy & modify
modified copy
subobject
copy?
subobject copy

It depends!

unless...


object
copy & modify
modified copy
immutable subobject

Lesson #1

Immutability admits worry-free sharing.

Lesson #2

Q: How to perform operations on immutable structures?

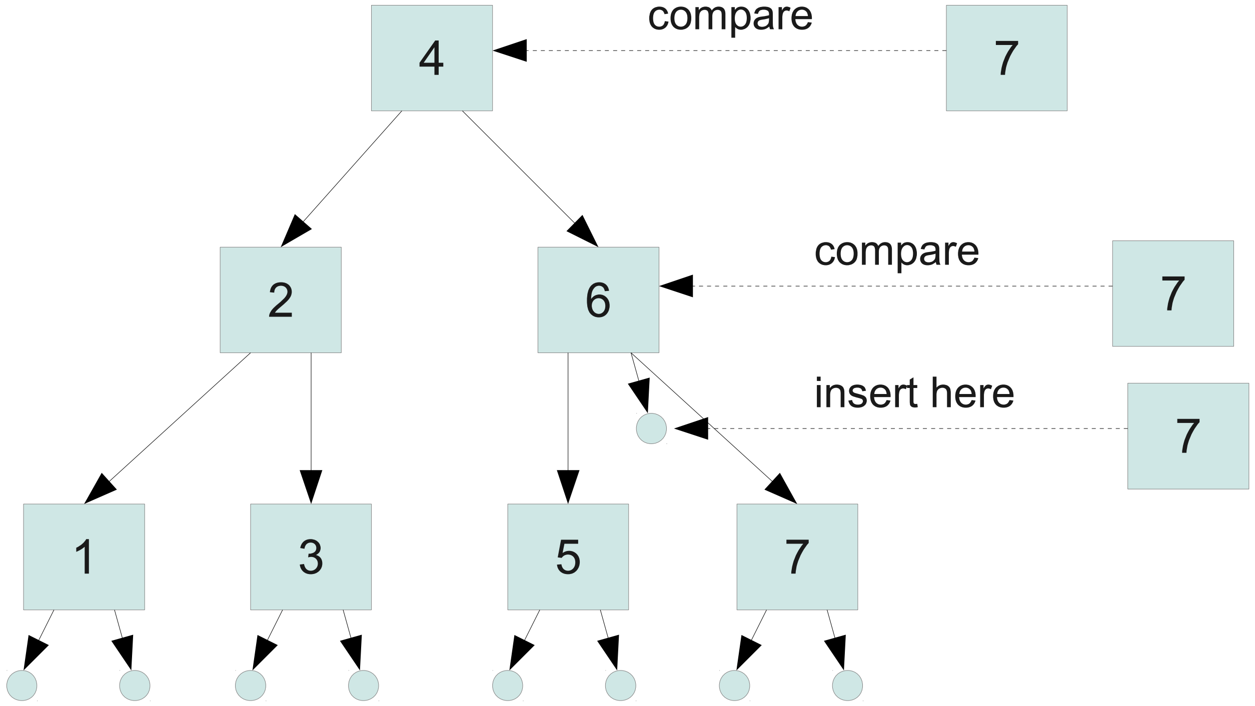
A: Make a modified copy!

```
insert :: Ord t => t → Set t → Set t
```

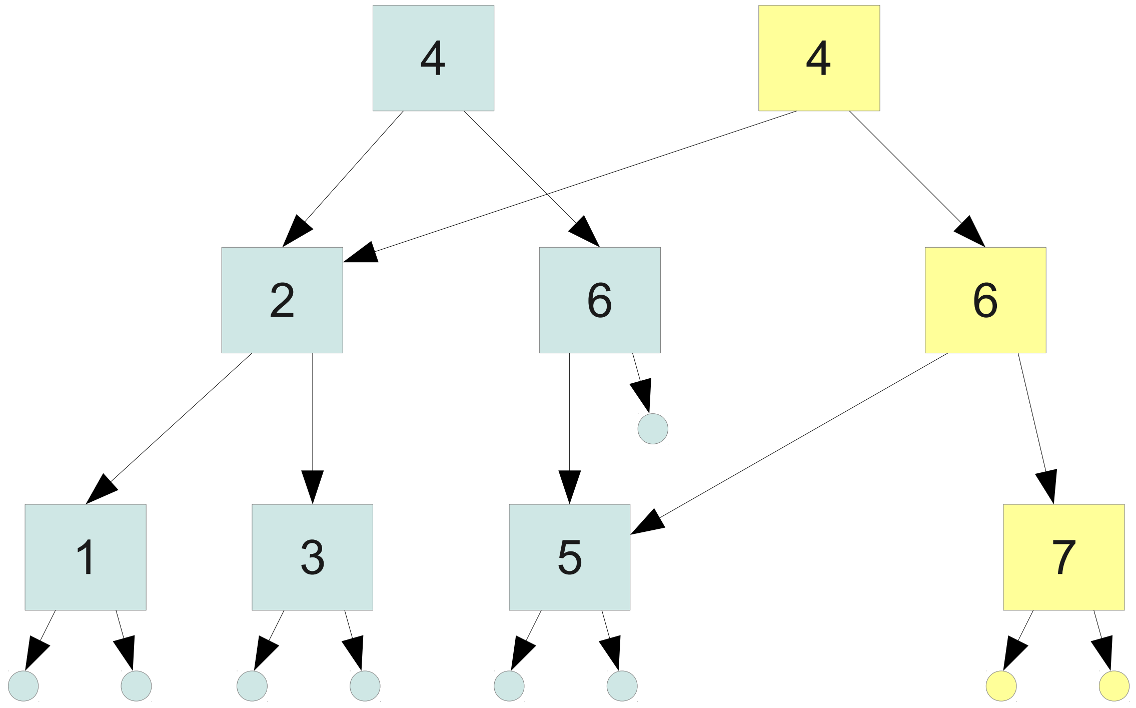
Q: But isn't that inefficient?

A: No!

search tree insertion



immutable search tree insertion



costs: mutable search tree insertion

$k_1L + k_2$

$L = \log_2 N$

k_1 = comparison, navigation

k_2 = construction, update

costs: immutable search tree insertion

$k_3L + k_4$

$L = \log_2 N$

k_3 = comparison, navigation, construction, clean-up

k_4 = construction

incremental costs: mutable copy and insertion

N	(copy the whole tree – ouch!)

incremental costs: immutable copy and insertion

0	(cost already paid)

Lesson #1

Immutability admits worry-free sharing.

Lesson #2

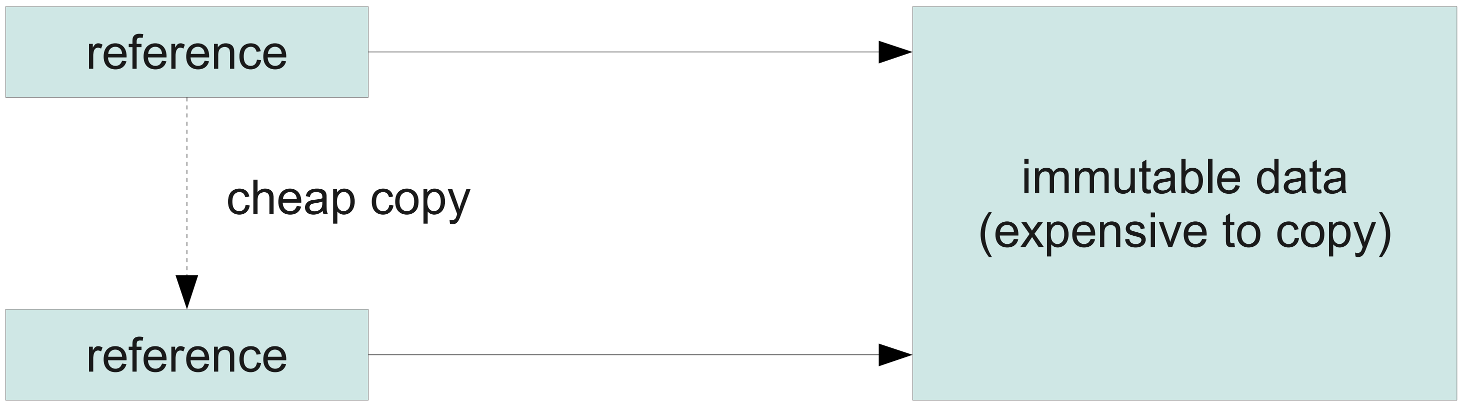
Sharing admits efficient operations on immutable structures.

Lesson #3

Implementing immutable data structures
in an object-oriented language.

As functional programmers, we like value semantics.

But for sharing, we need a reference-based implementation.



Share by copying references, not data.

Warning: C++ ahead!

C++ supports both value semantics (for user view) and reference semantics (for implementation).

C++ also supports compile-time metaprogramming.