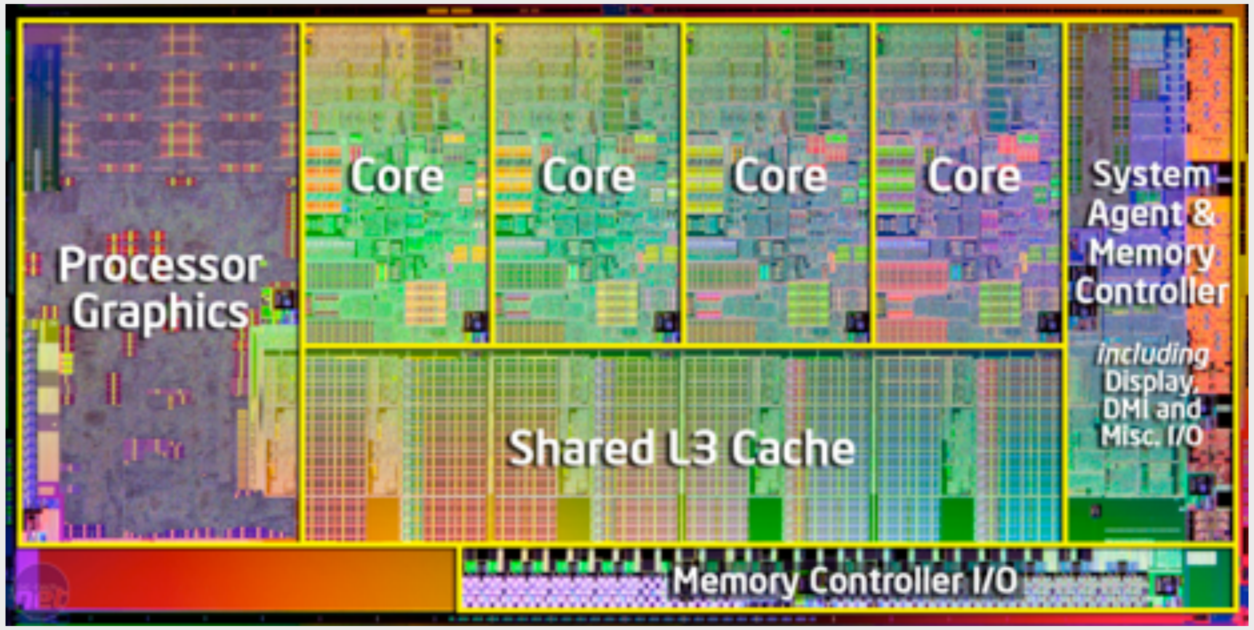
LVars:

Lattice-based Data Structures *for* Deterministic Parallel *and* Distributed Programming

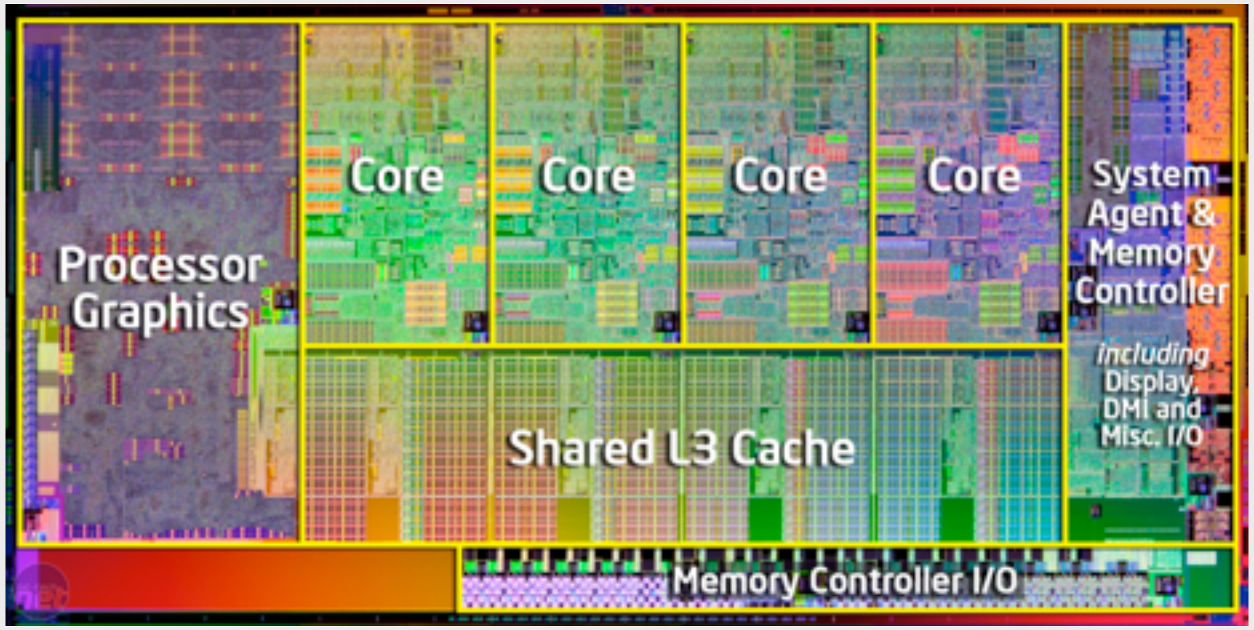
Lindsey Kuper
Indiana University

Intel Labs · Santa Clara, CA
March 21, 2014

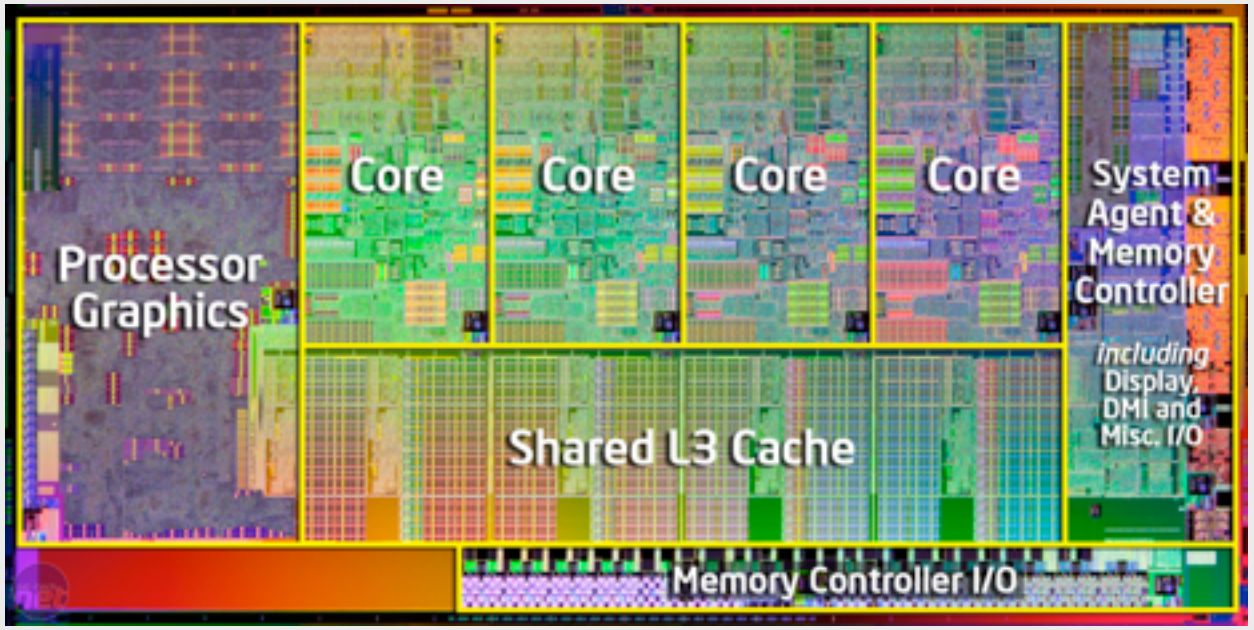


Parallel systems

Distributed systems

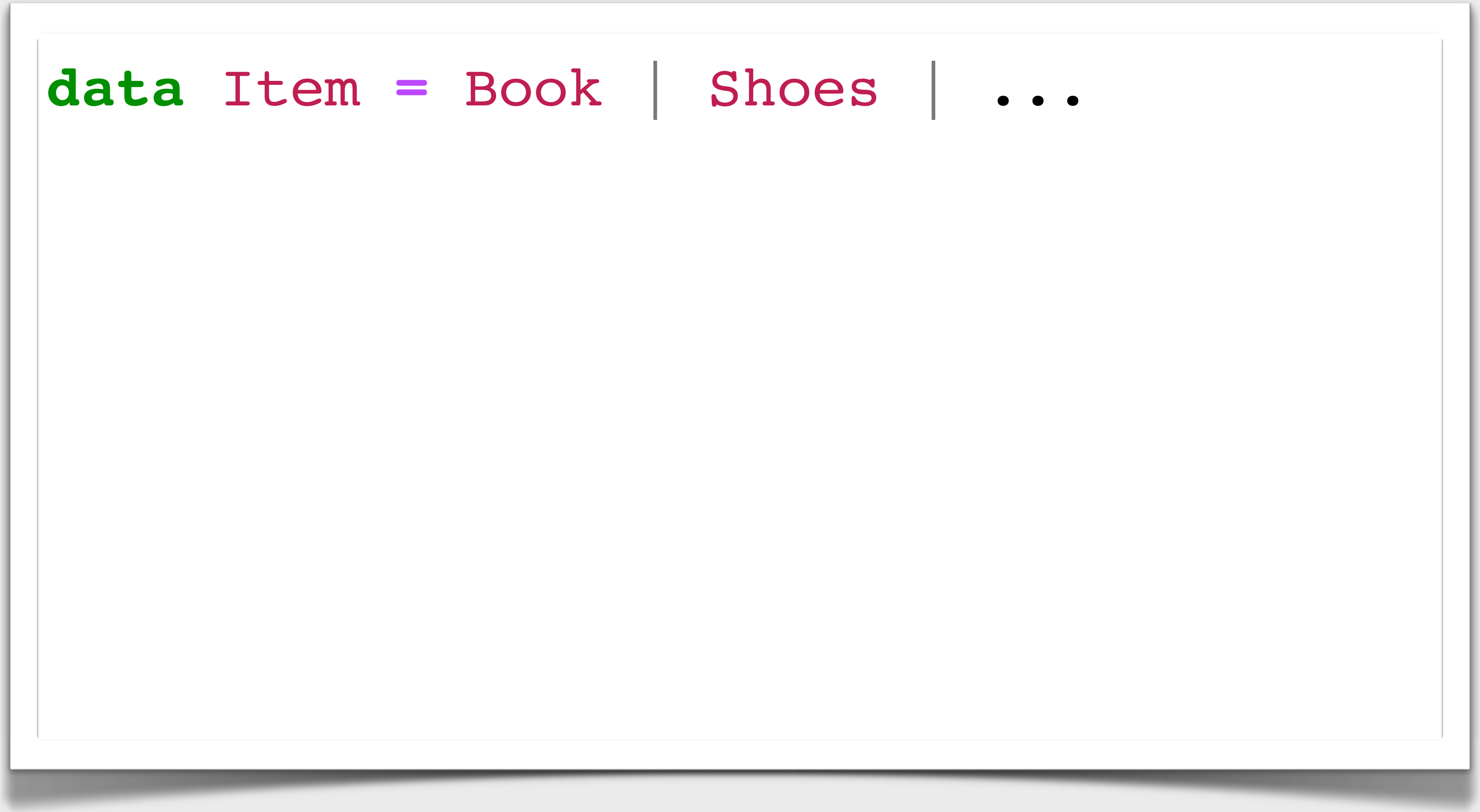


Deterministic Parallel Programming

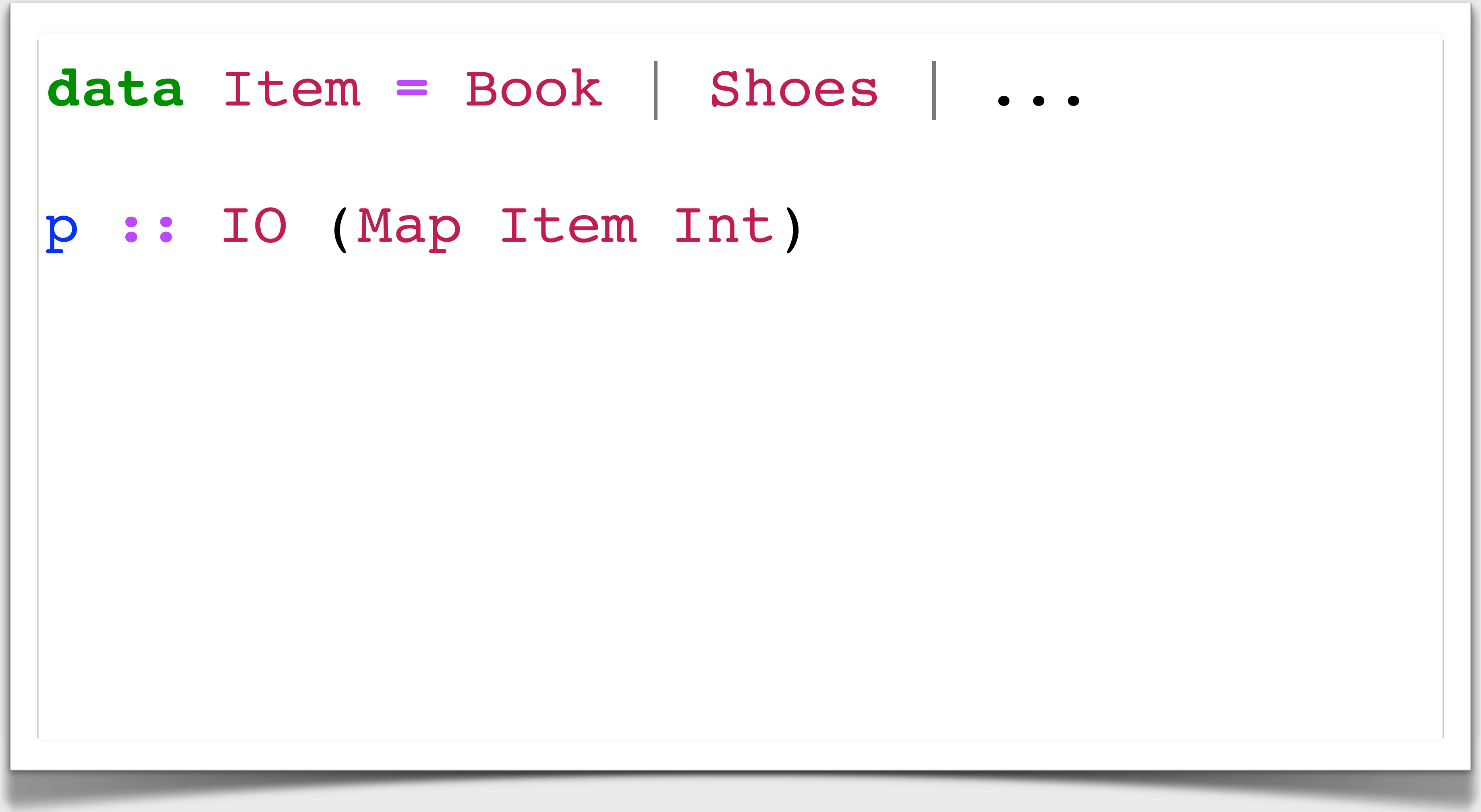


(observably)

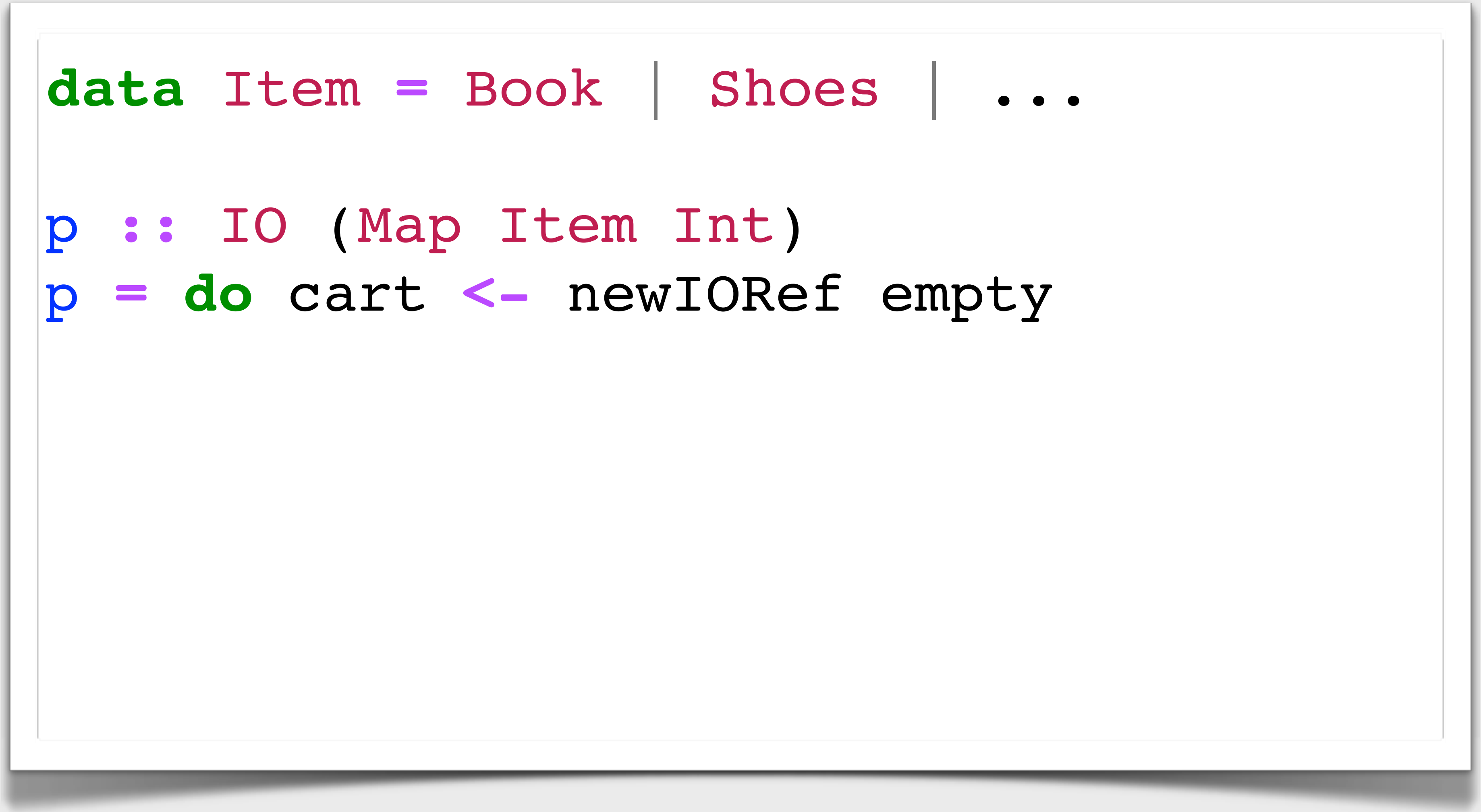
Deterministic Parallel Programming



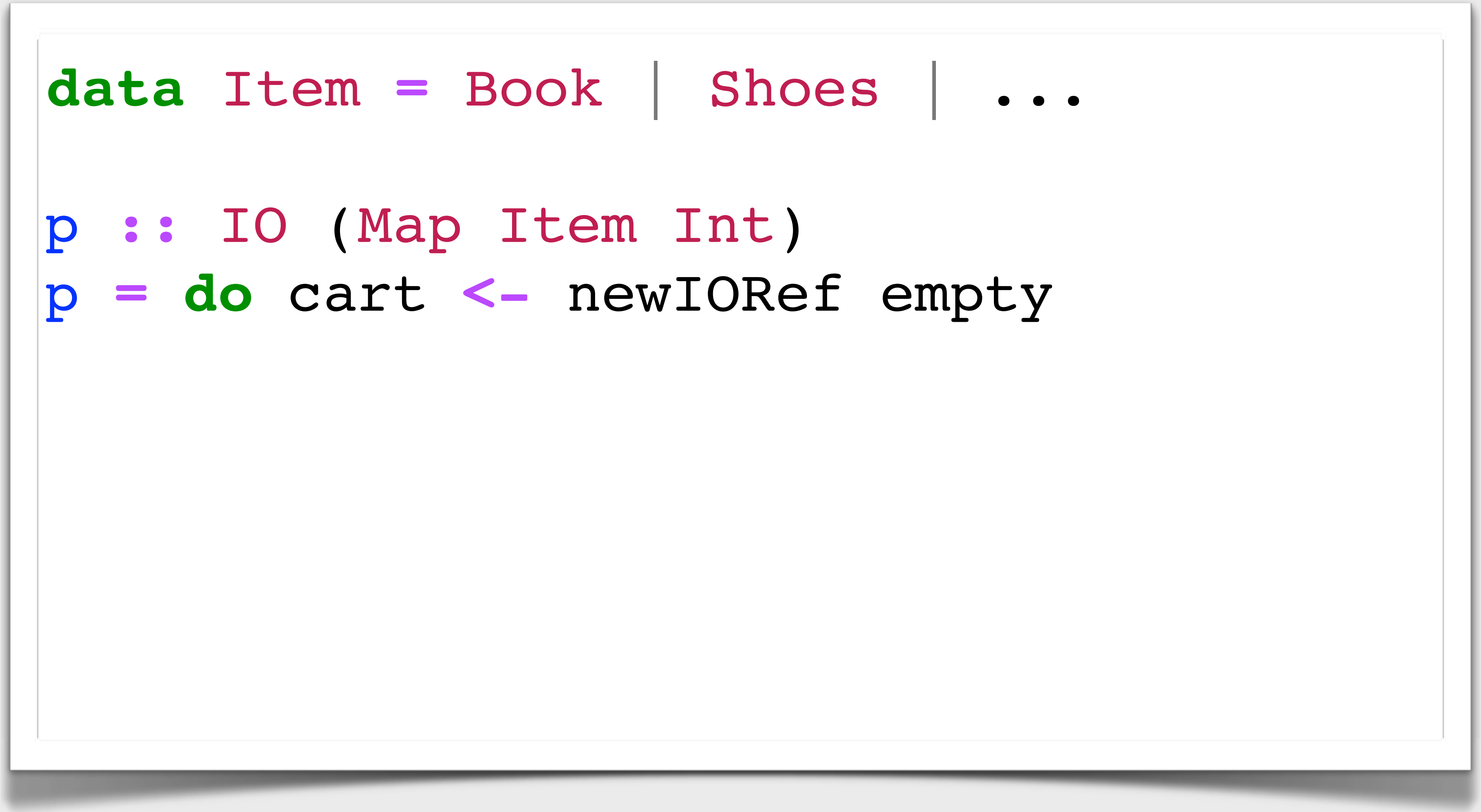
```
data Item = Book | Shoes | ...
```



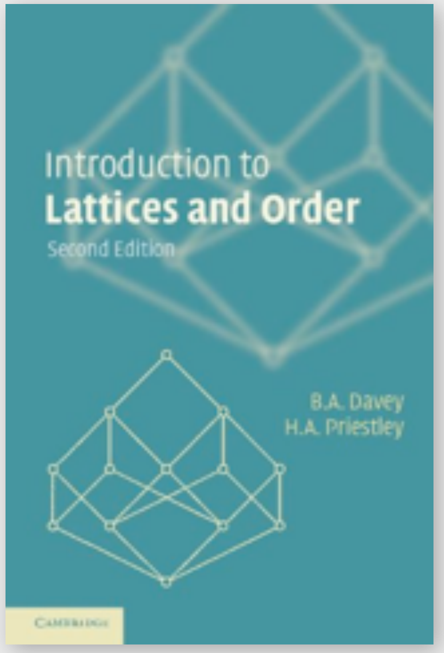
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
```

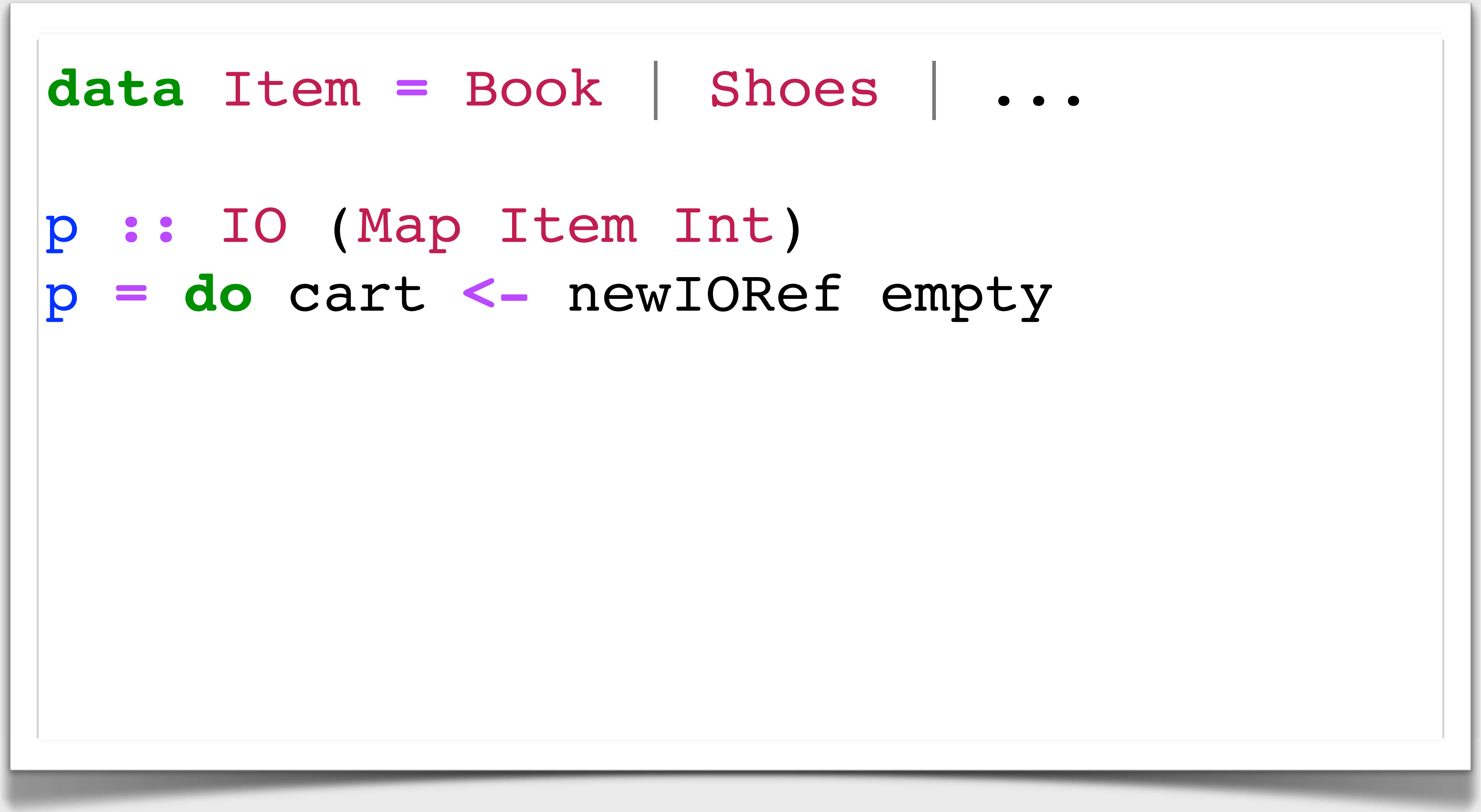


```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```



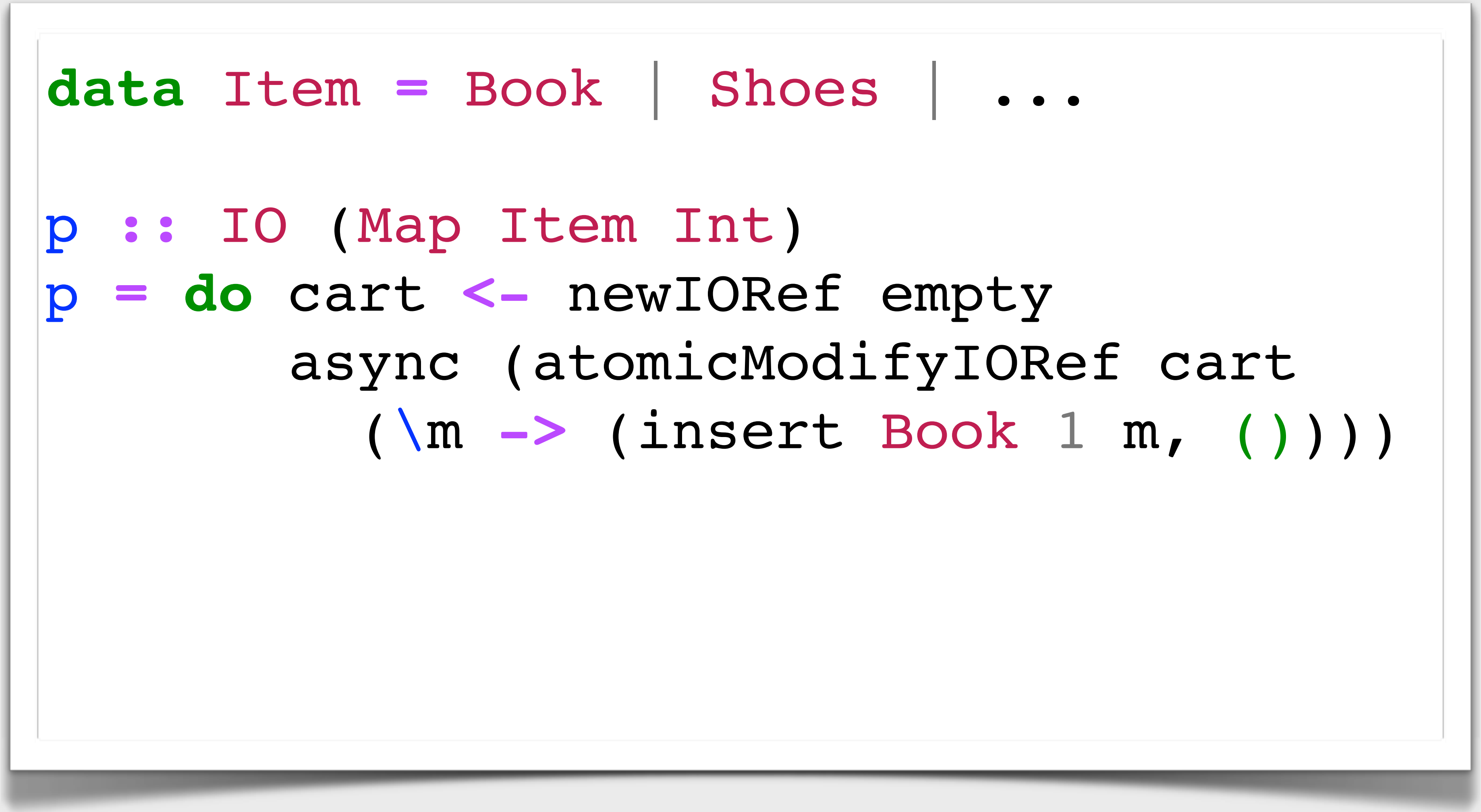
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```





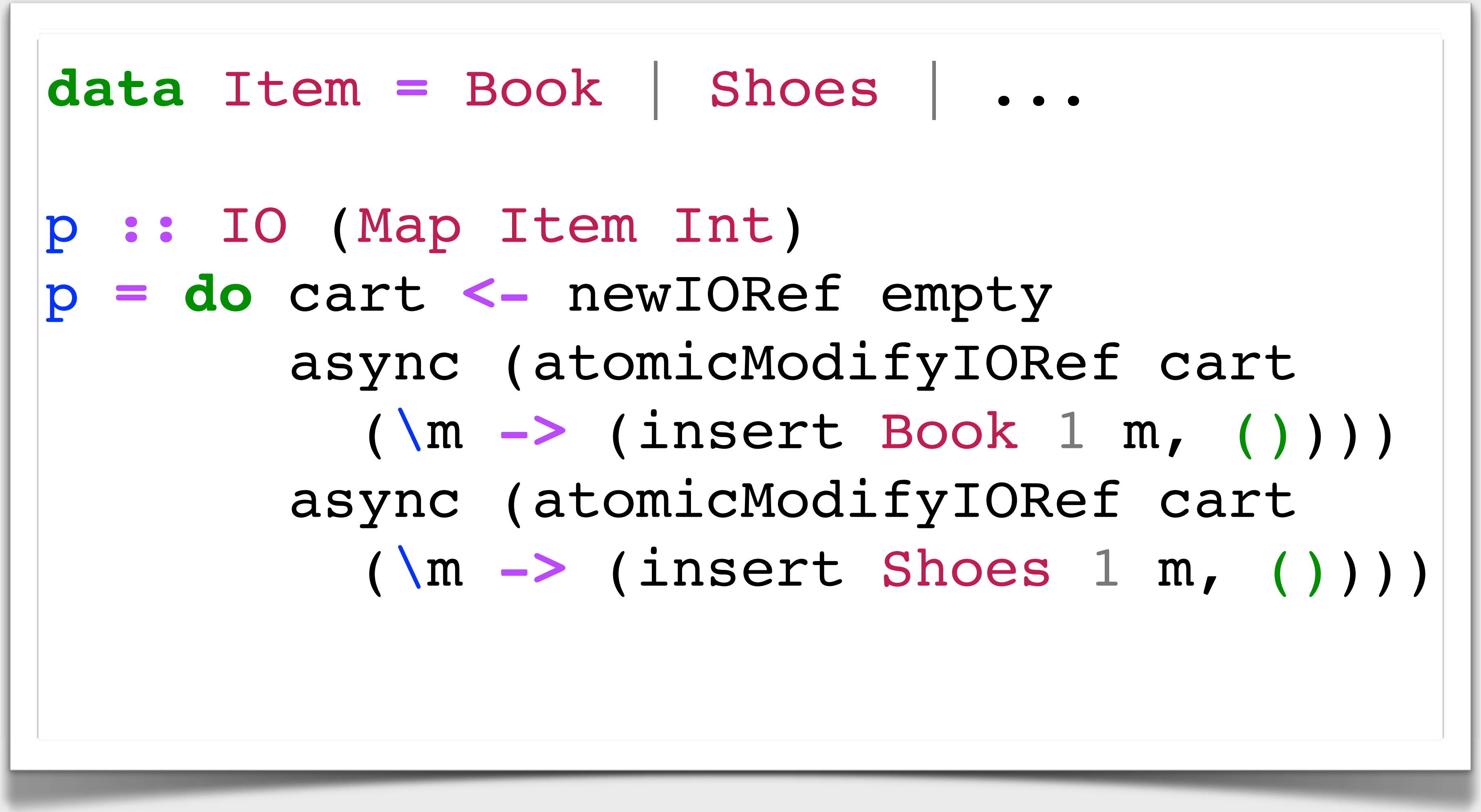
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```



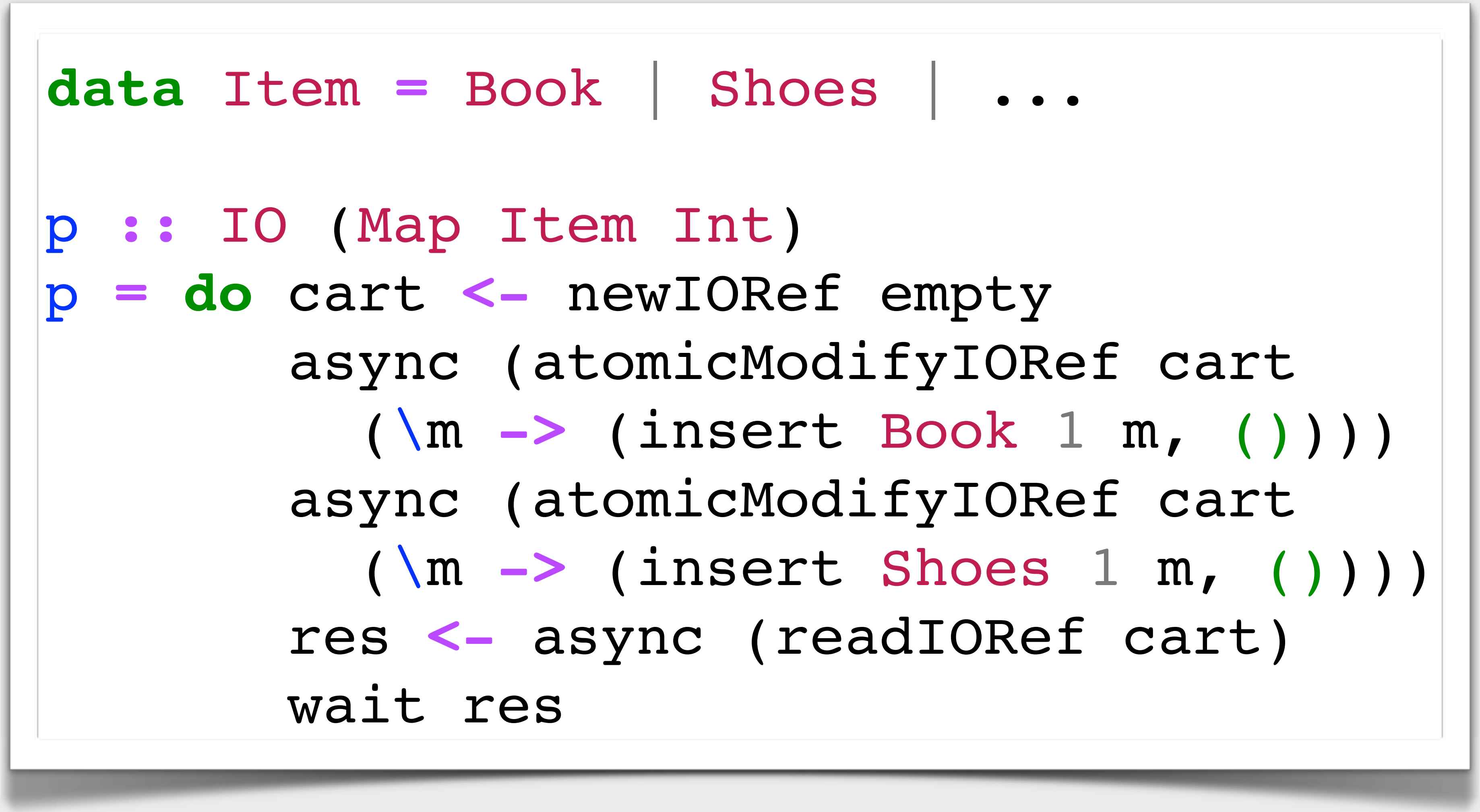
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

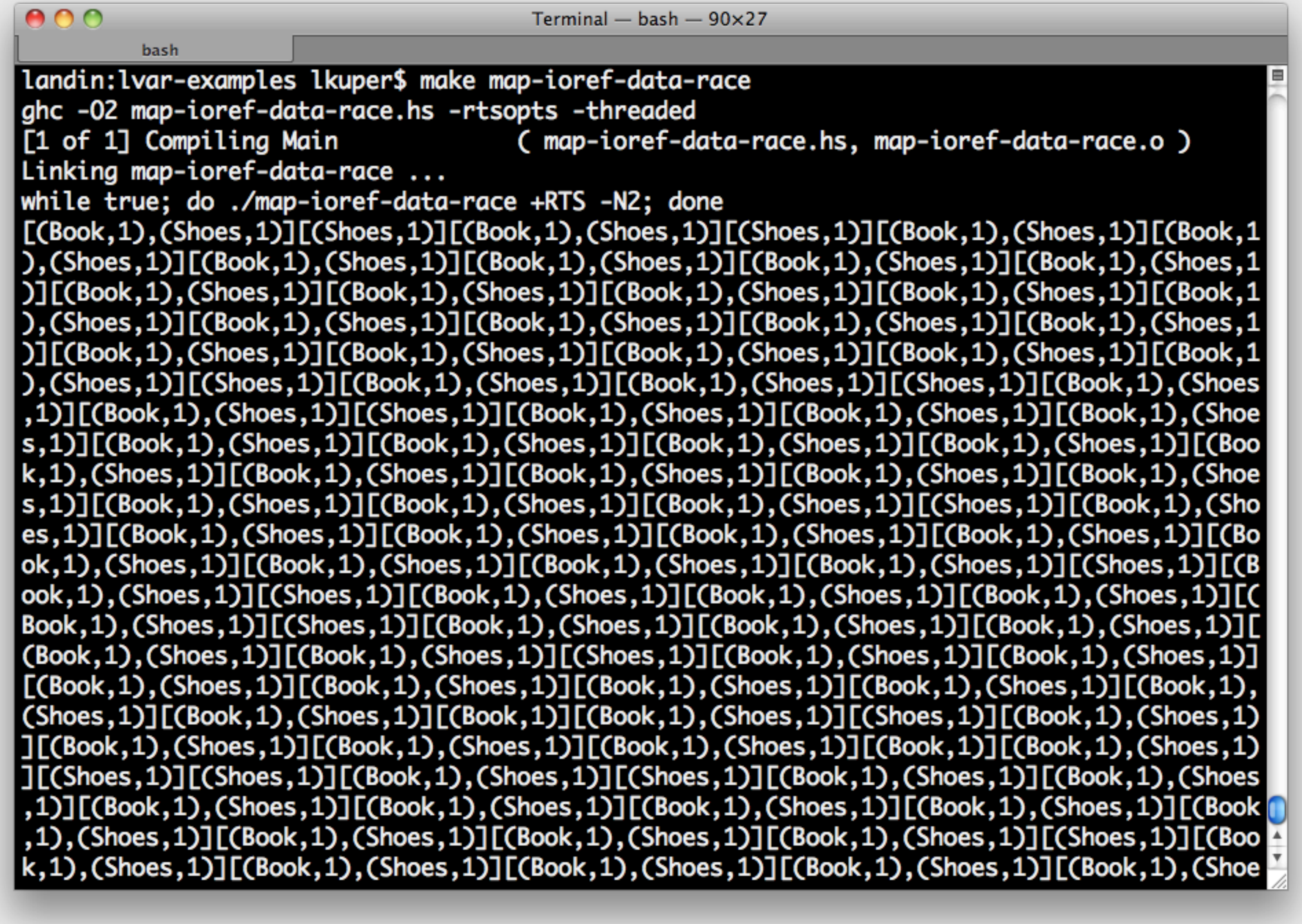


```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
```

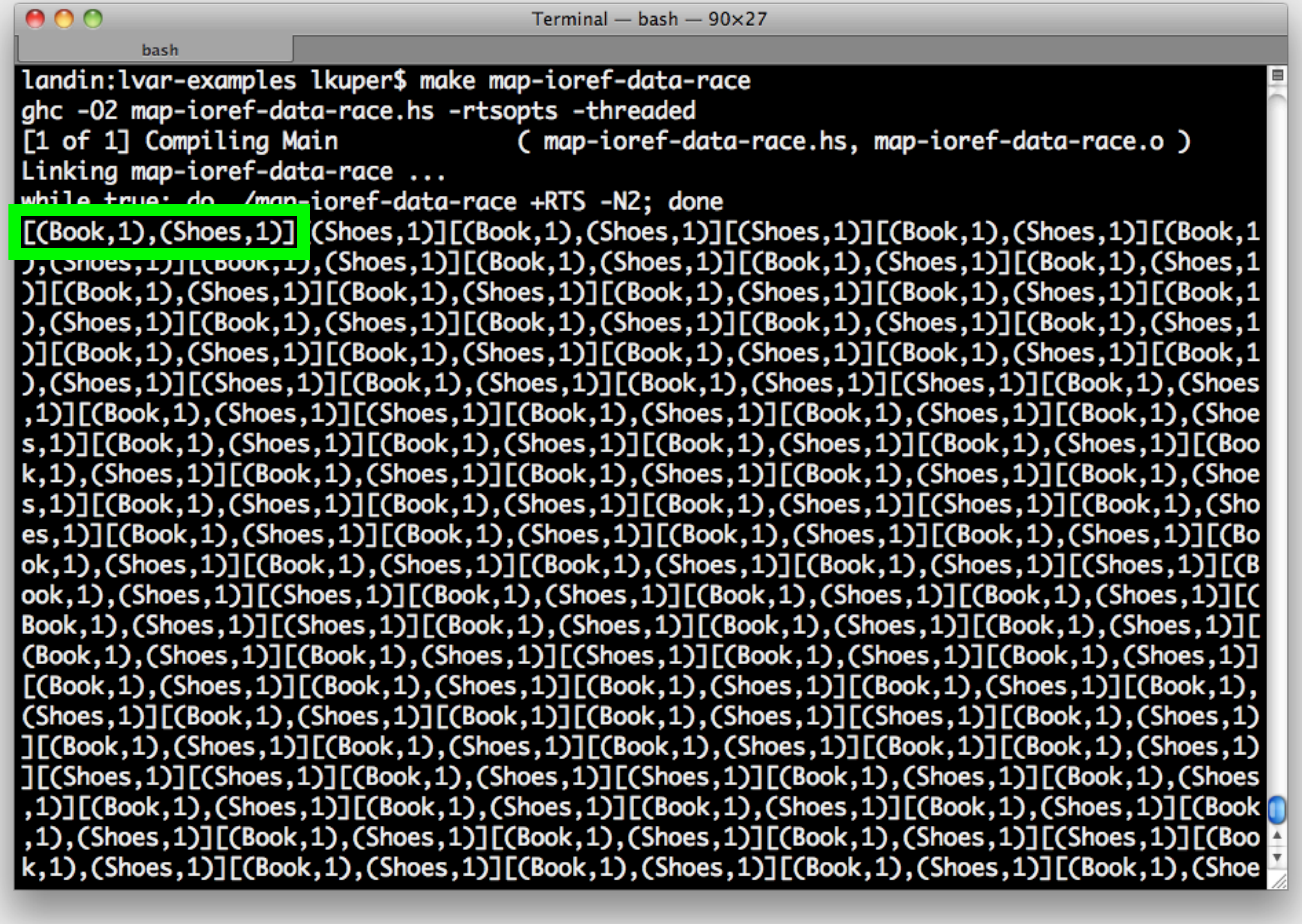
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
```



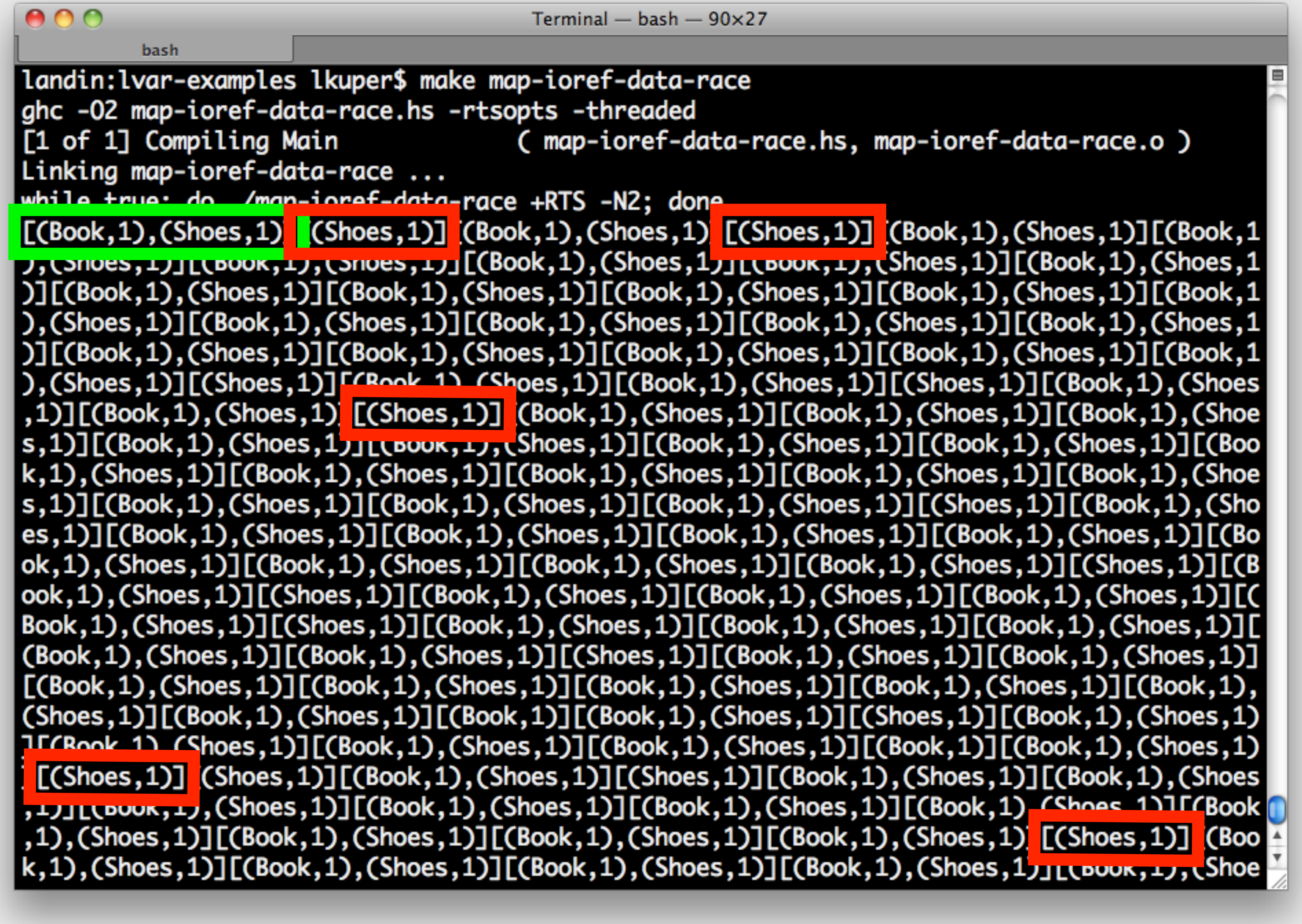
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



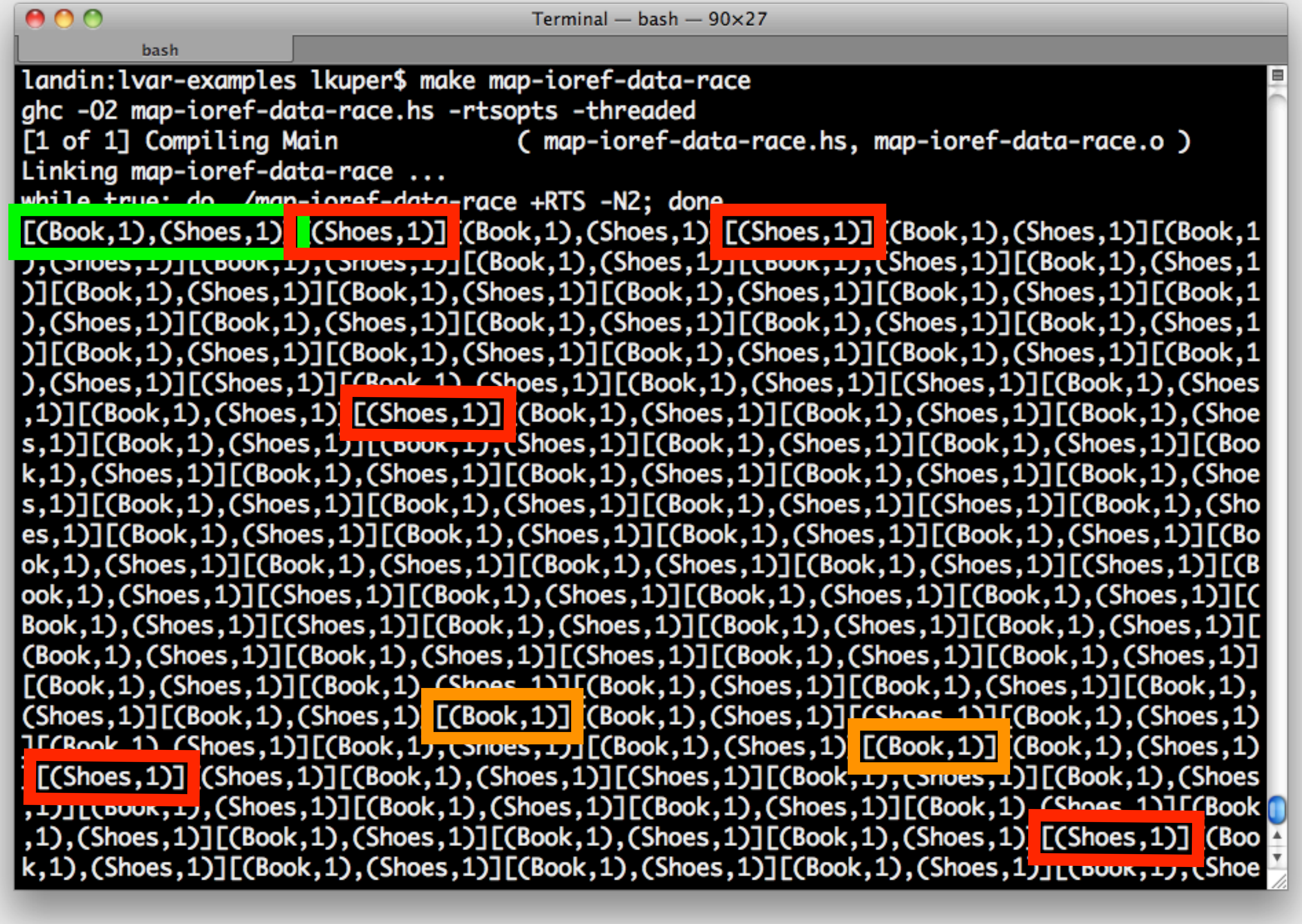
```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```



```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```



```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```



```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

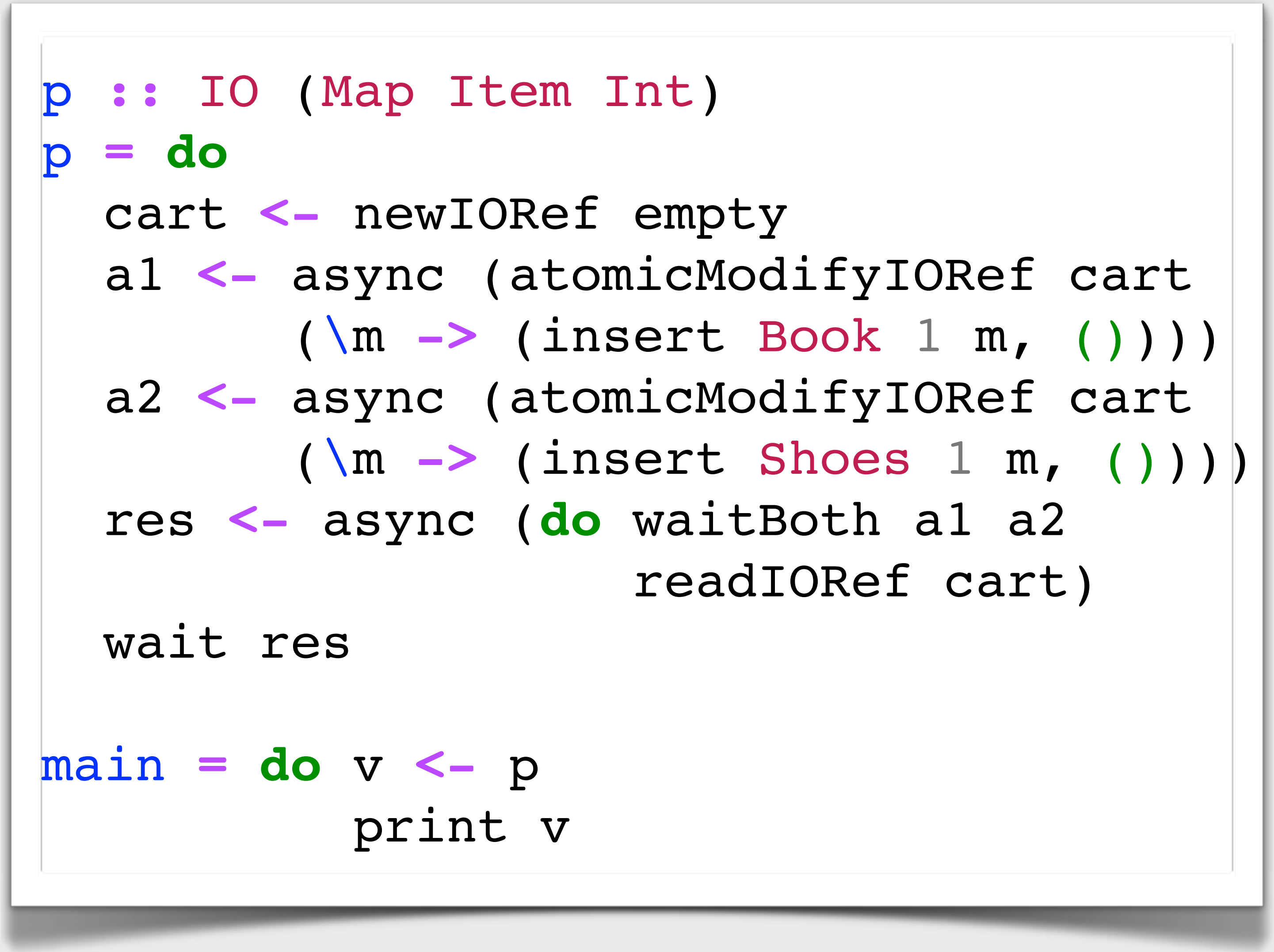
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                      (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
                      (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
                      (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

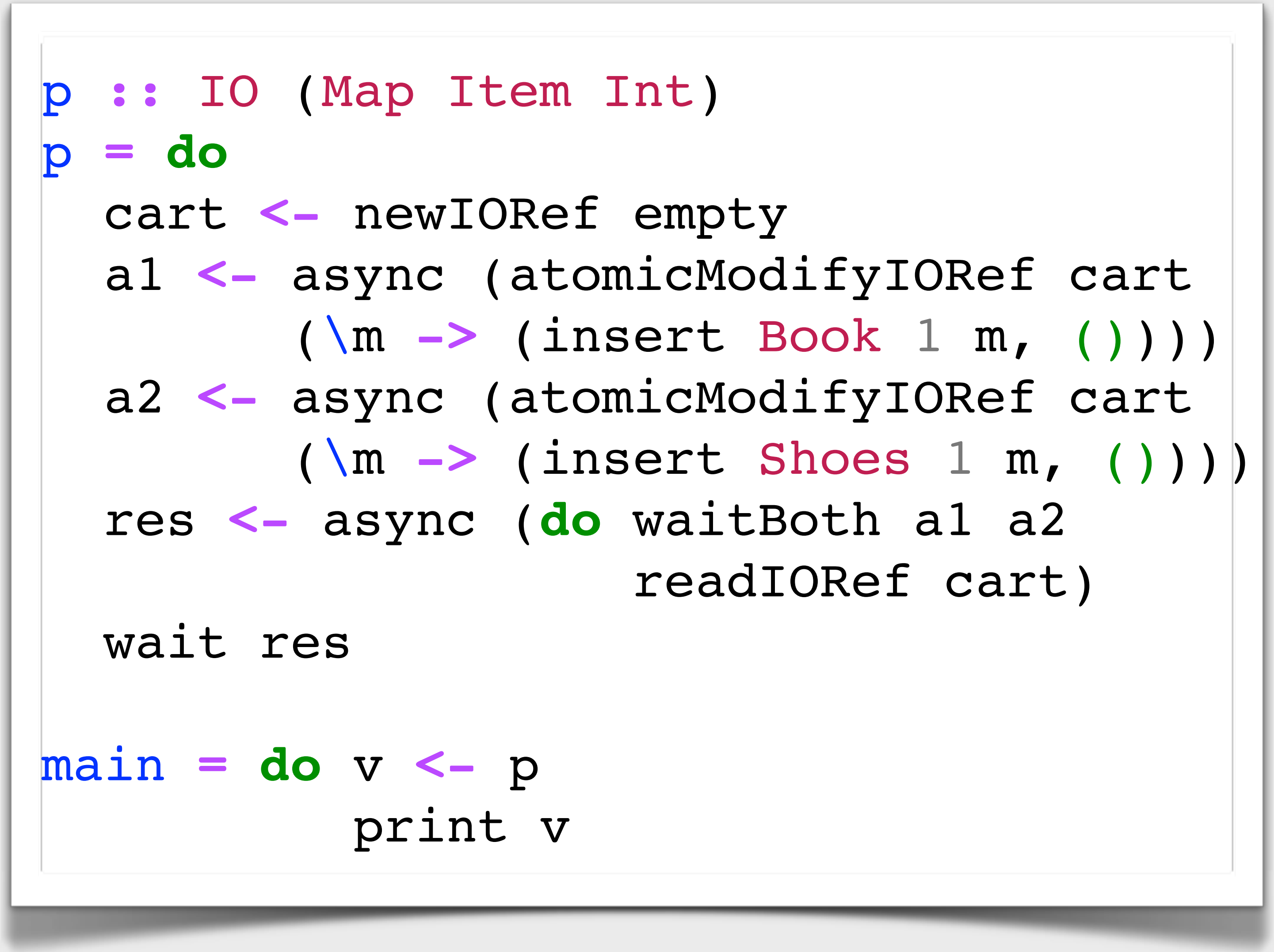
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
               (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
               (\m -> (insert Shoes 1 m, ())))
       res <- async (do waitBoth a1 a2
       wait res         readIORef cart)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async (atomicModifyIORef cart
               (\m -> (insert Book 1 m, ())))
       a2 <- async (atomicModifyIORef cart
               (\m -> (insert Shoes 1 m, ())))
       res <- async (do waitBoth a1 a2
                        readIORef cart)
       wait res
```



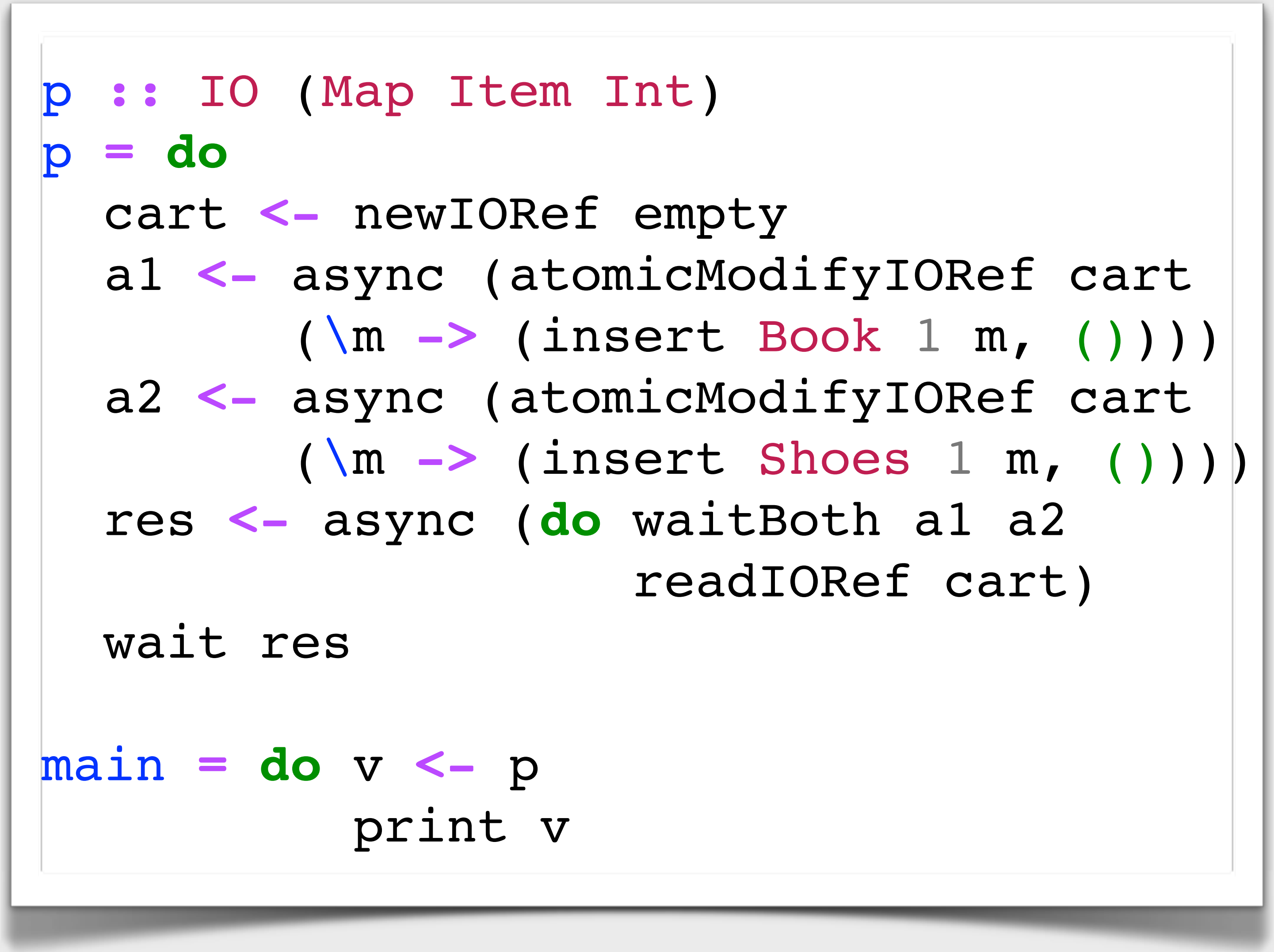
```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic



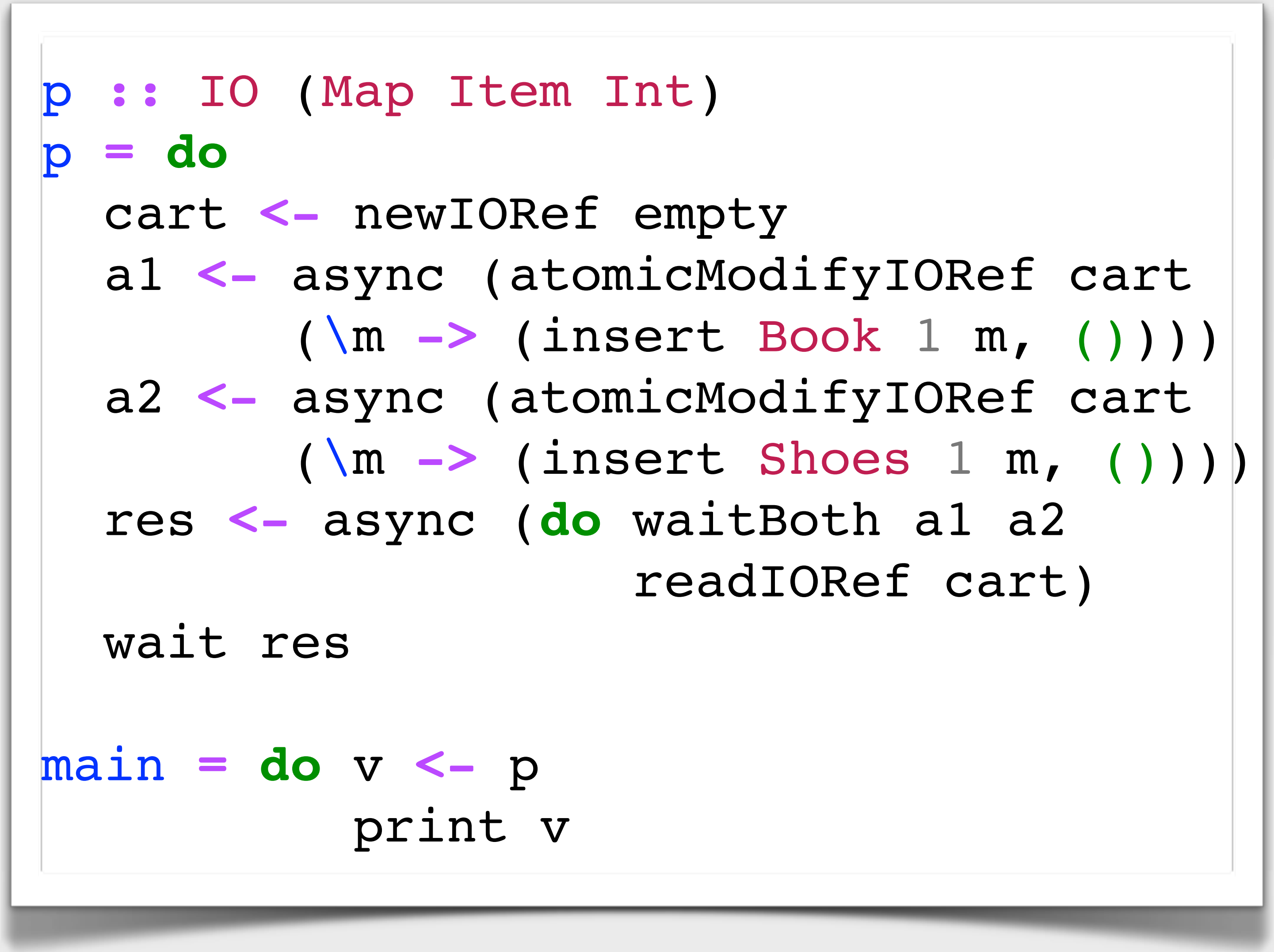
```
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic…now



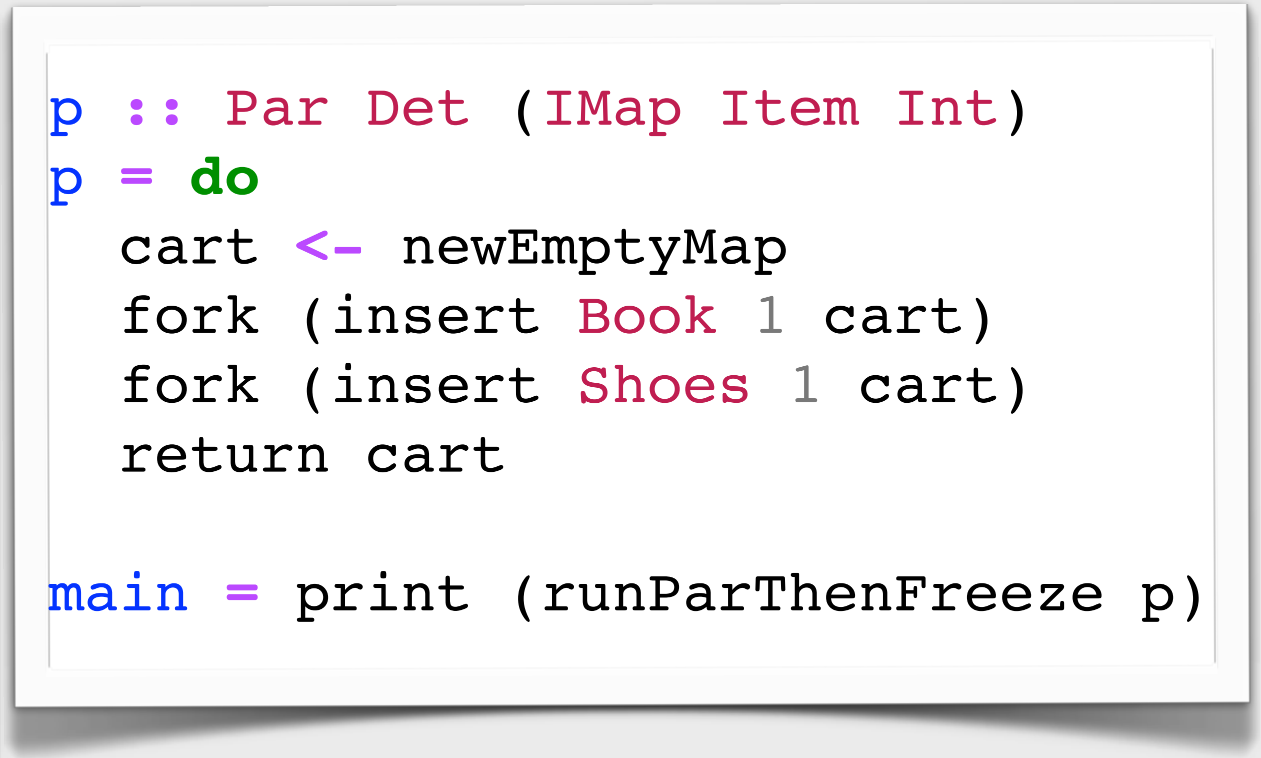
```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic...now...we hope



```
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
  wait res

main = do v <- p
          print v
```

deterministic...now...we hope



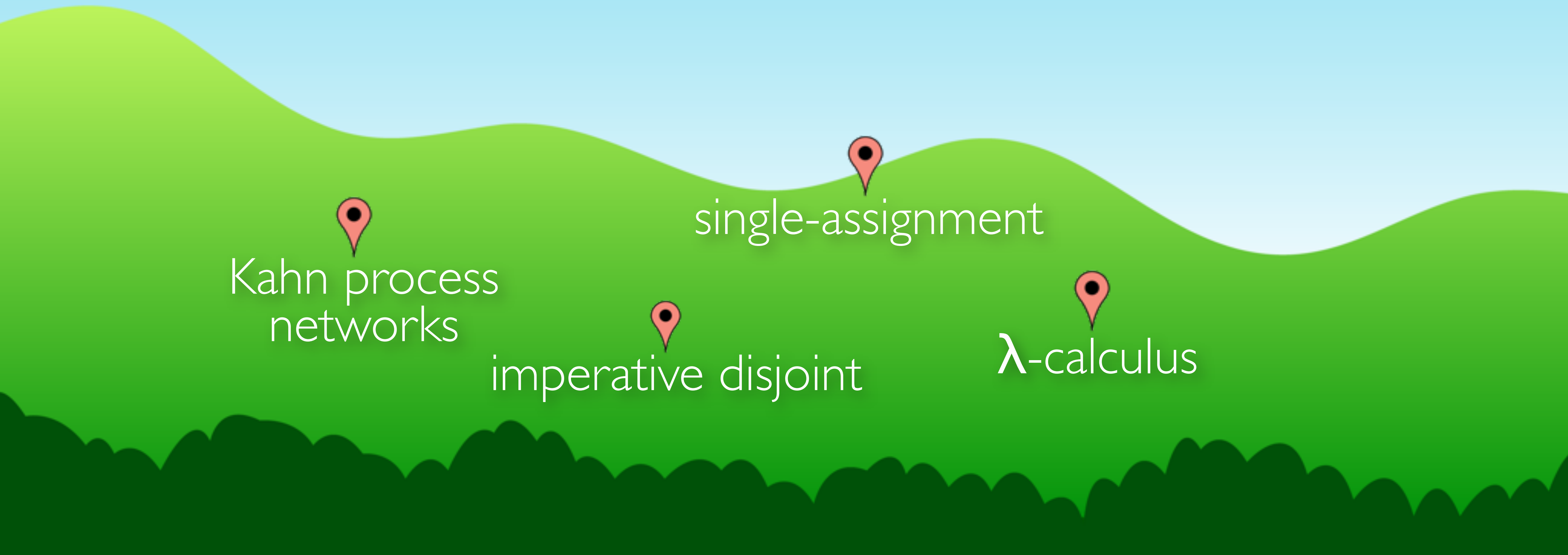
```
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

deterministic by construction

[FHPC '13, POPL '14]

The deterministic by construction parallel programming landscape:

The deterministic by construction parallel programming landscape:



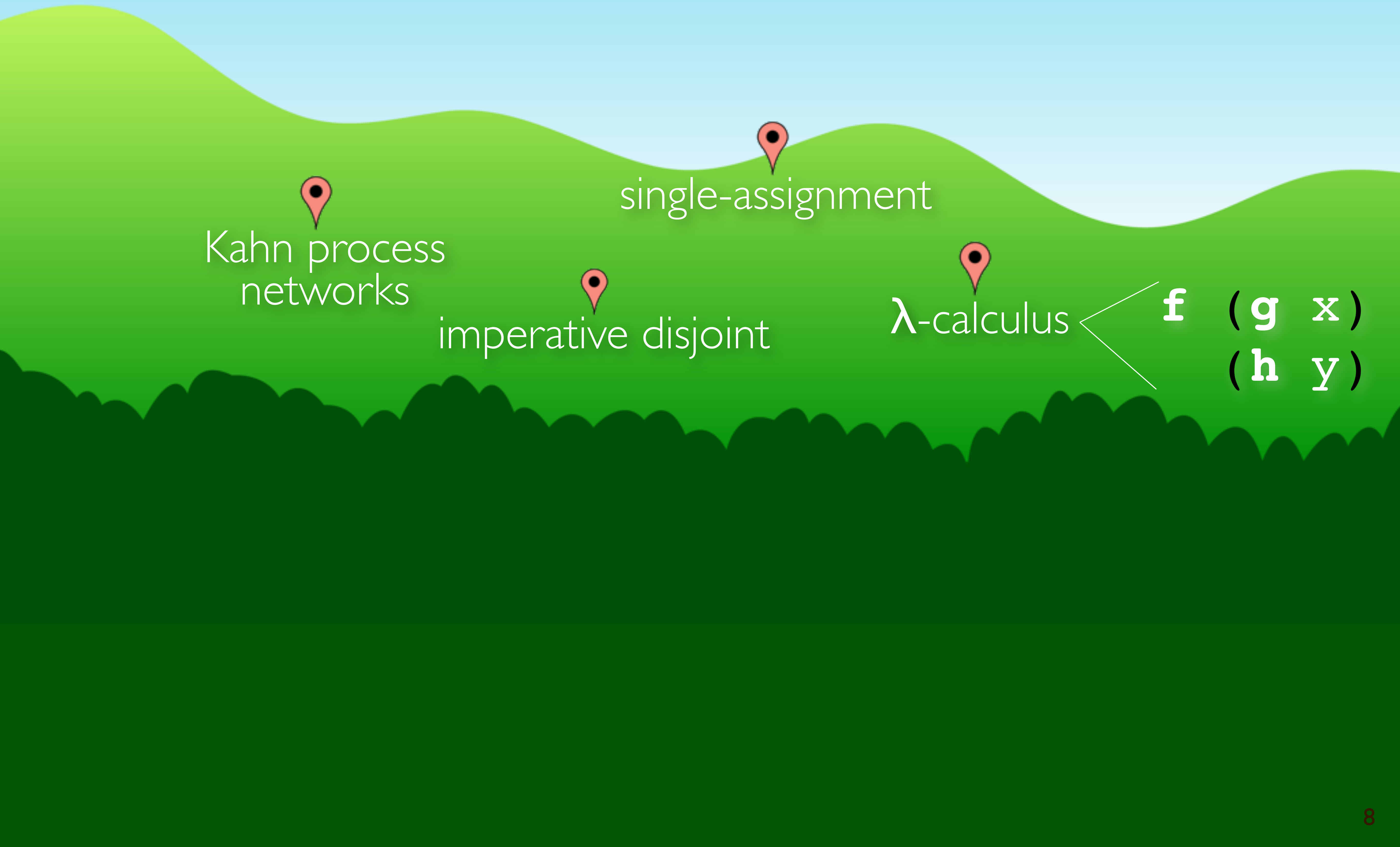
single-assignment

Kahn process networks

imperative disjoint

λ-calculus

The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment

imperative disjoint

λ-calculus

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:

single-assignment

Kahn process networks

imperative disjoint

λ-calculus

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:

Kahn process networks

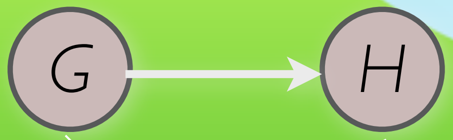
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

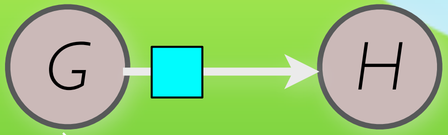
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

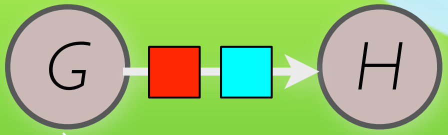
single-assignment

imperative disjoint

λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment

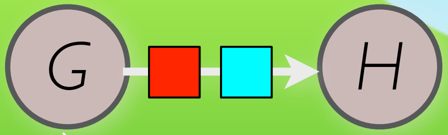
imperative disjoint

λ-calculus

(& array langs, ...)

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

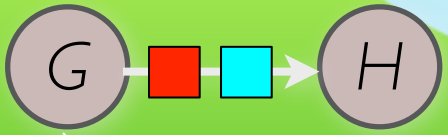
single-assignment
(IVars, CnC, ...)

imperative disjoint

λ-calculus
(& array langs, ...)

```
f (g x)
  (h y)
```

The deterministic by construction parallel programming landscape:

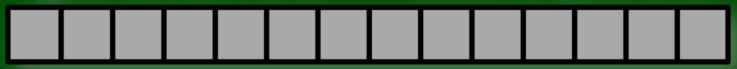


Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint

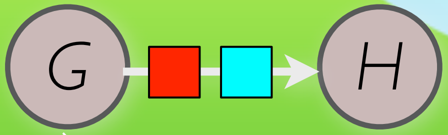
```
g(left)   h(rght)
```



λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

The deterministic by construction parallel programming landscape:

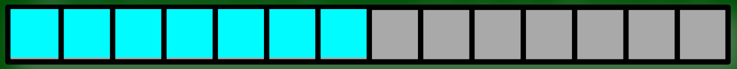


Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint

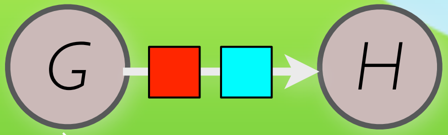
```
g(left)   h(rght)
```



λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

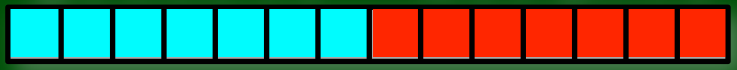
The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint

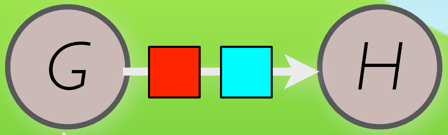


```
g(left)   h(rght)
```

λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

The deterministic by construction parallel programming landscape:

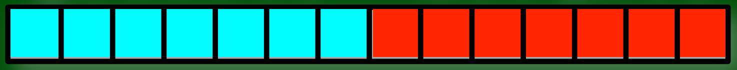


Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint
(DPJ, ...)

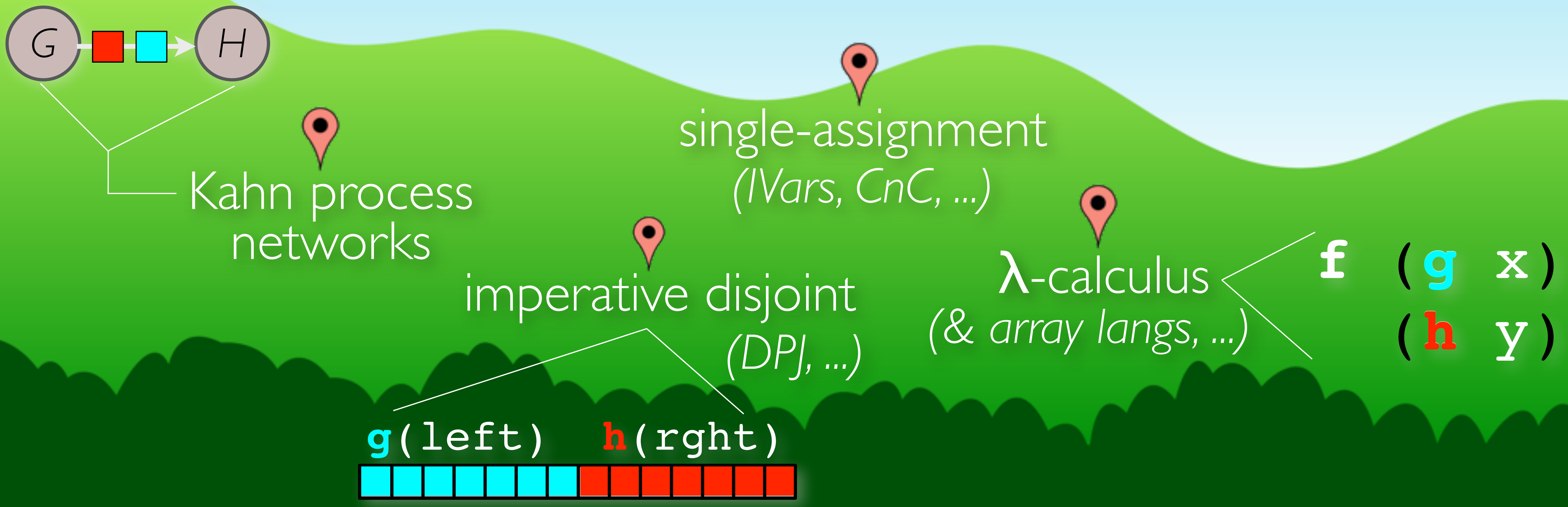
```
g(left)   h(rght)
```



λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

The deterministic by construction parallel programming landscape:



Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint
(DPJ, ...)

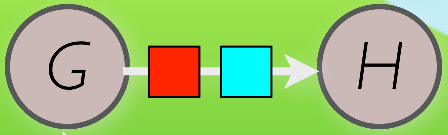
```
g(left)   h(rght)
```

λ-calculus
(& array langs, ...)

```
f  (g  x)
   (h  y)
```

Can we *generalize* and *unify* these points on the map?

The deterministic by construction parallel programming landscape:

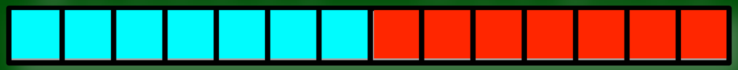


Kahn process networks

single-assignment
(IVars, CnC, ...)

imperative disjoint
(DPJ, ...)

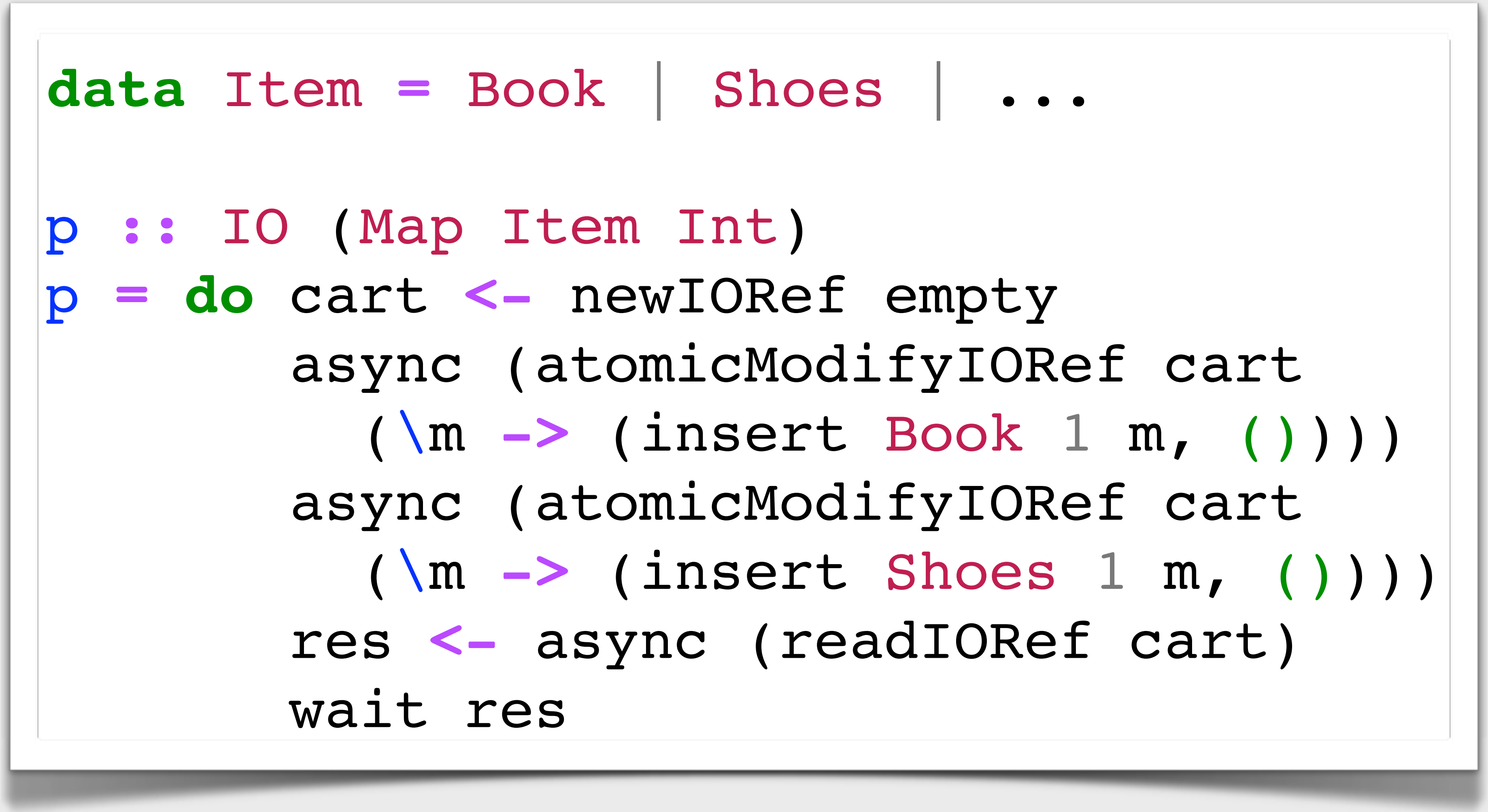
```
g(left)  h(rght)
```



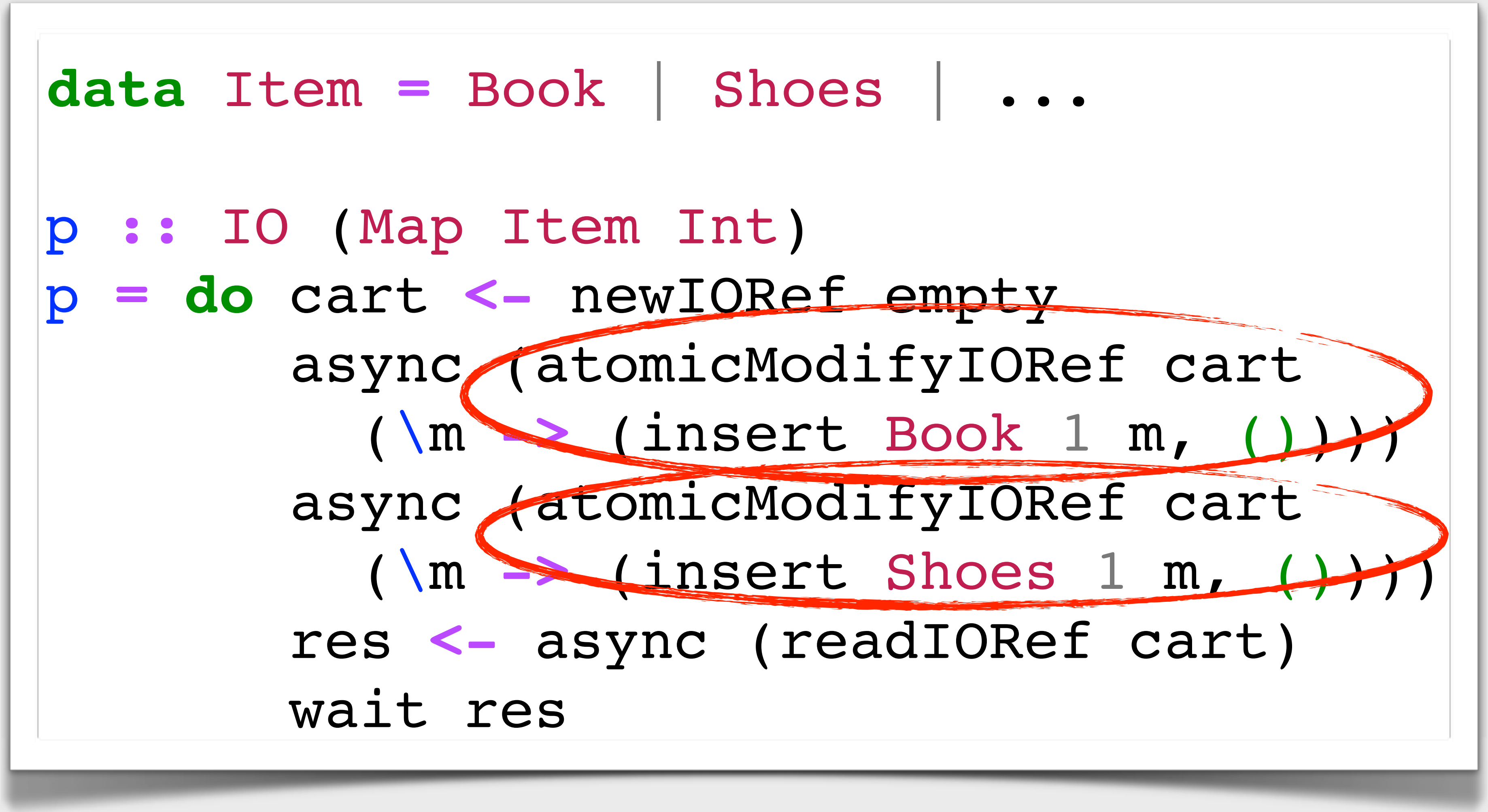
λ-calculus
(& array langs, ...)

```
f (g x)
  (h y)
```

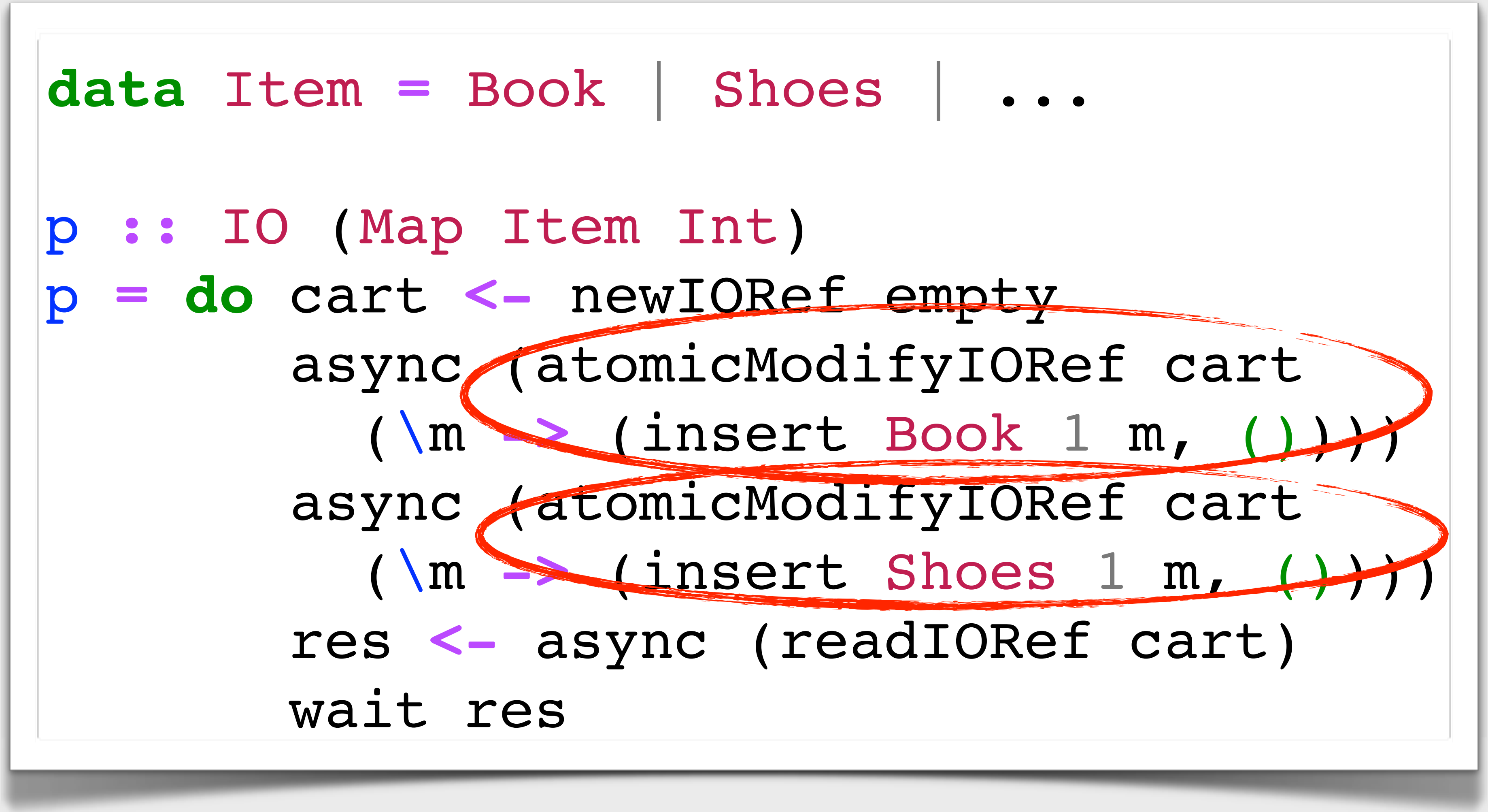
Can we *generalize* and *unify* these points on the map? Yes!



```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



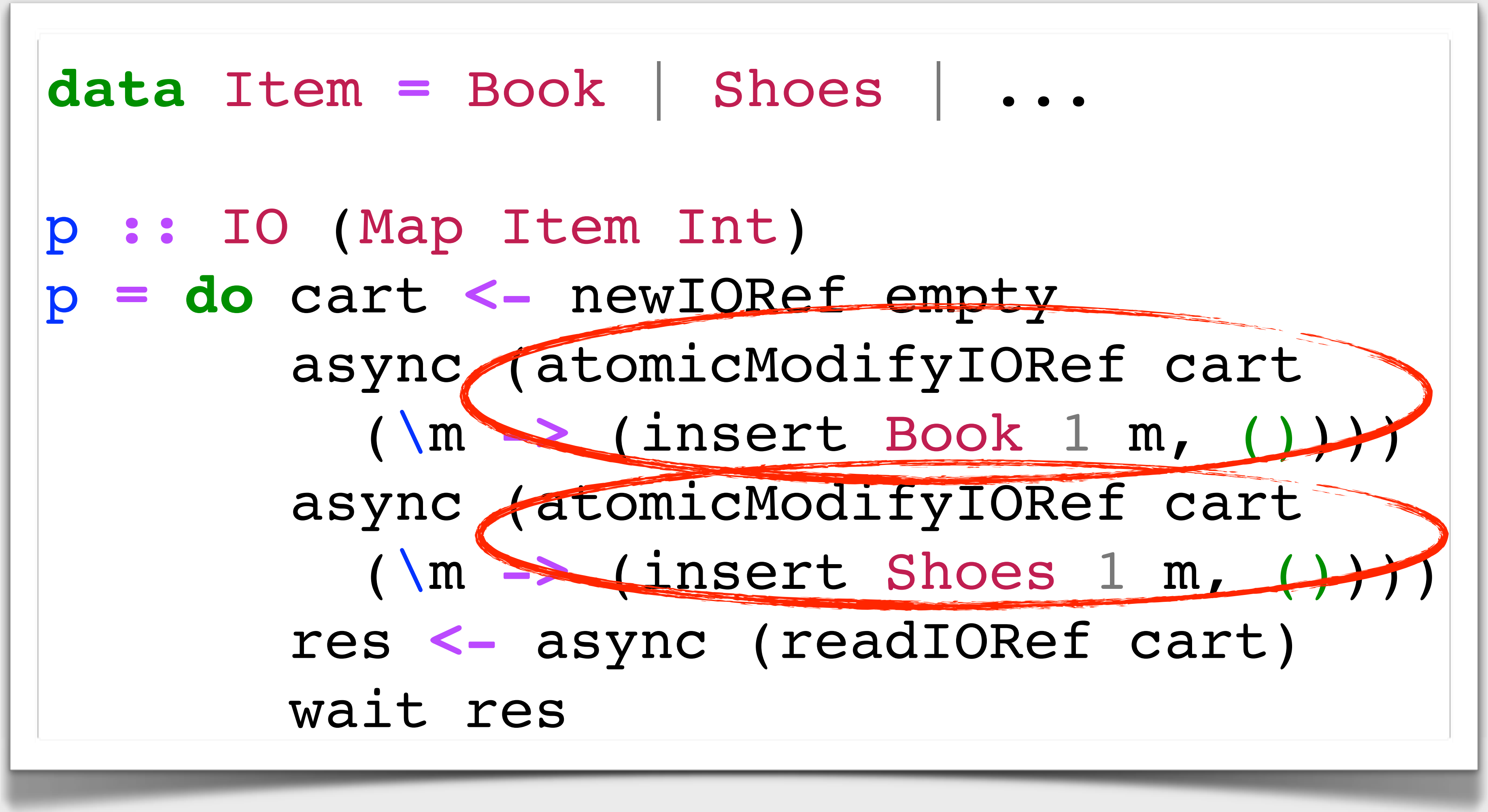
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



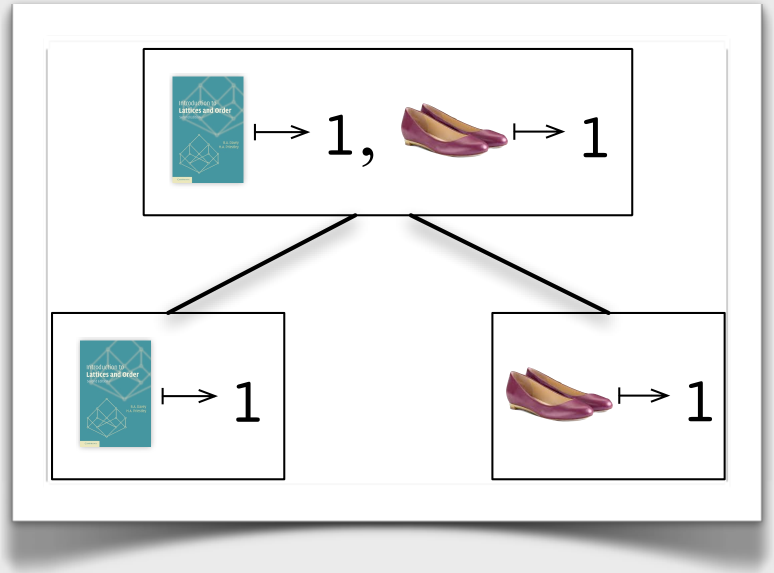
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]



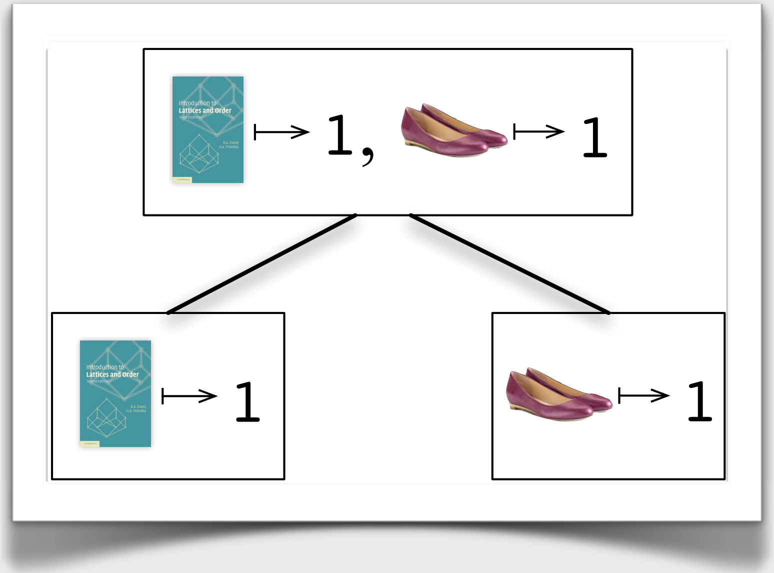
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

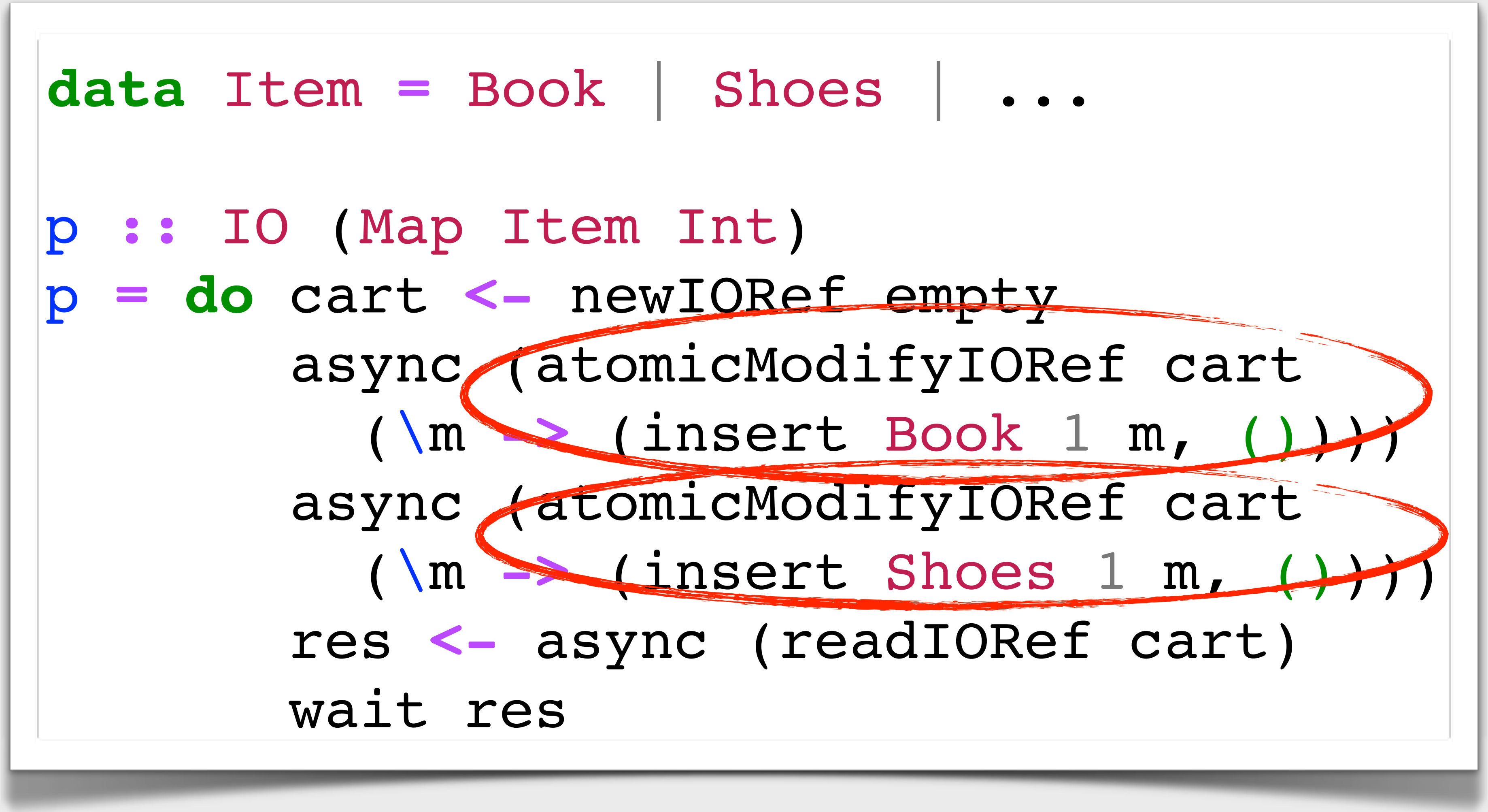


IVars: single writes, blocking (but exact) reads

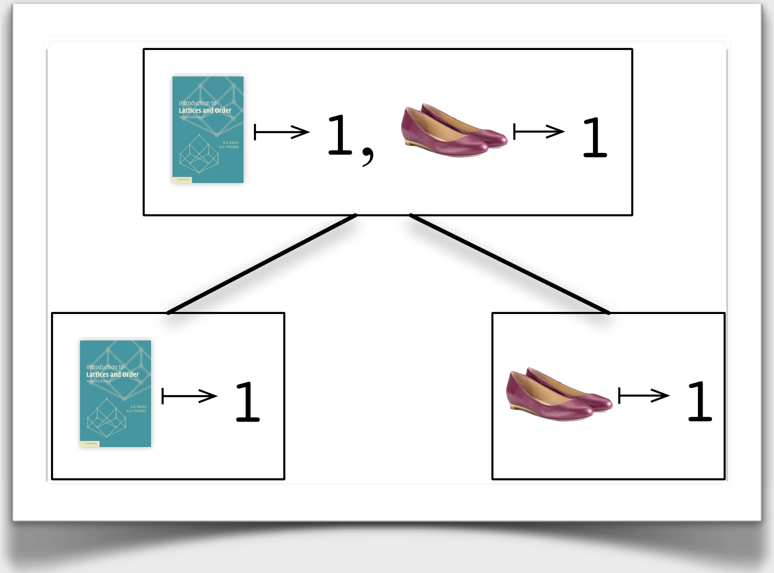
[Arvind *et al.*, 1989]

LVars: multiple *least-upper-bound* writes, blocking *threshold* reads

[FHPC '13]



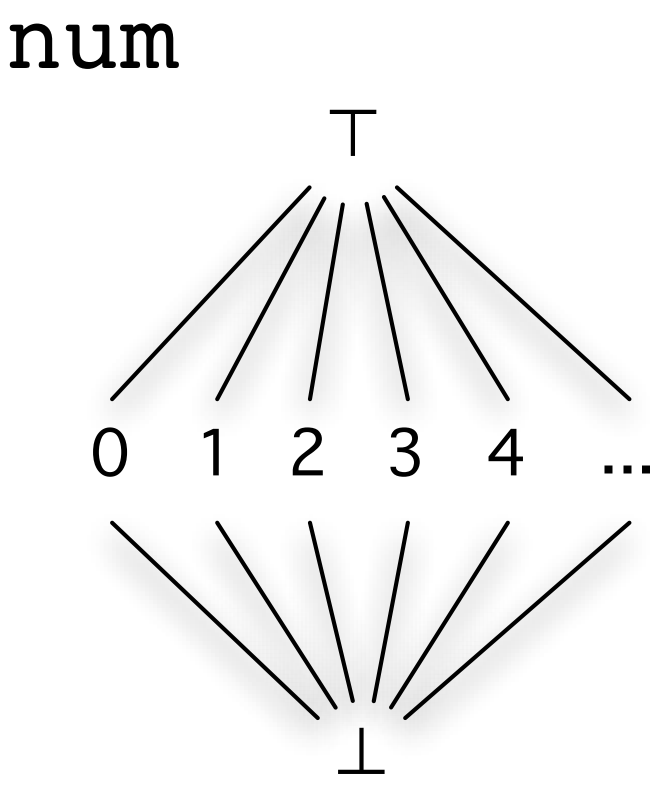
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```



IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *least-upper-bound* writes, blocking *threshold* reads
[FHPC '13]

* actually a bounded join-semilattice

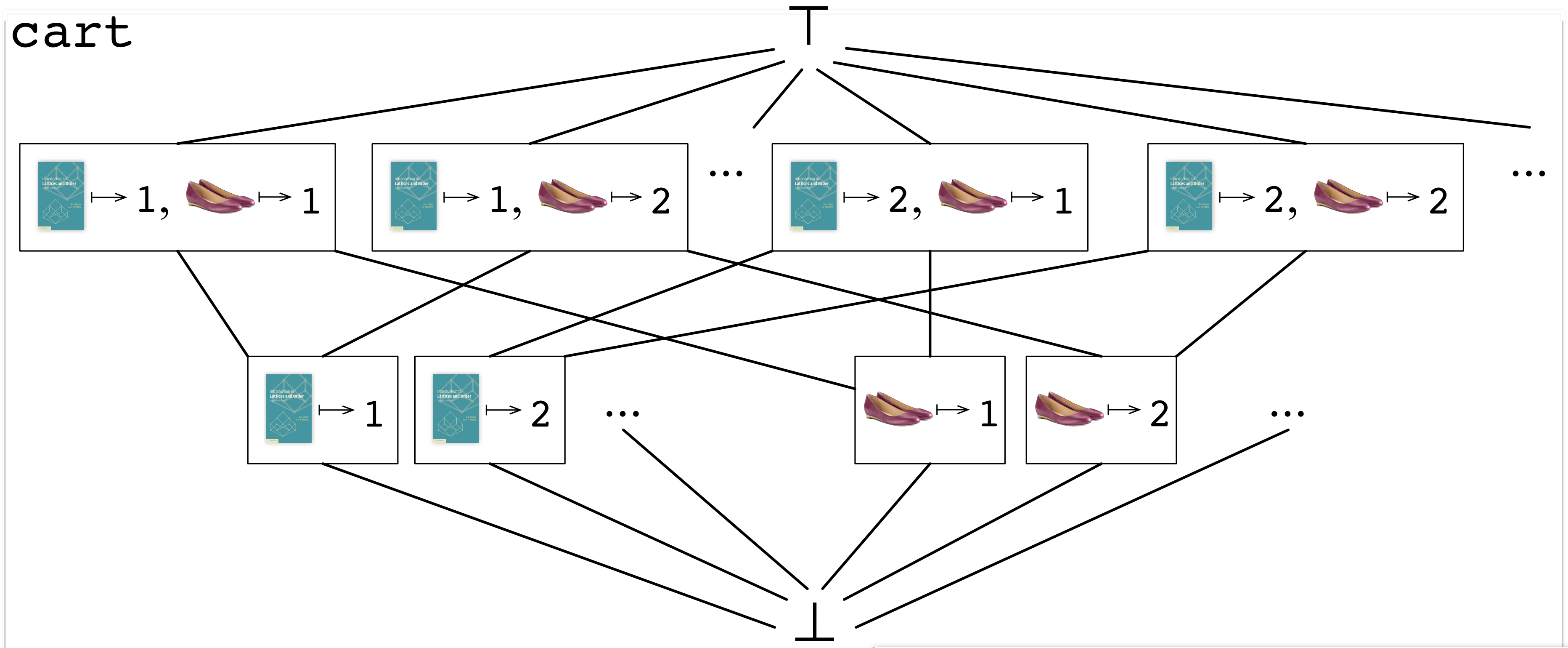


Raises an error, since 3 ⊔ 4 = ⊤

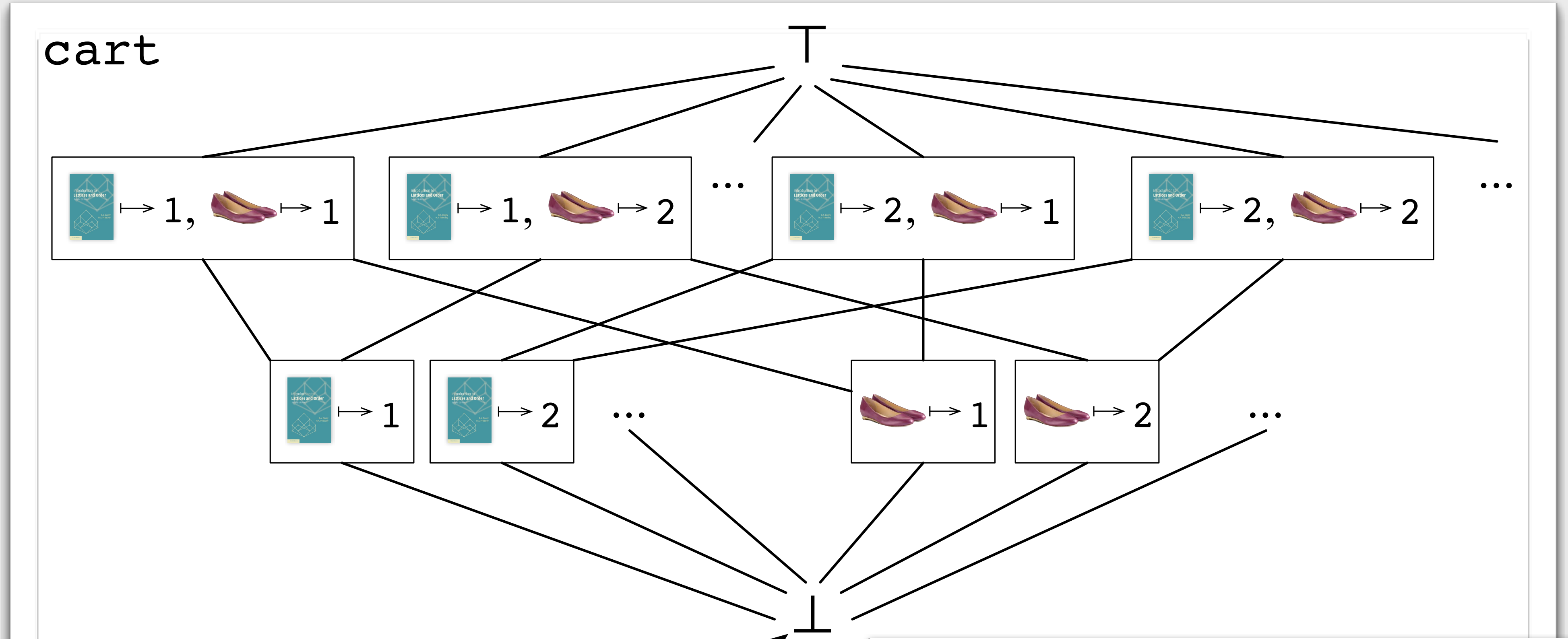
```
do
  fork (put num 3)
  fork (put num 4)
```

Works fine, since 4 ⊔ 4 = 4

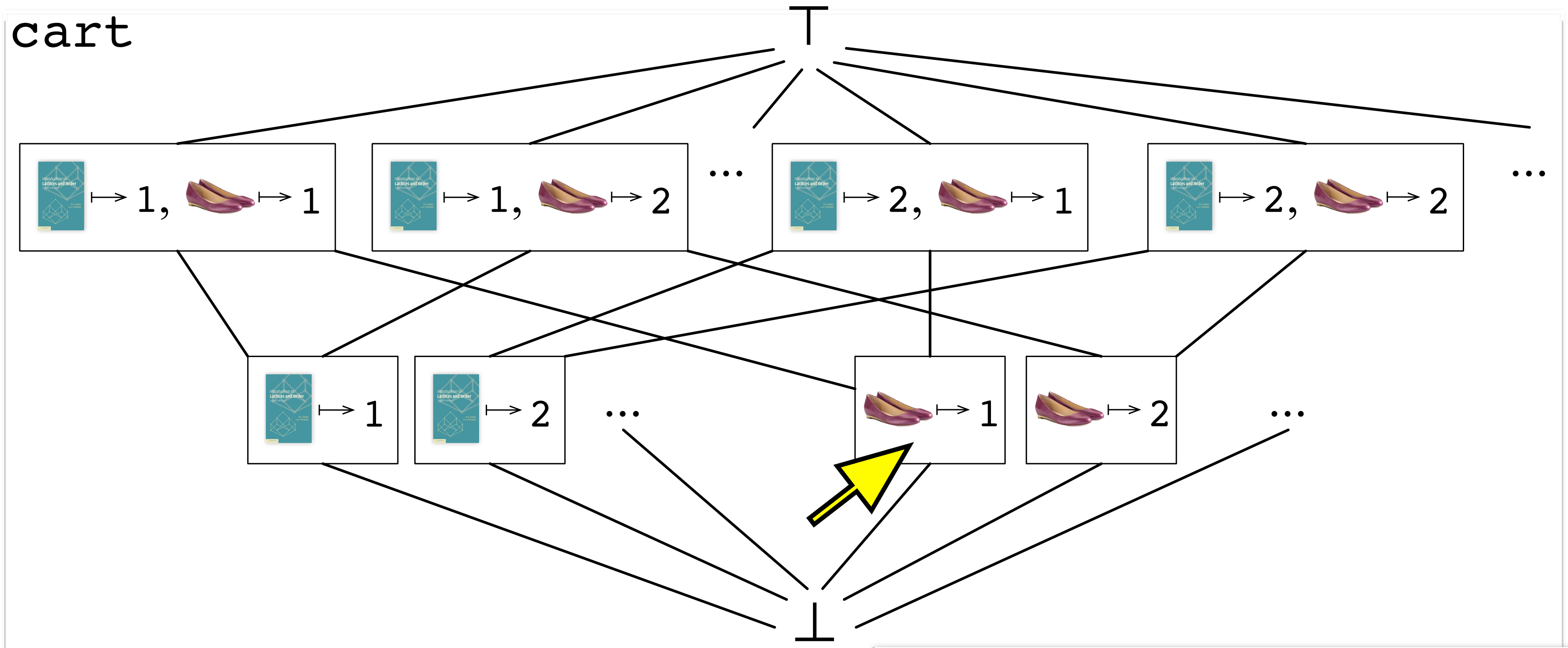
```
do
  fork (put num 4)
  fork (put num 4)
```



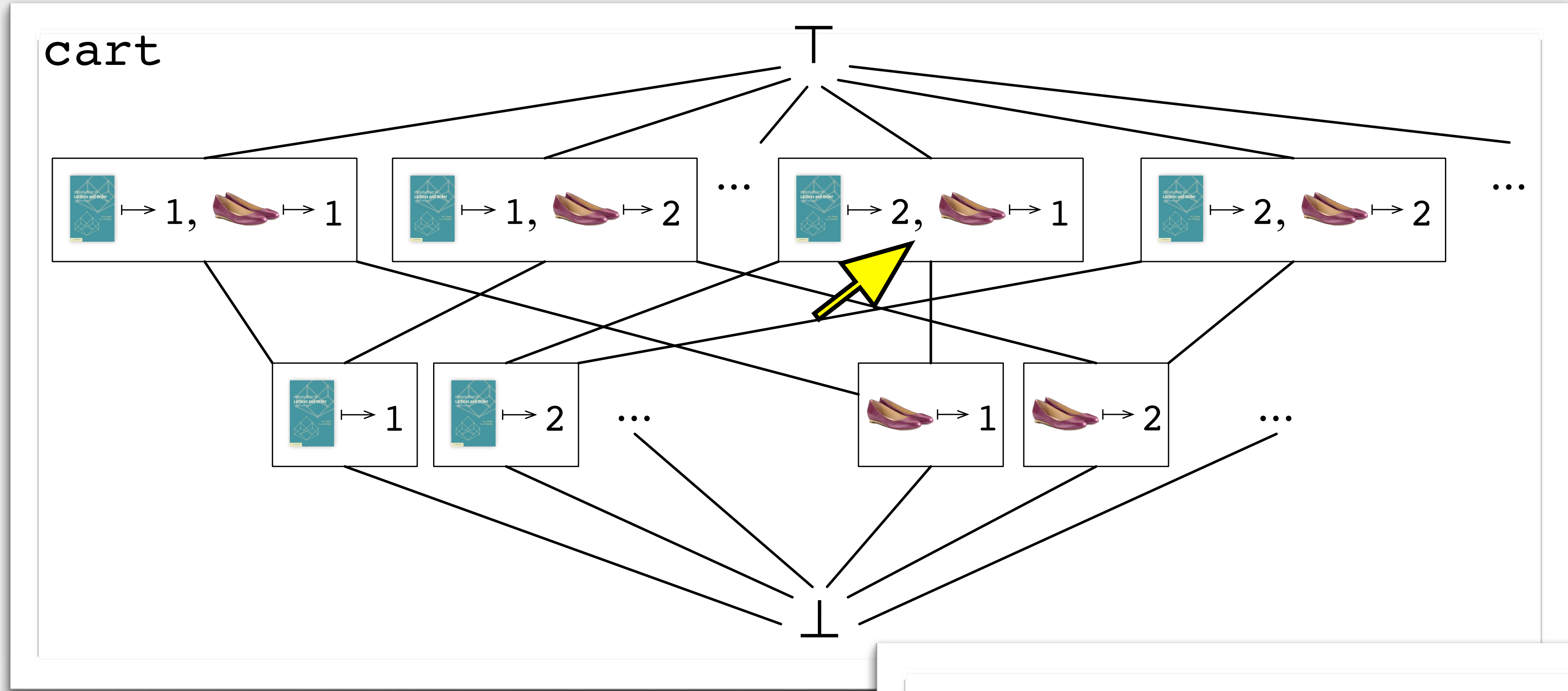
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



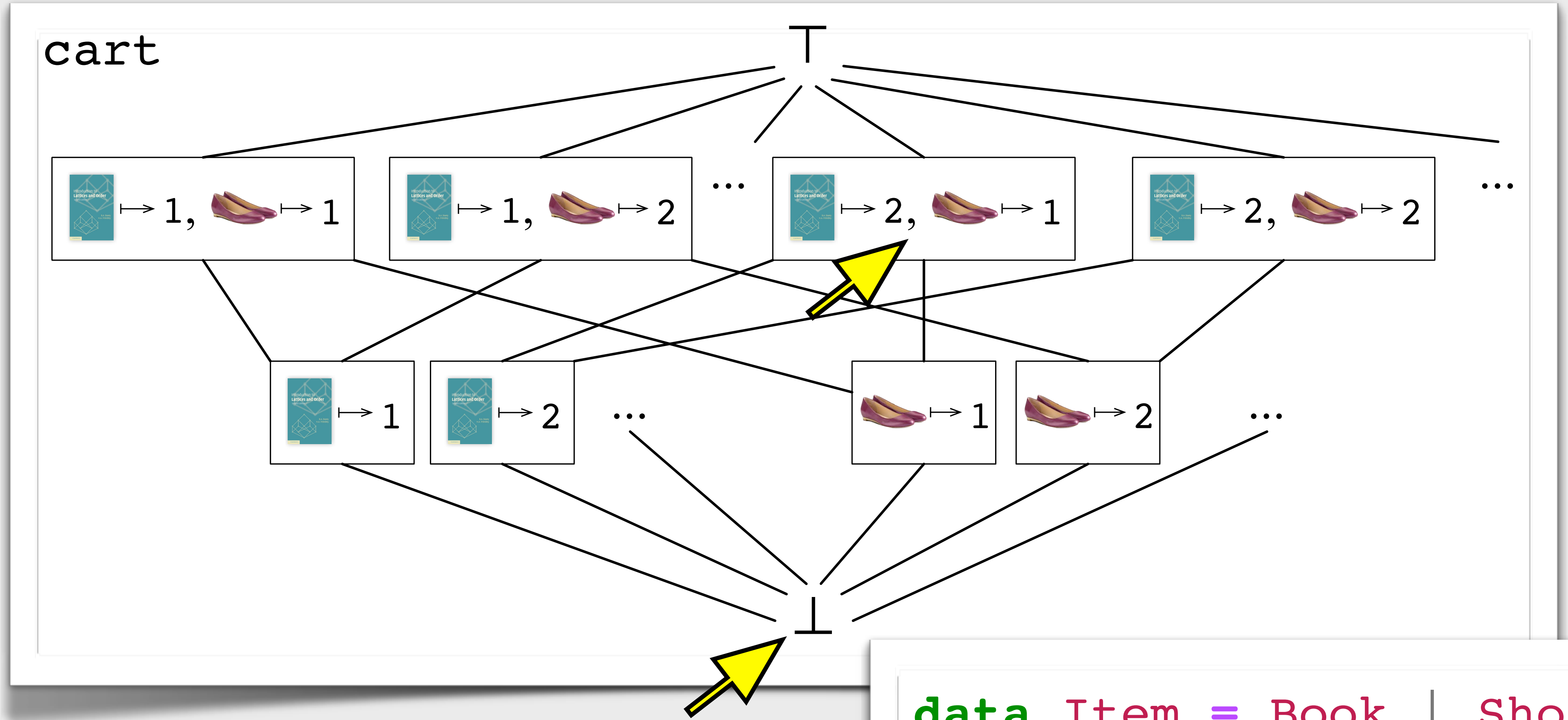
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



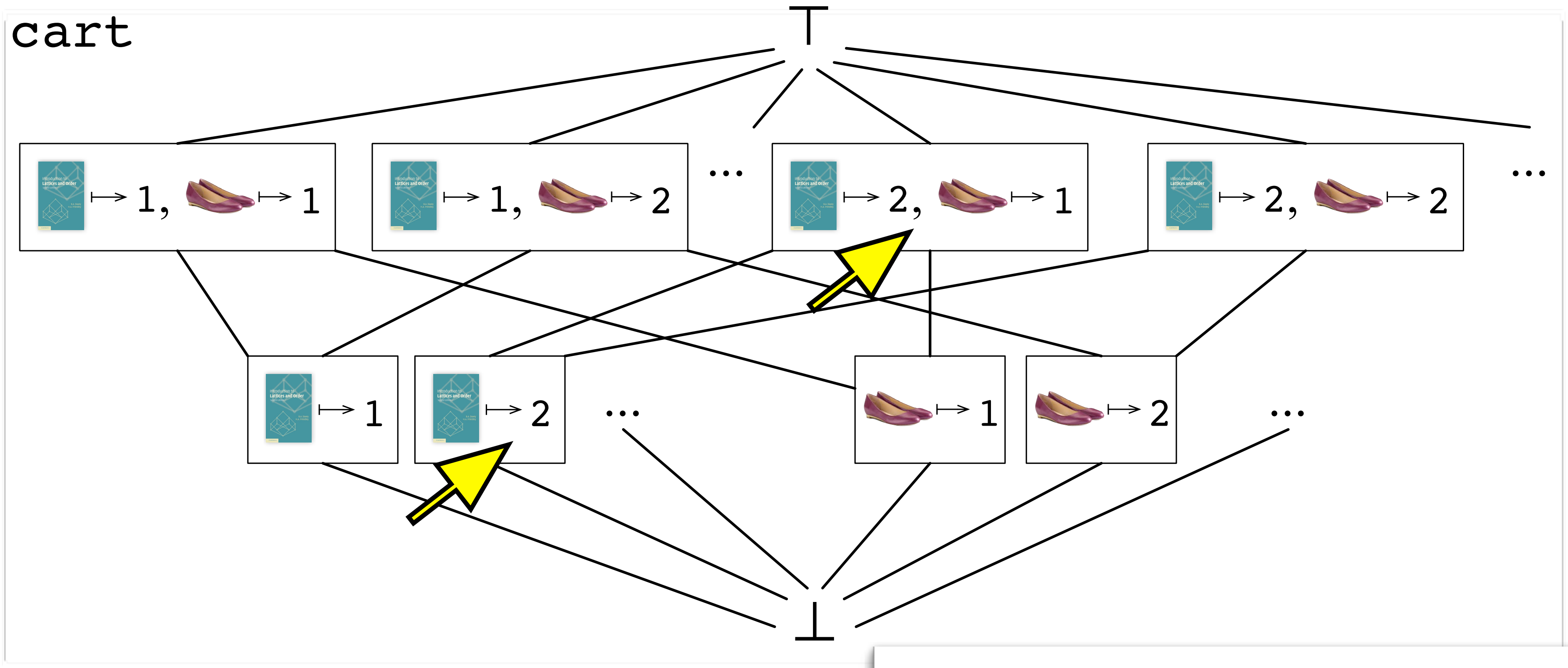
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



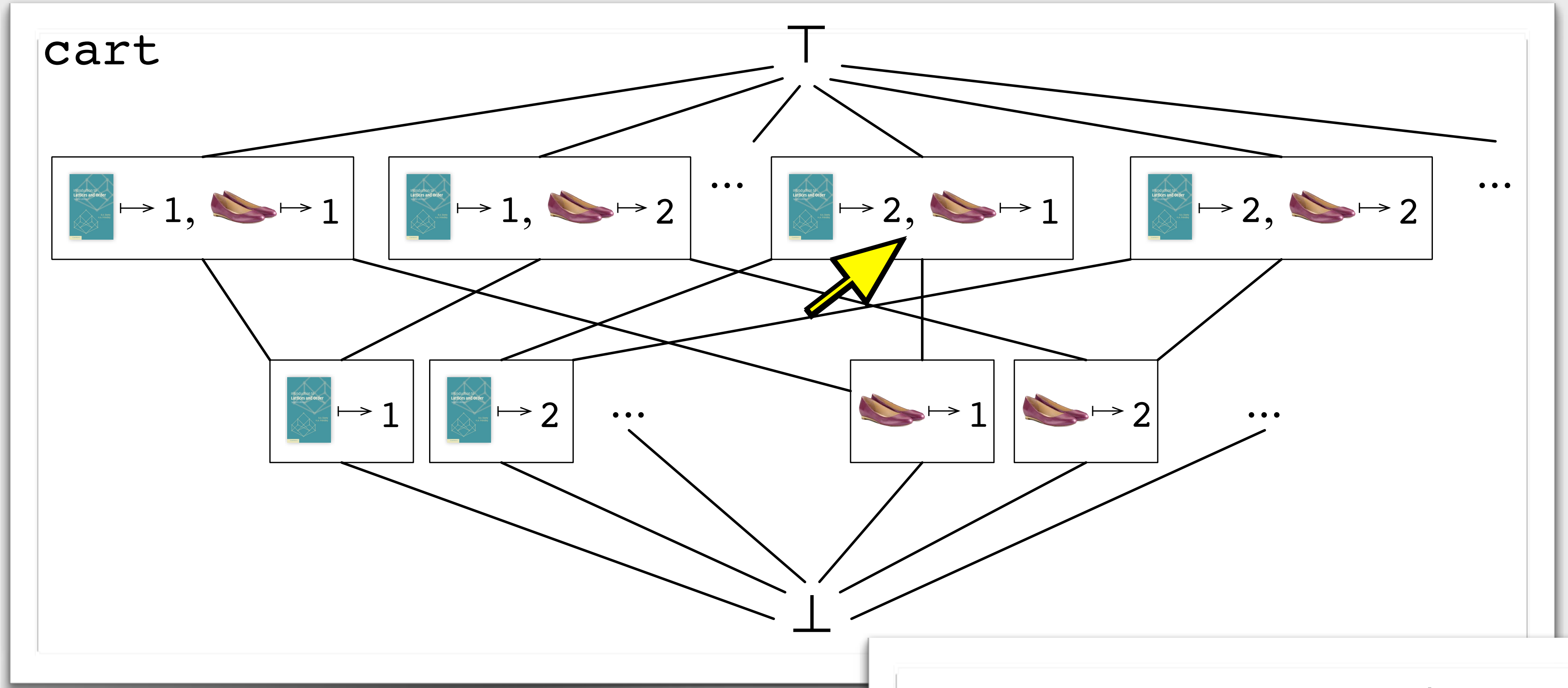
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



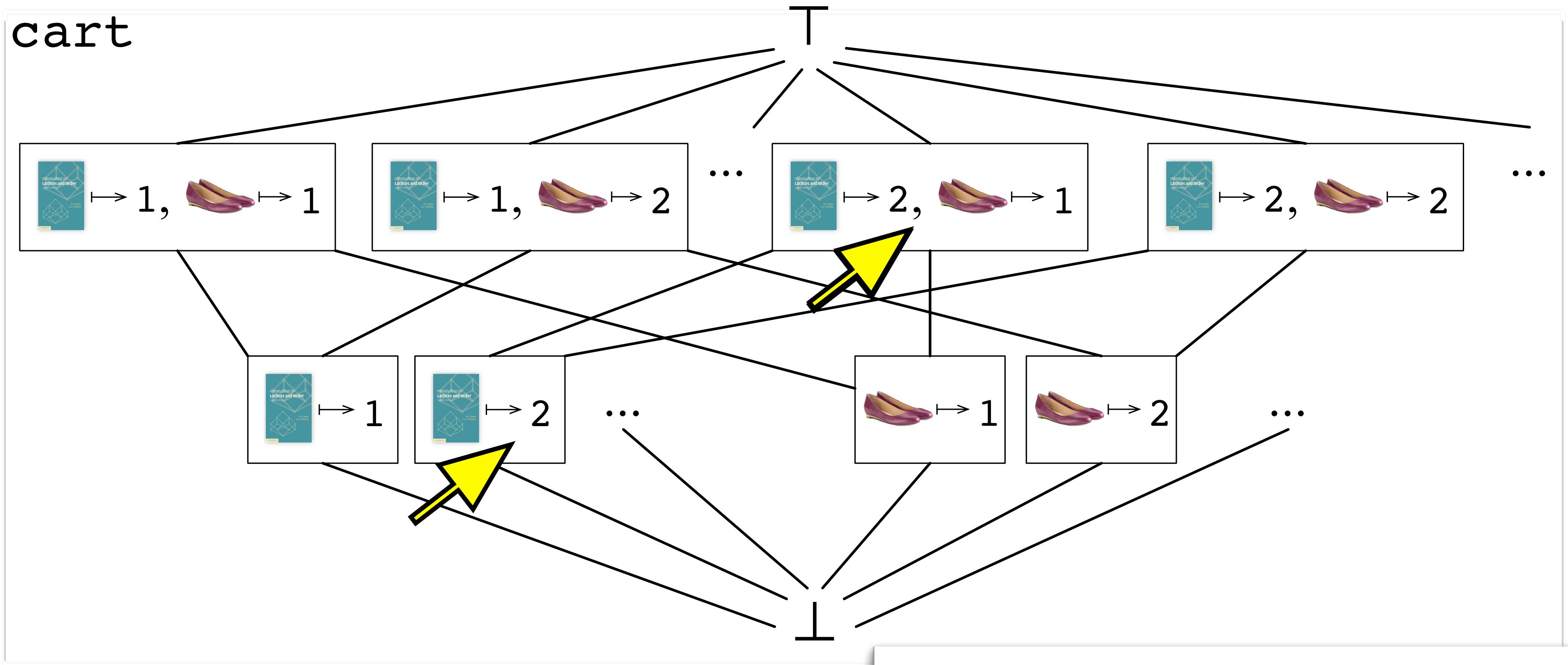
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



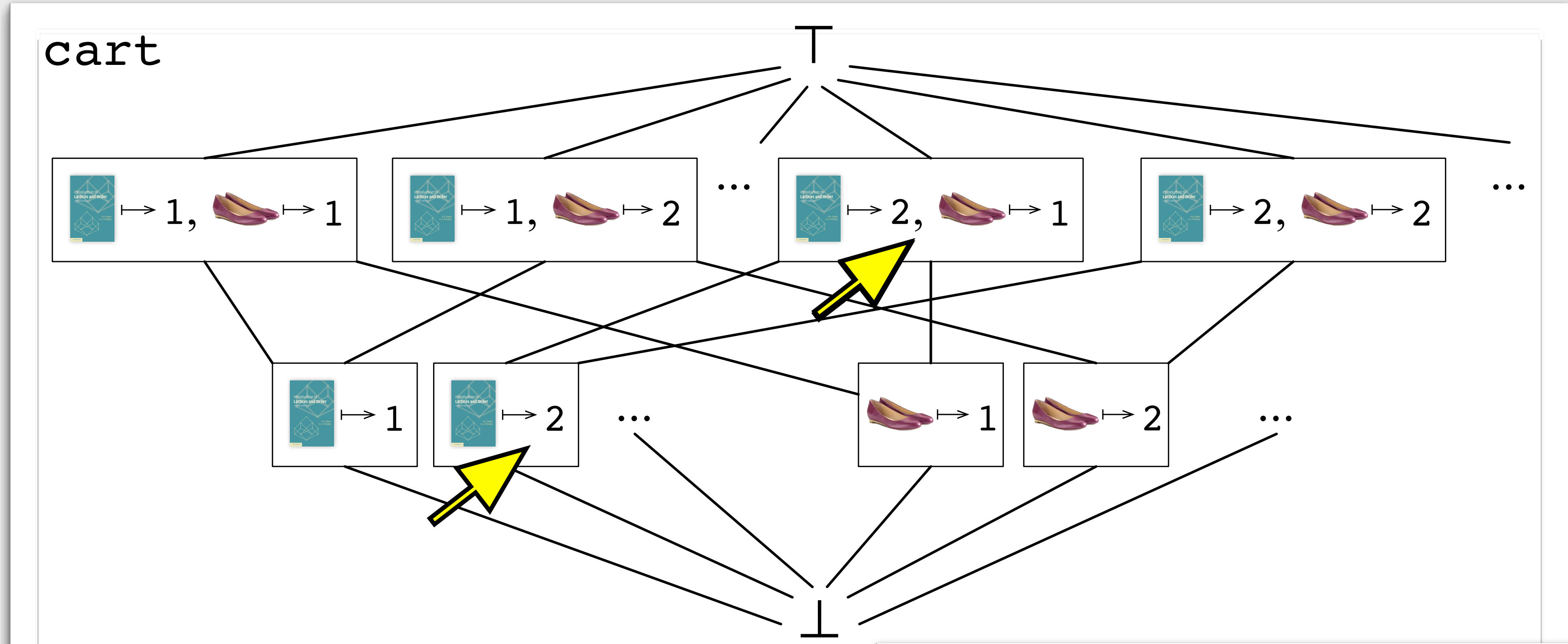
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

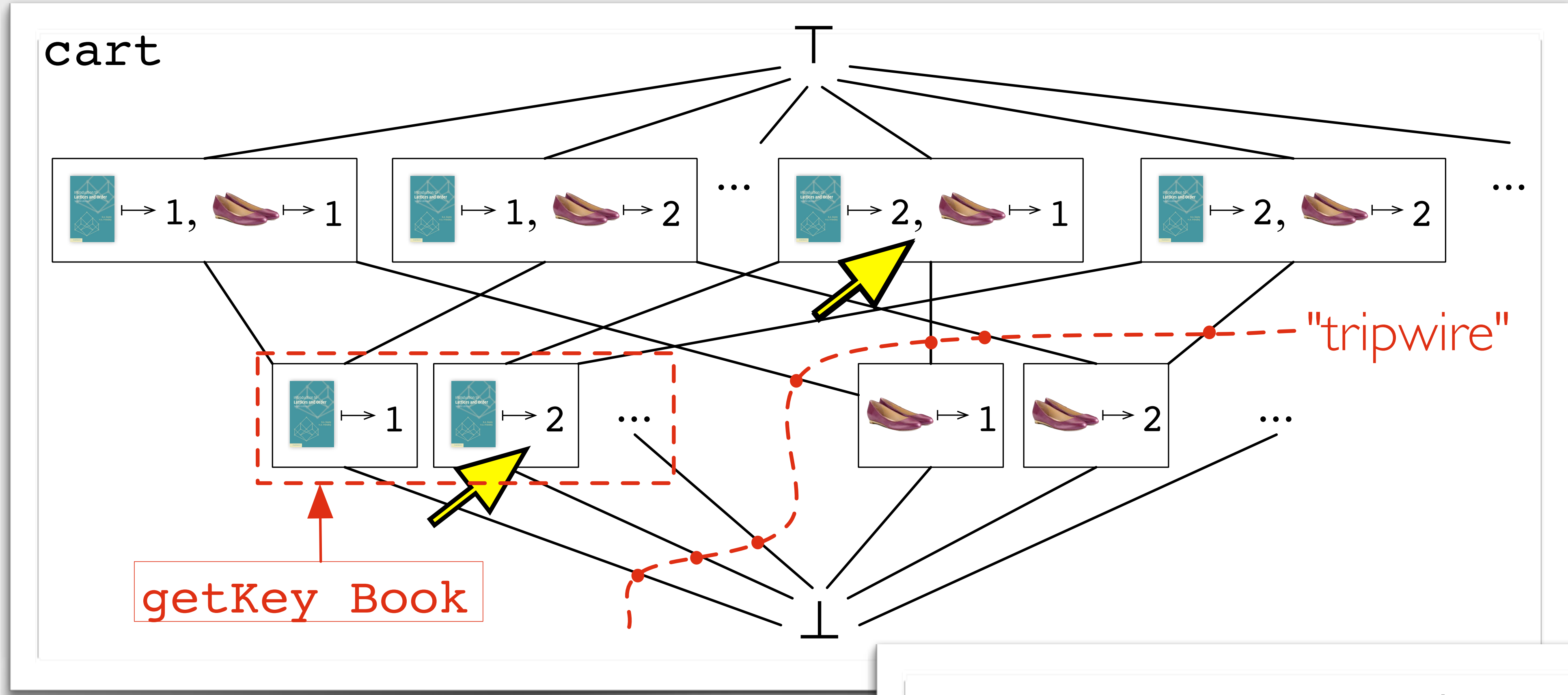


```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



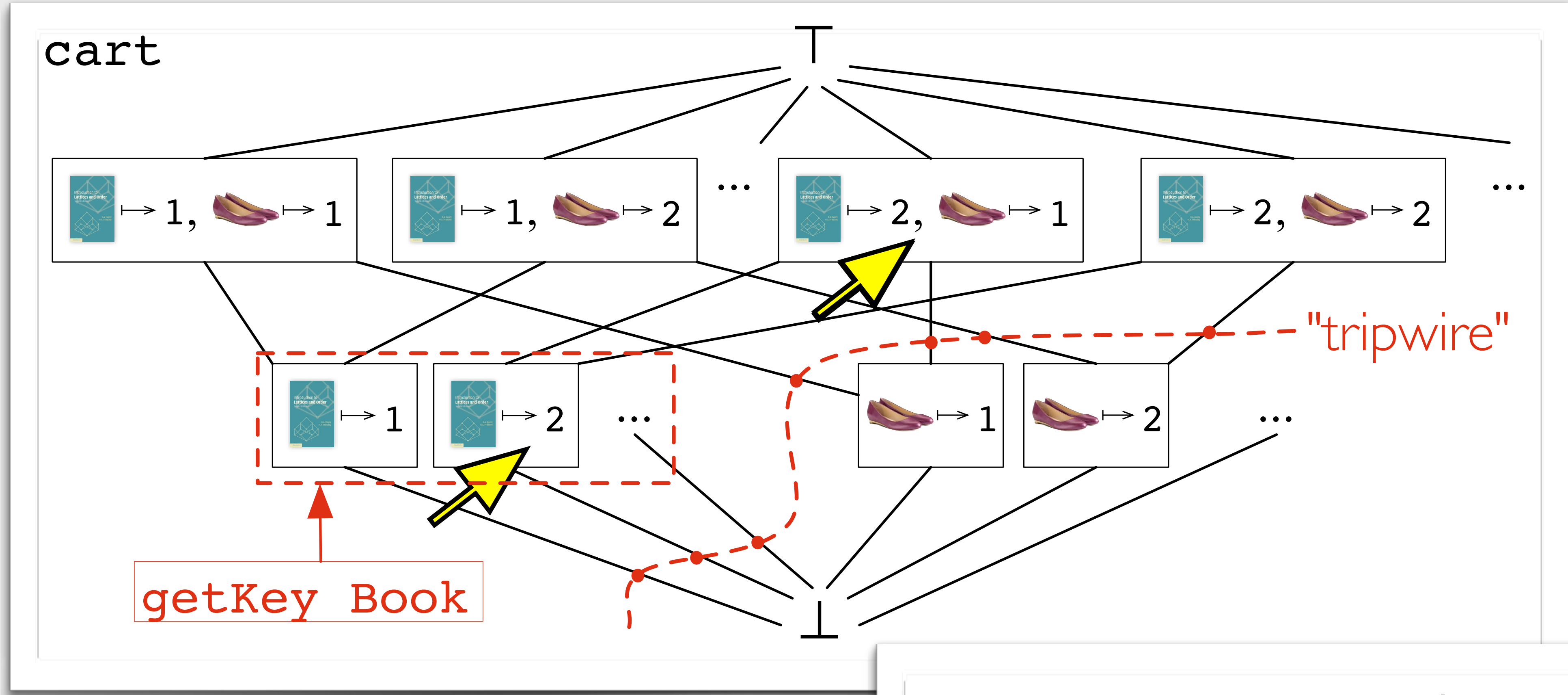
{(Book,1), (Book,2), …}

```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```



{(Book,1), (Book,2), …}

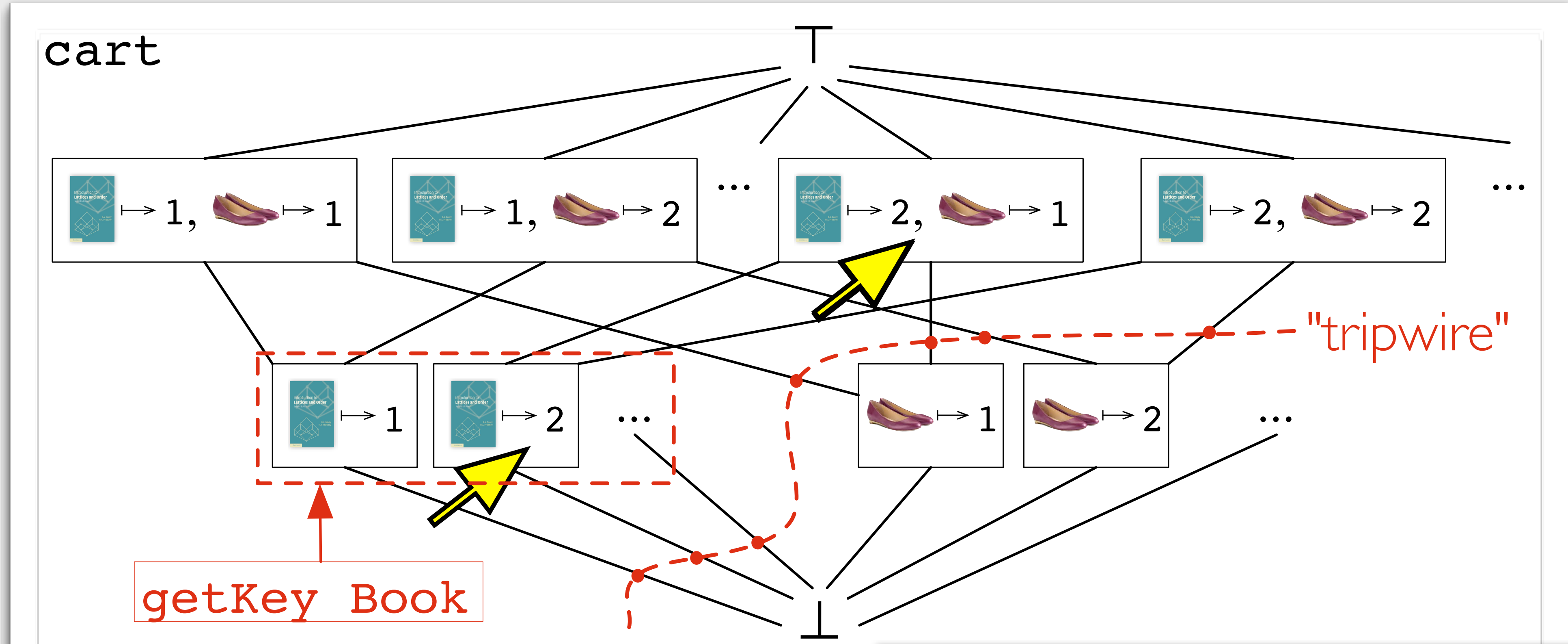
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

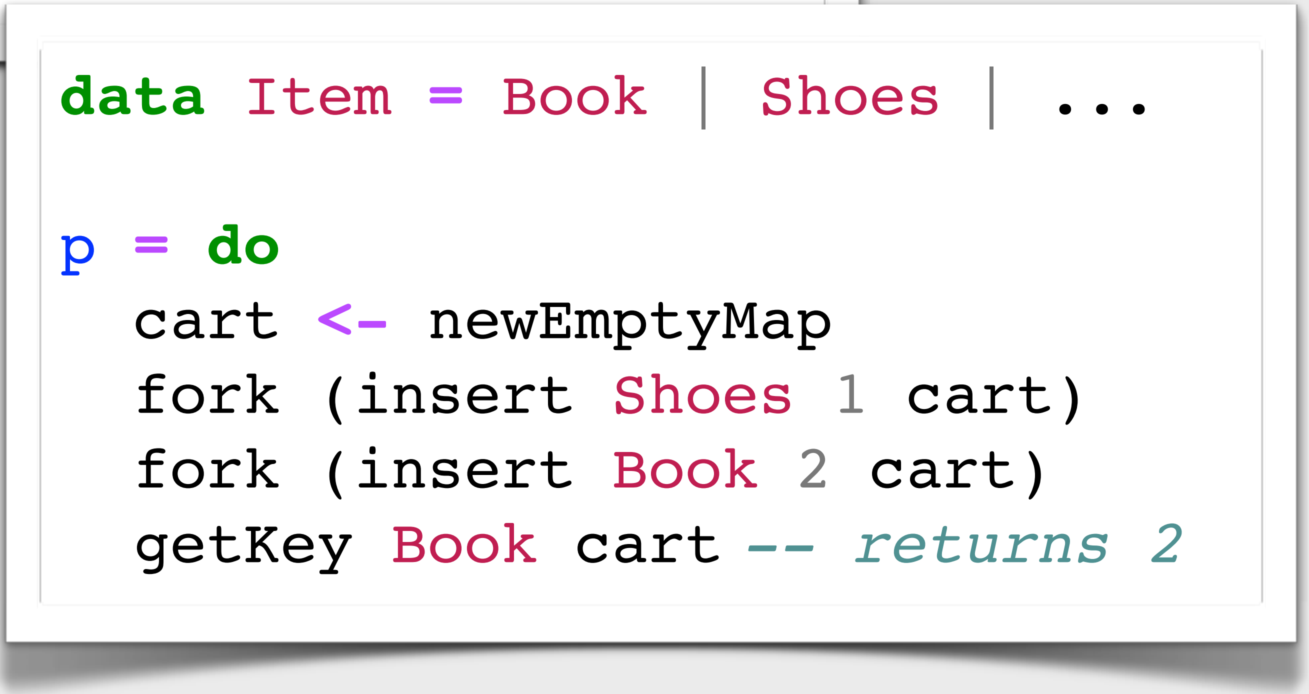


```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

{(Book,1), (Book,2), …}

The threshold set must be *pairwise incompatible*



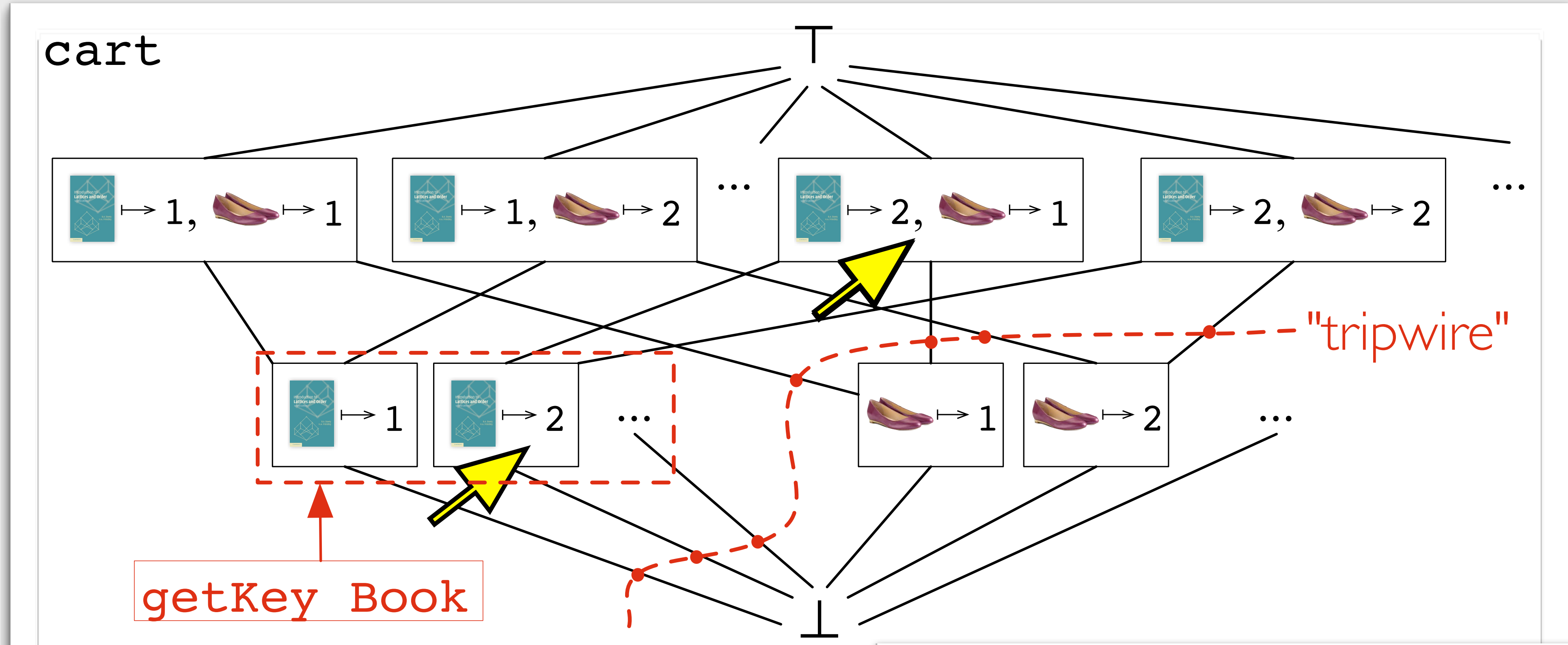


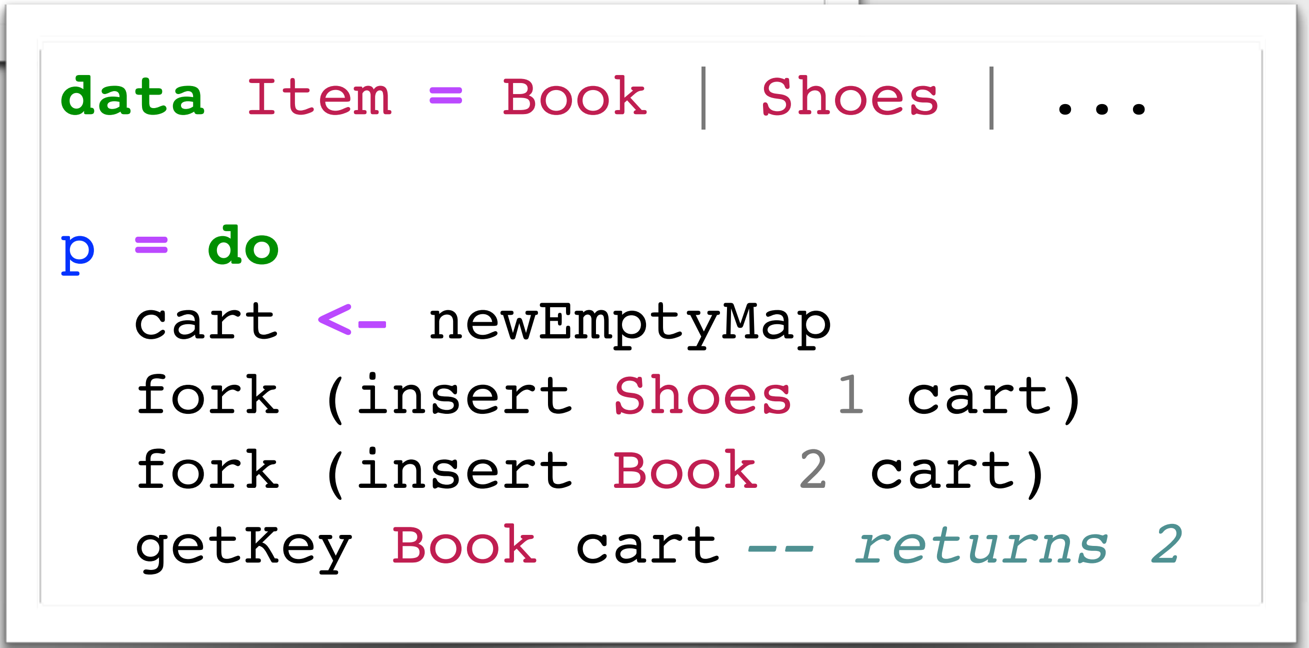
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

{(Book,1), (Book,2), …}

The threshold set must be *pairwise incompatible*

proof obligation





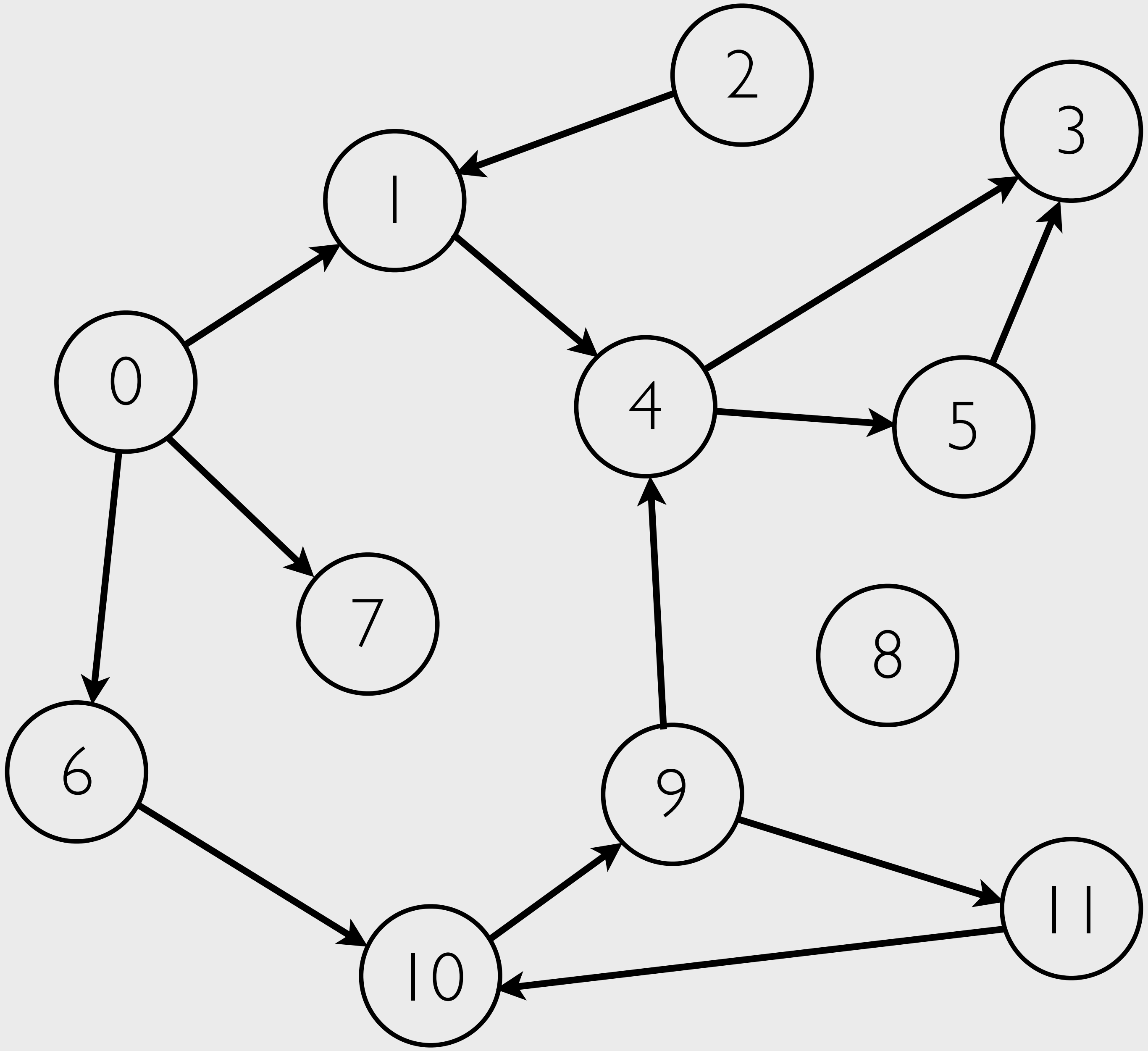
```
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

{(Book,1), (Book,2), …}

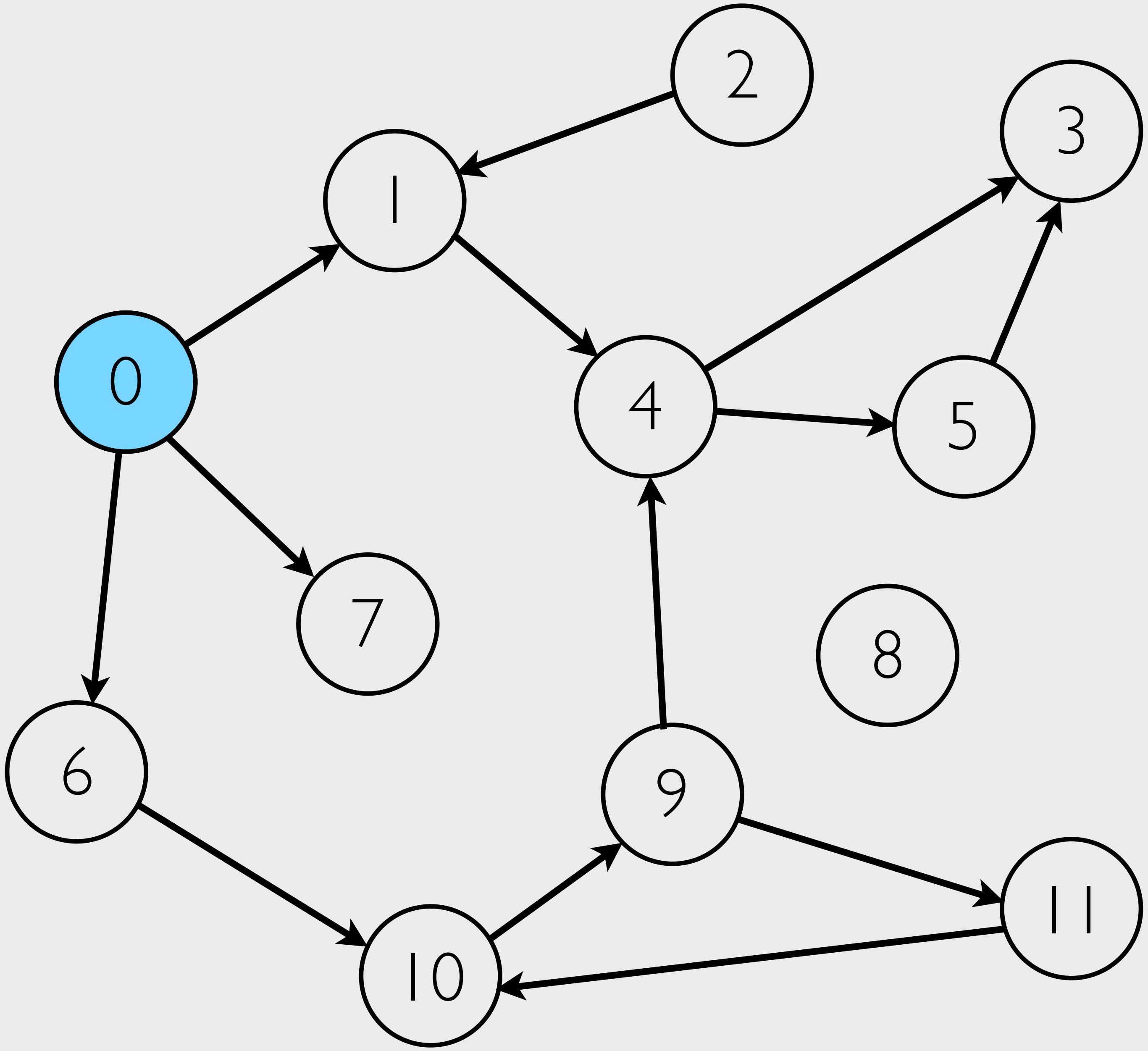
The threshold set must be *pairwise incompatible*

proof obligation

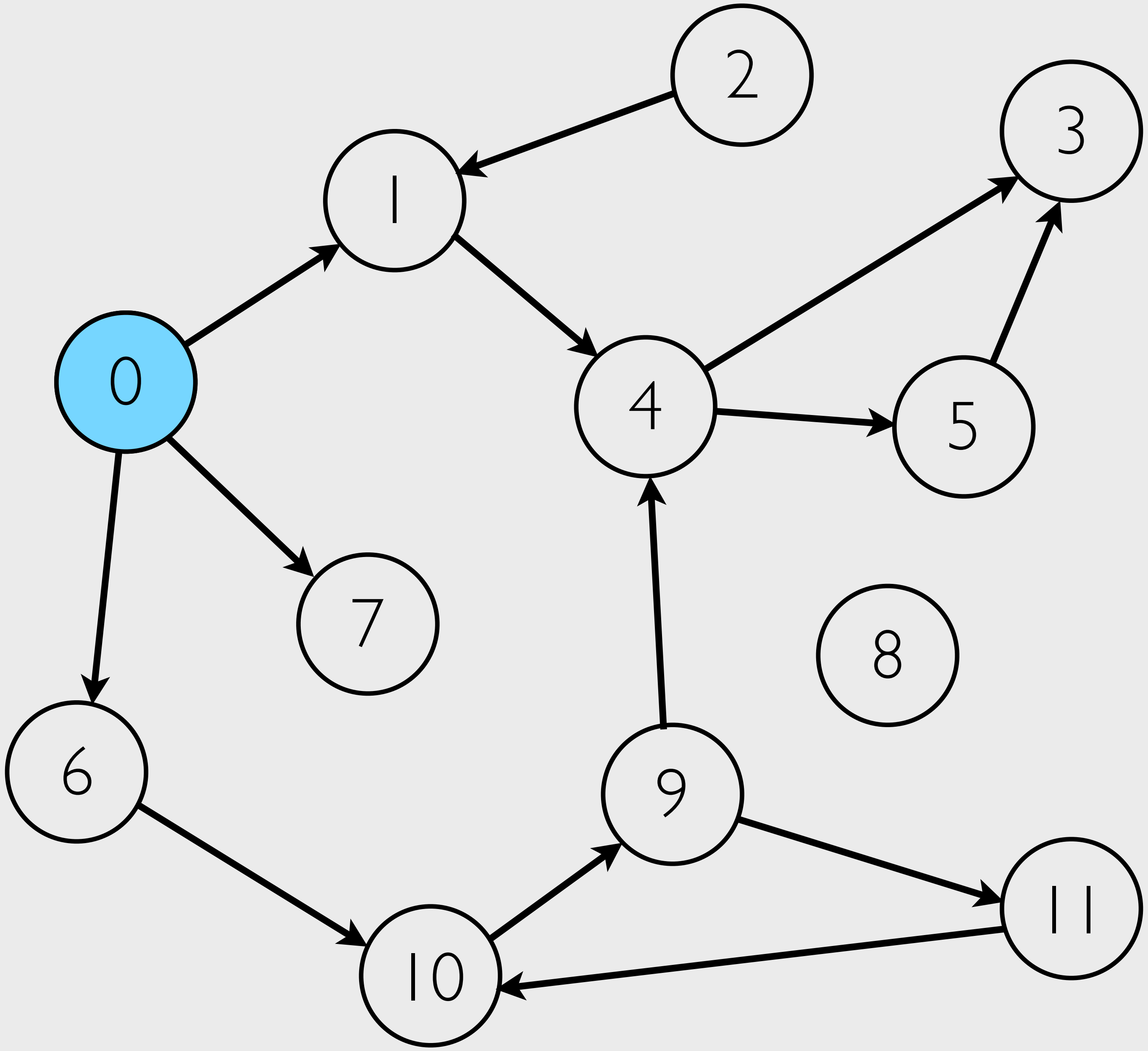
client guarantee

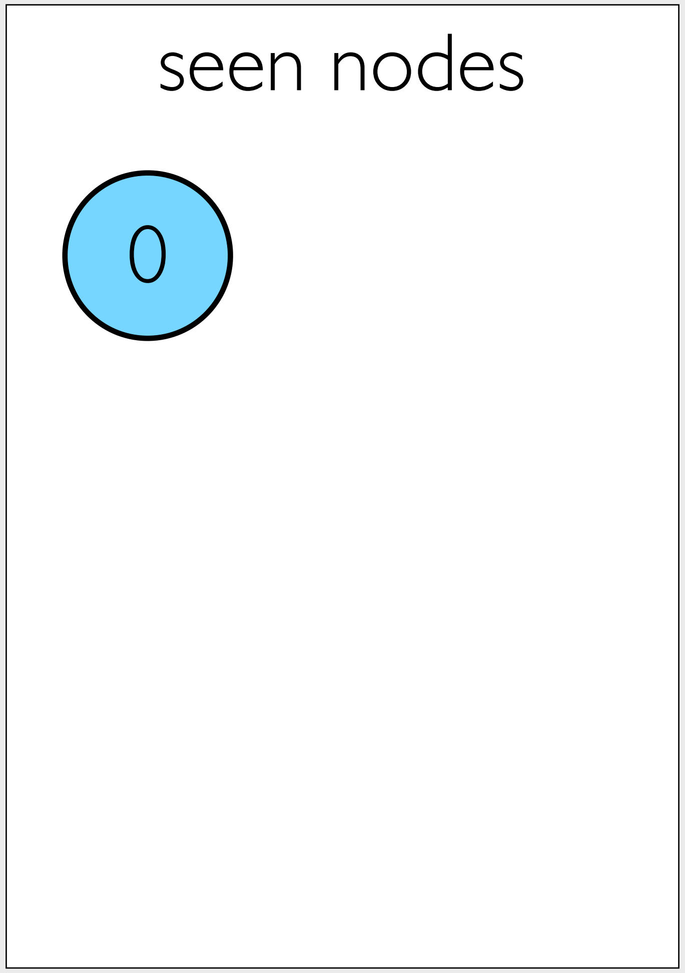


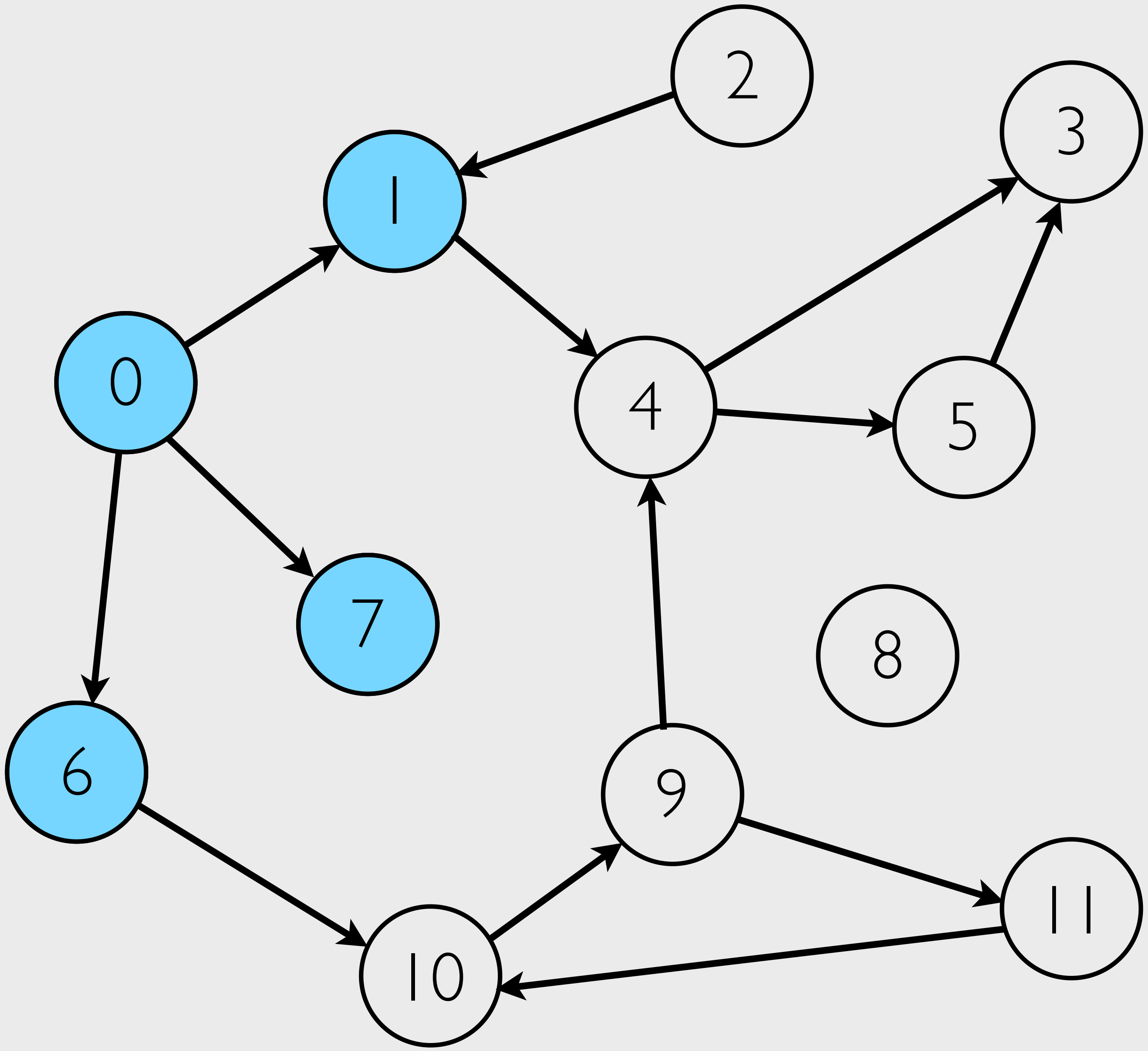
seen nodes

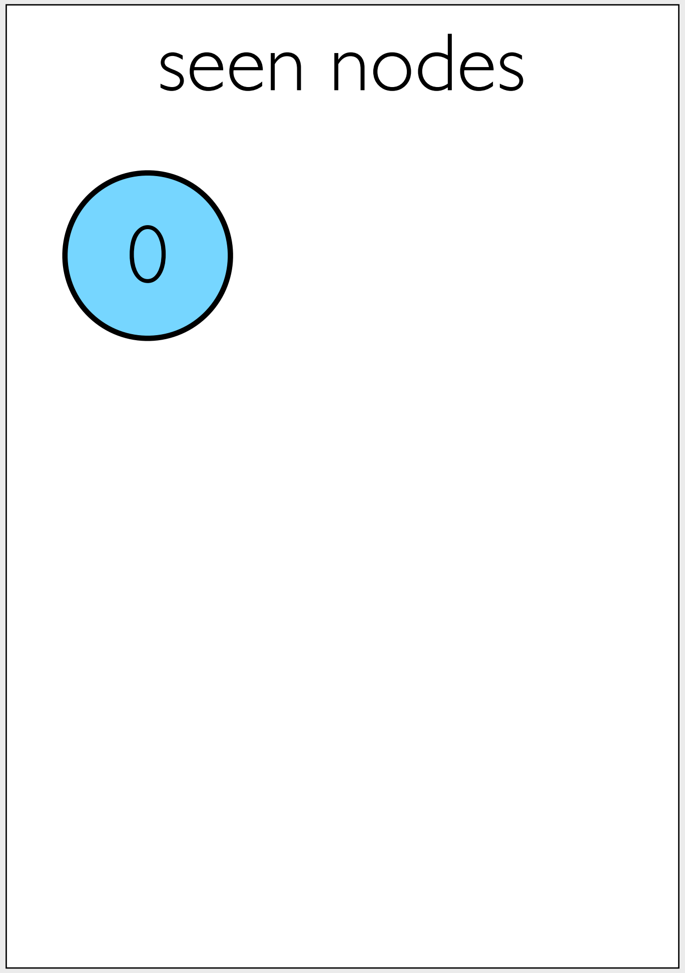


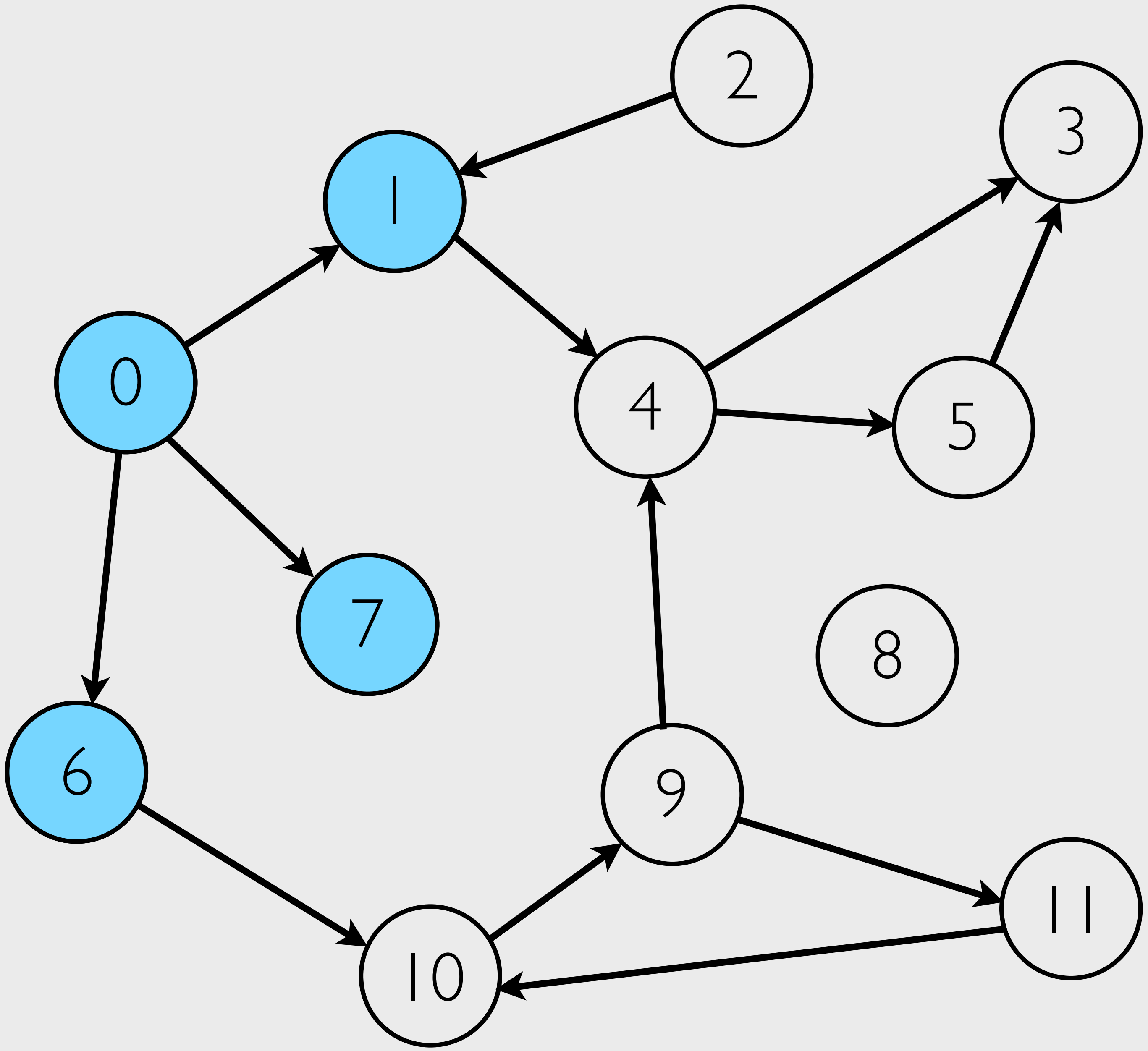
seen nodes

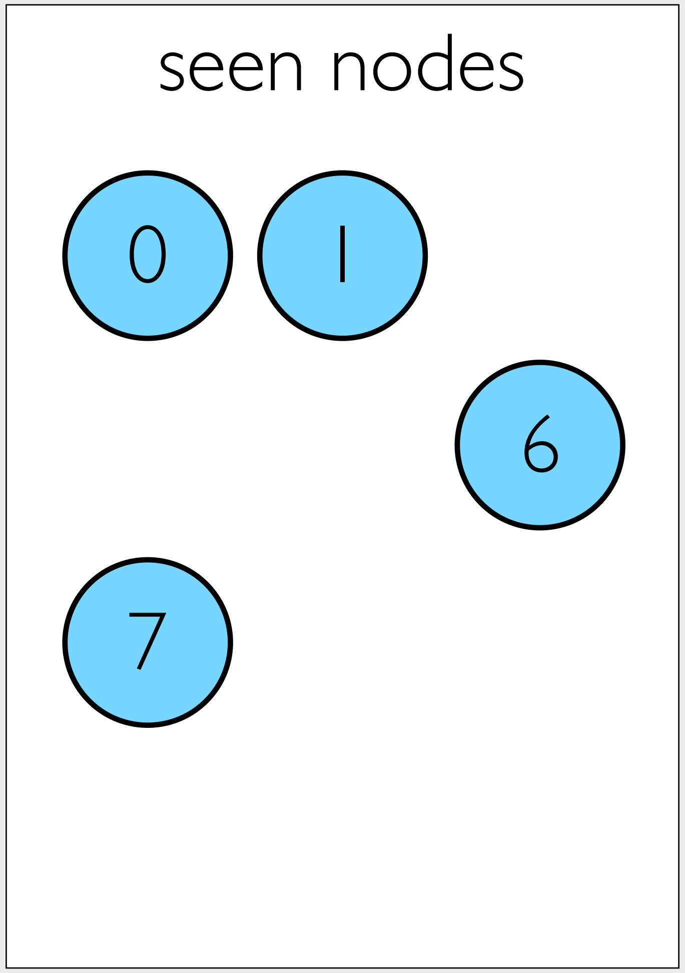

2
1
3
0
4
5
7
8
6
9
11
10

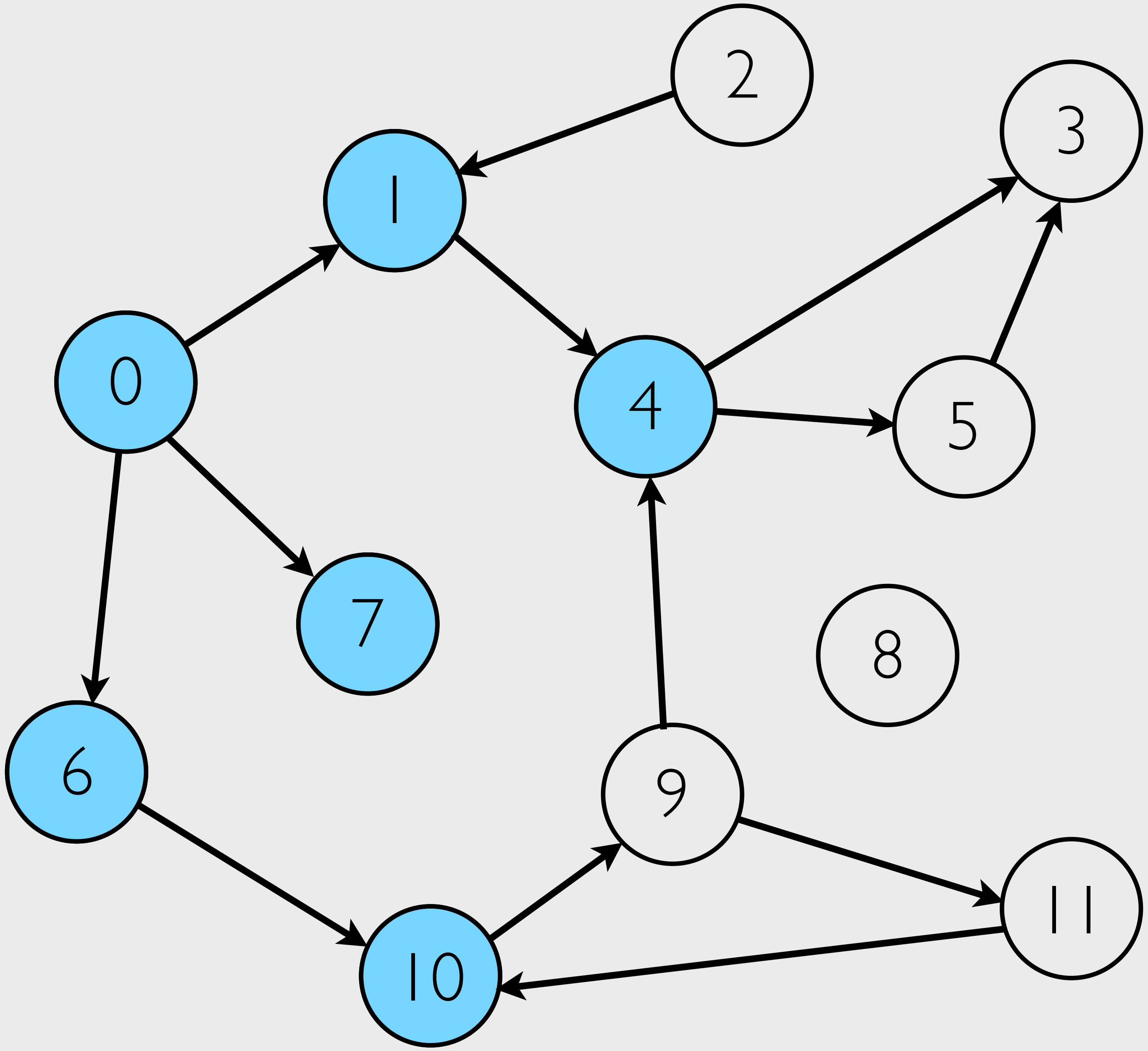

seen nodes
0

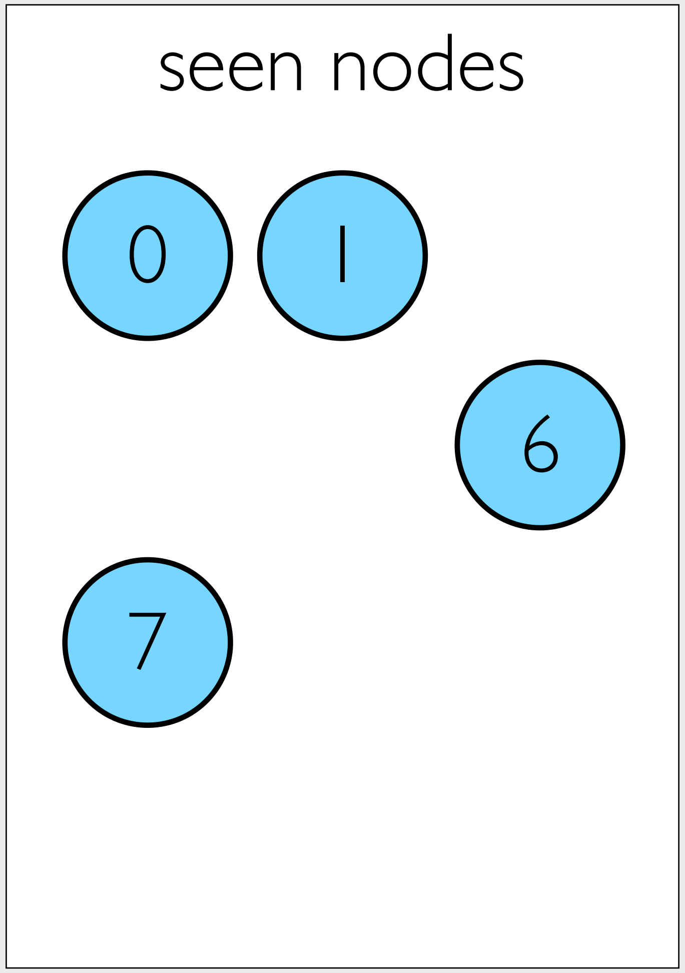

2
1
3
0
4
5
7
8
6
9
11
10

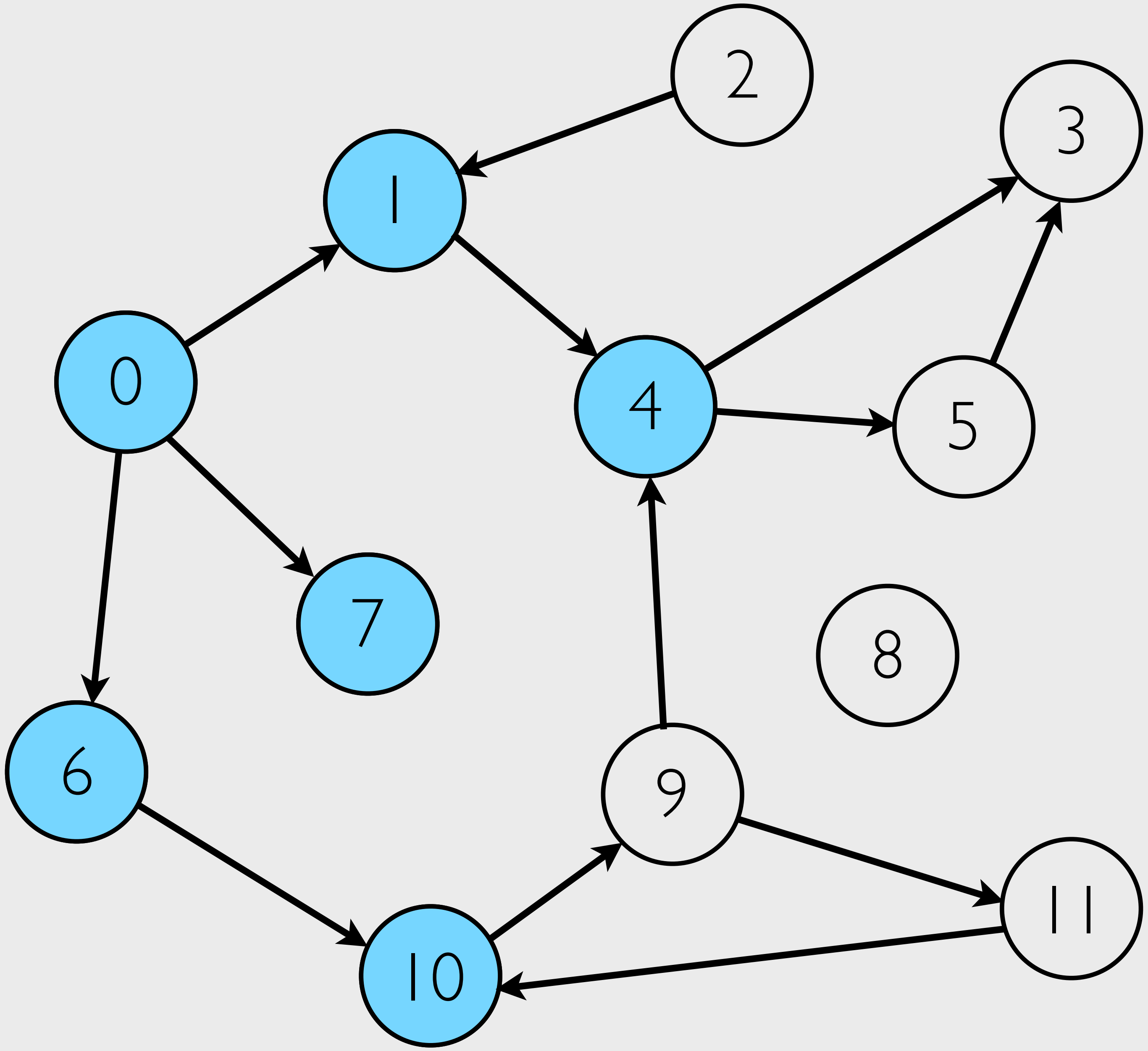

seen nodes
0

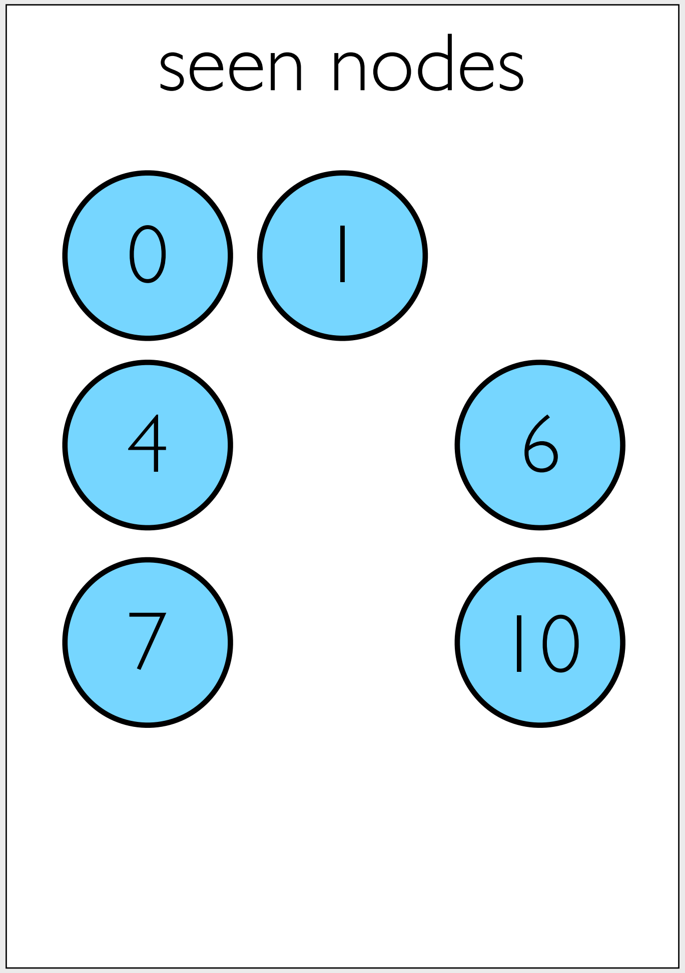

2
1
3
0
4
5
7
8
6
9
11
10

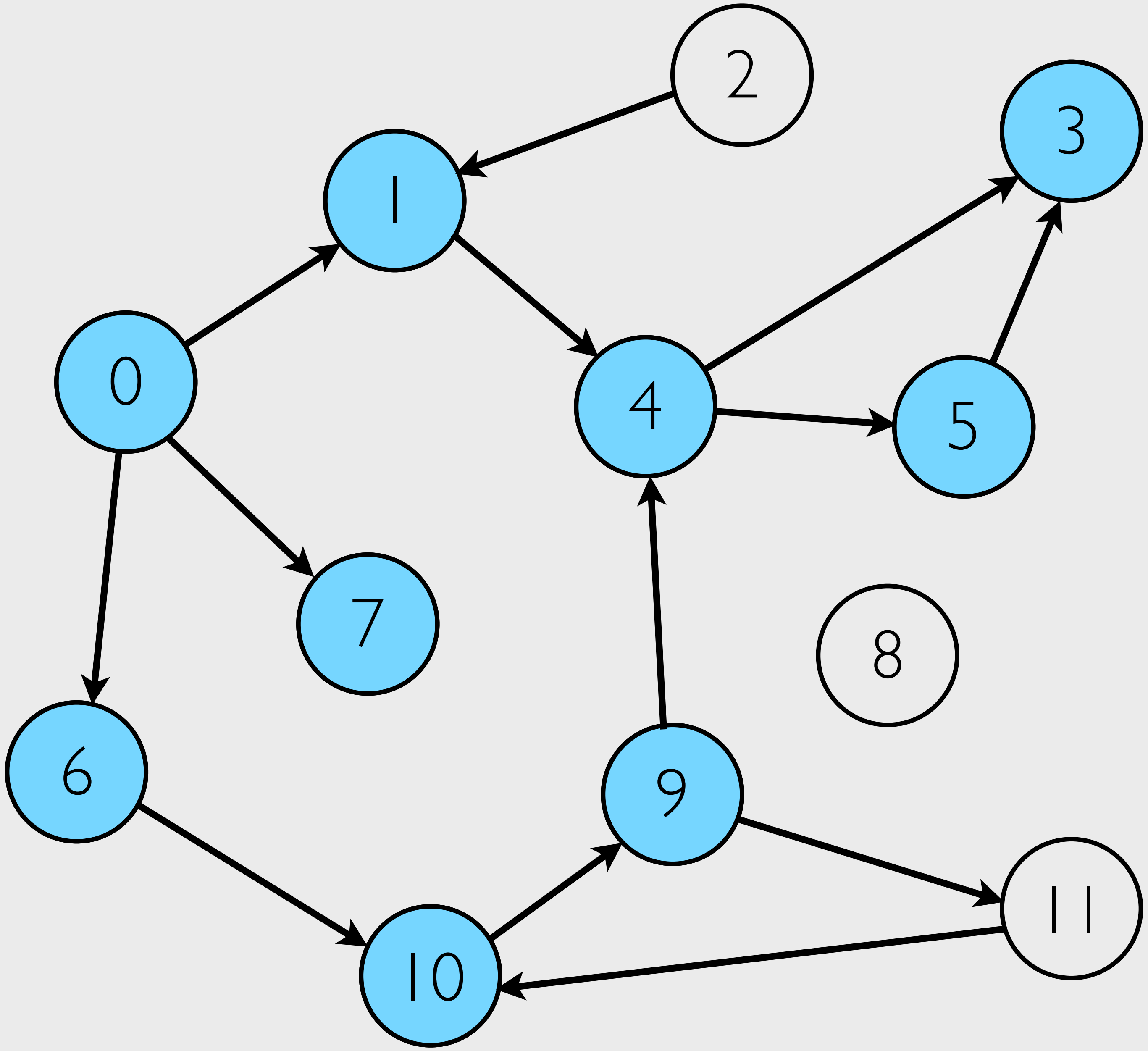

seen nodes
0
1
6
7



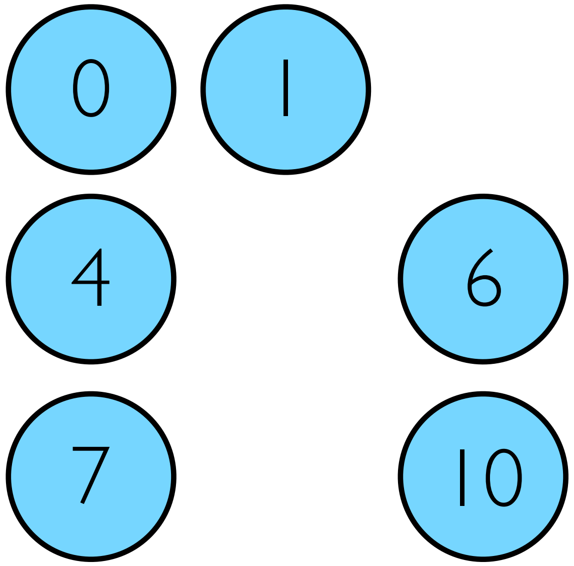

seen nodes

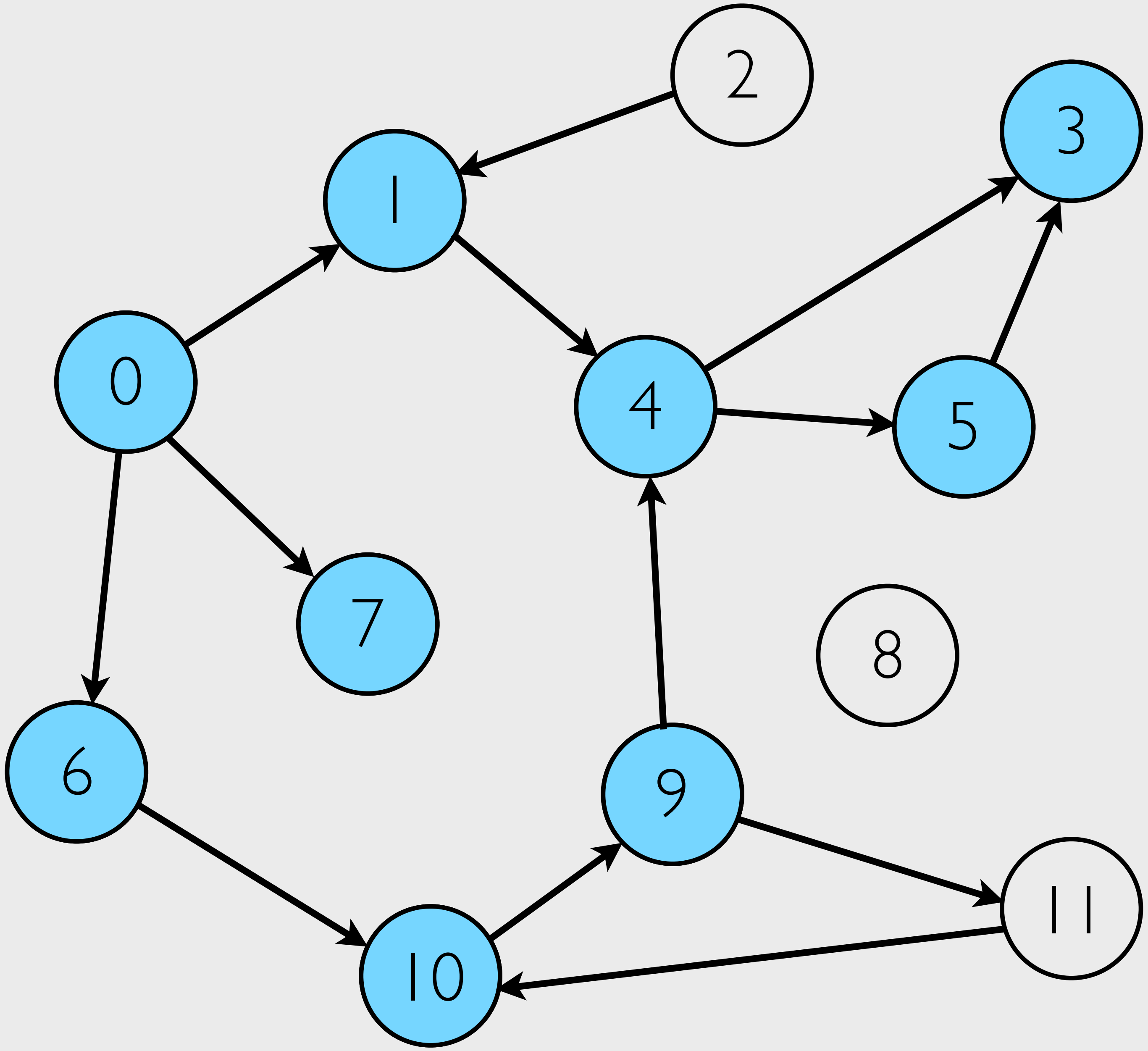


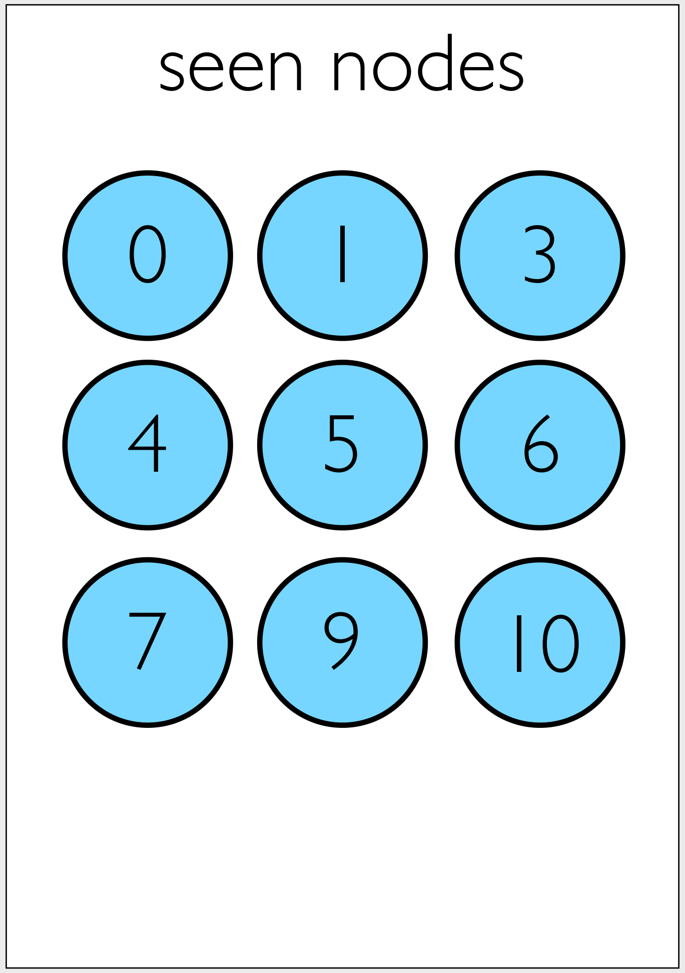

seen nodes

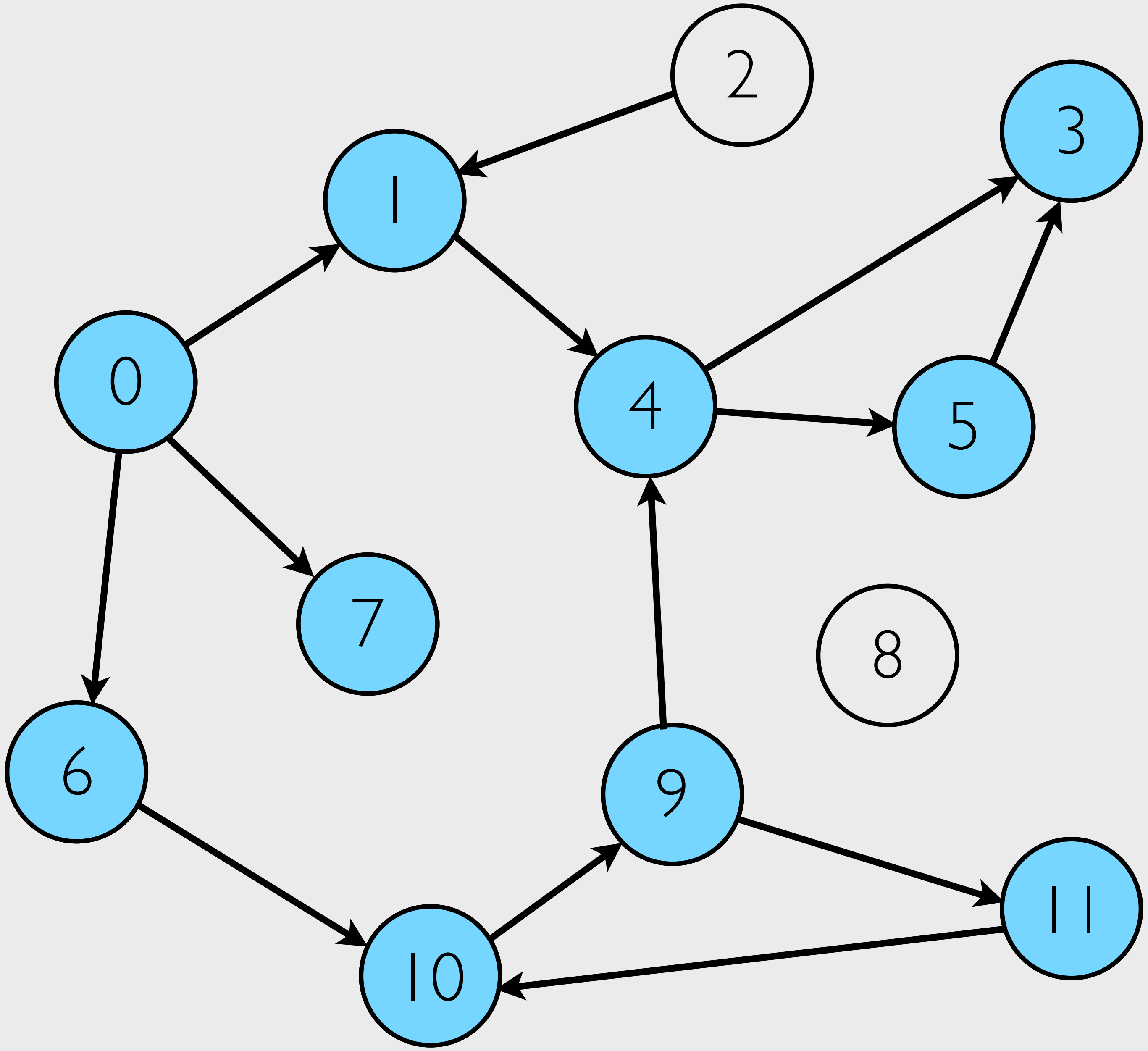


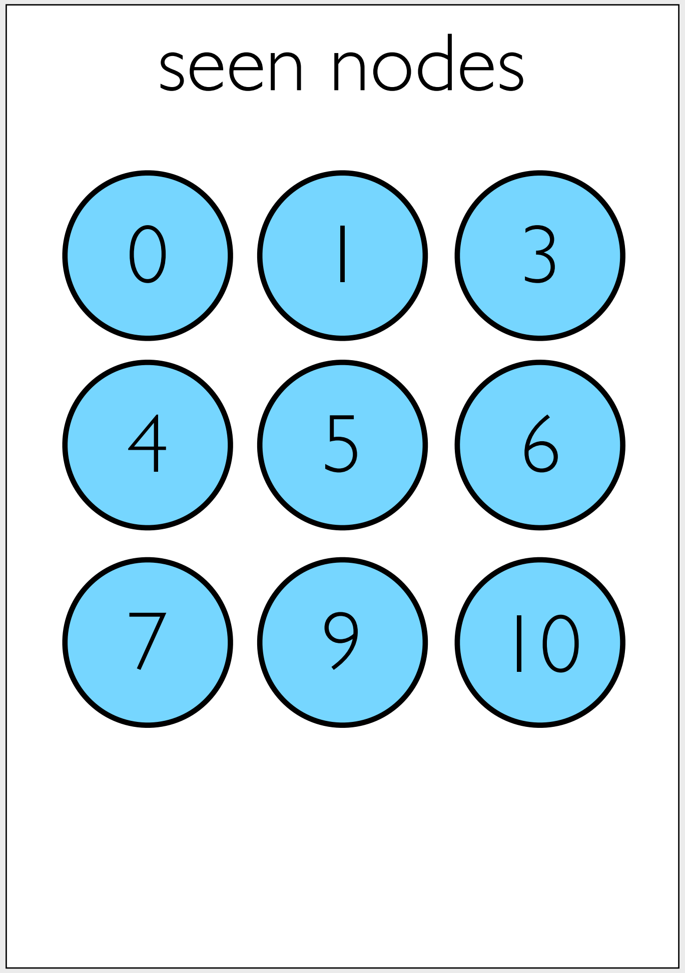
seen nodes

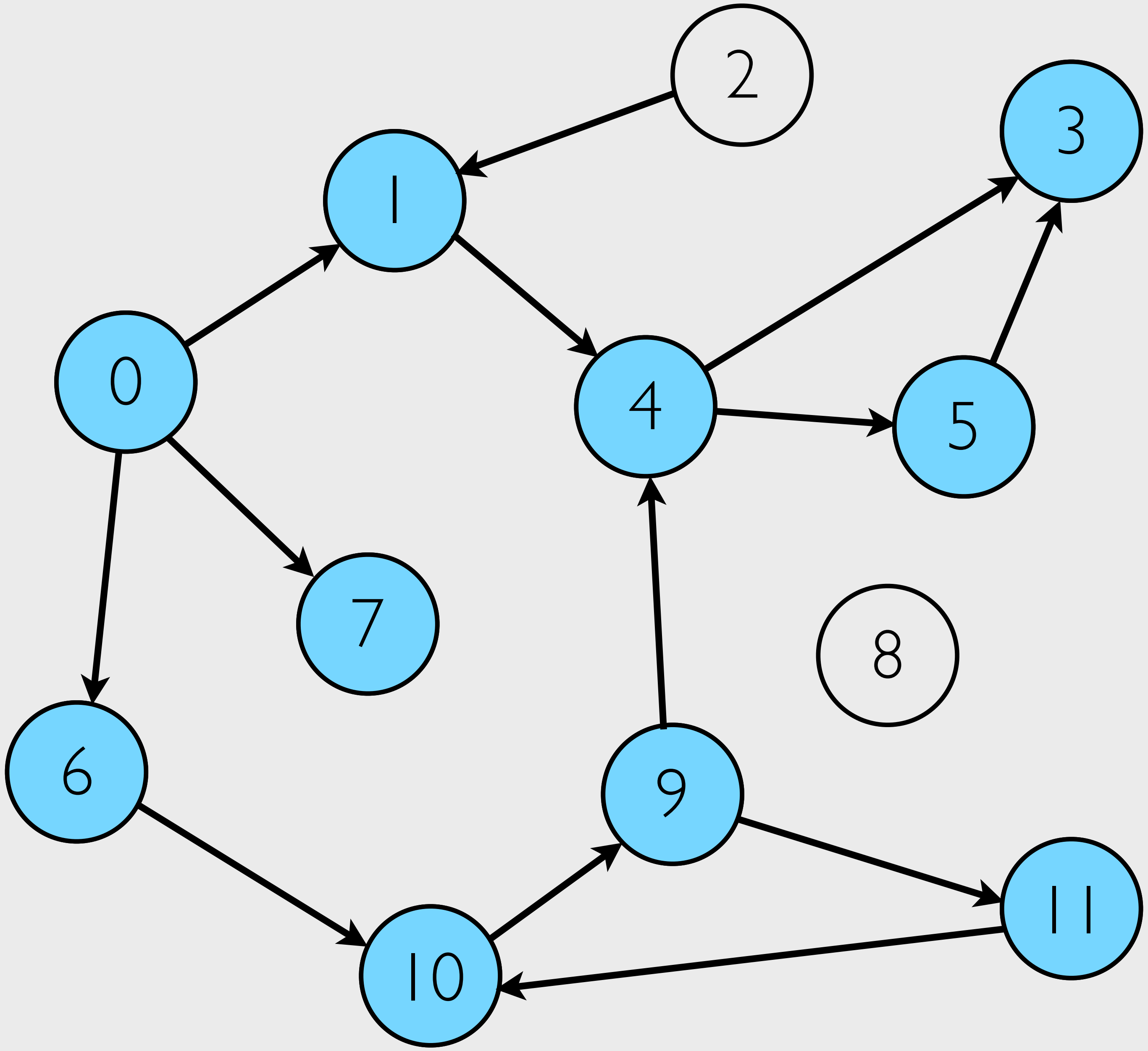


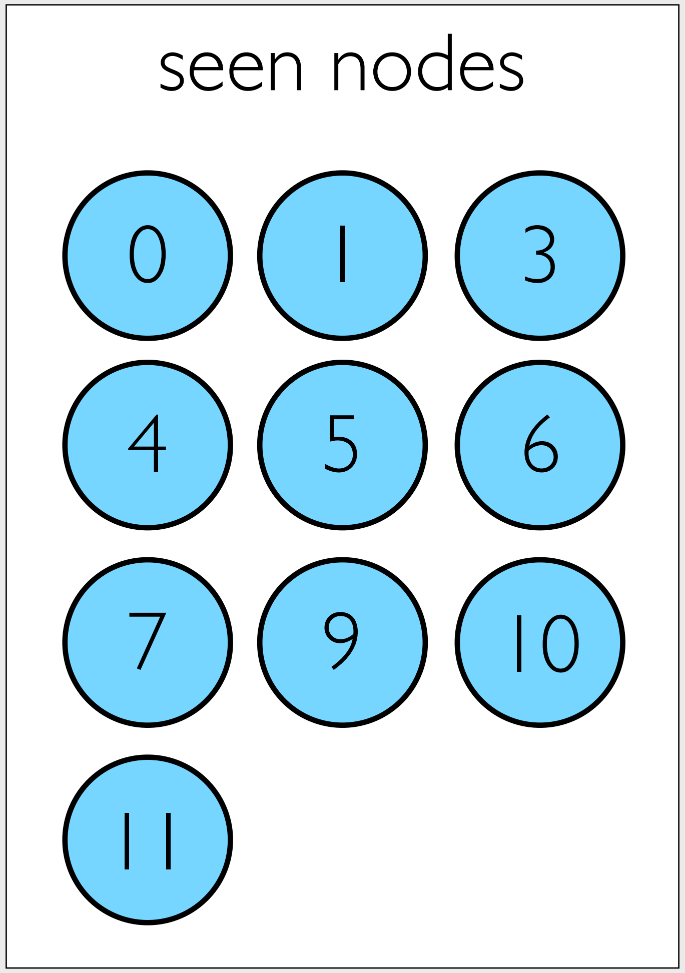


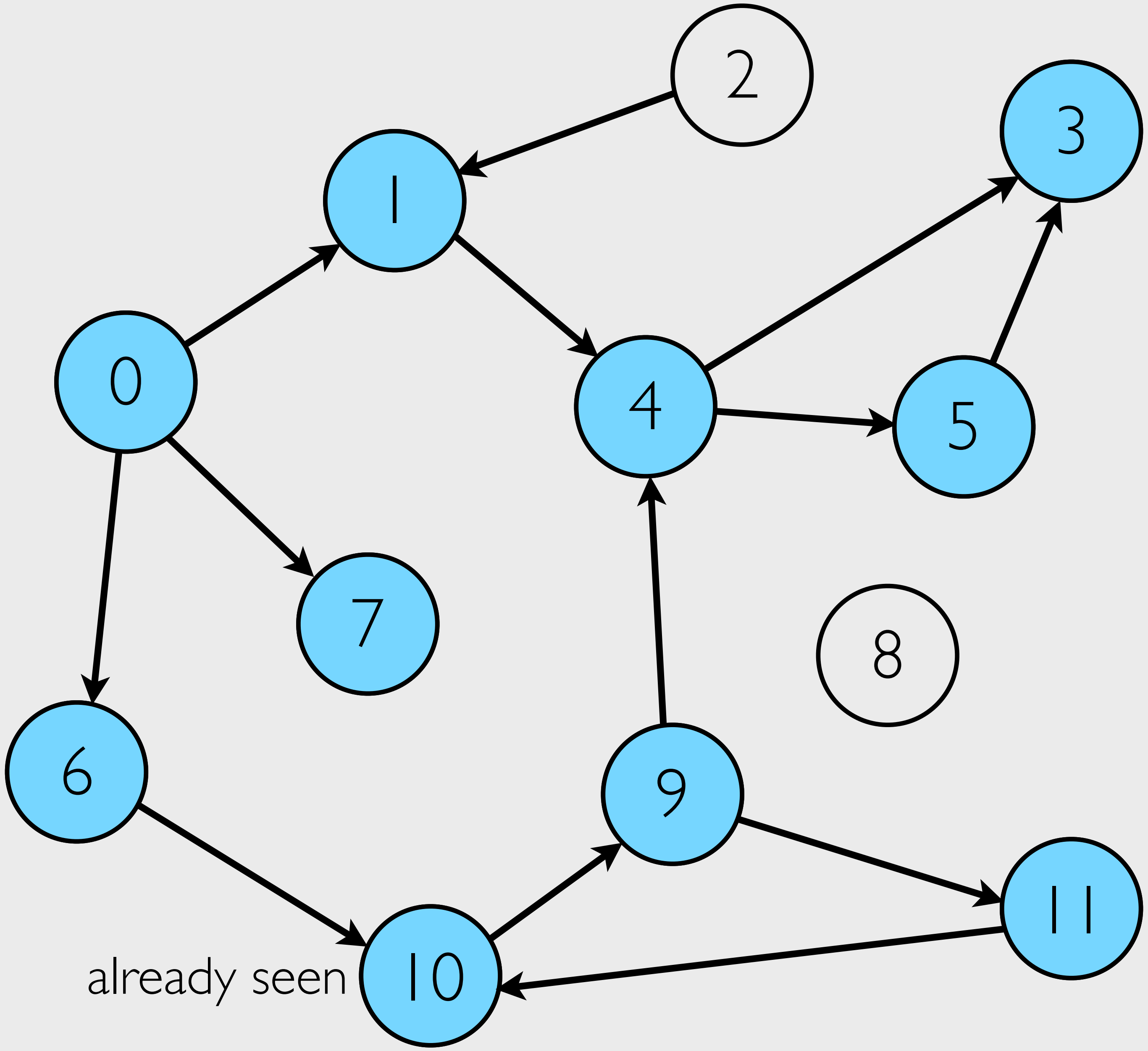

seen nodes



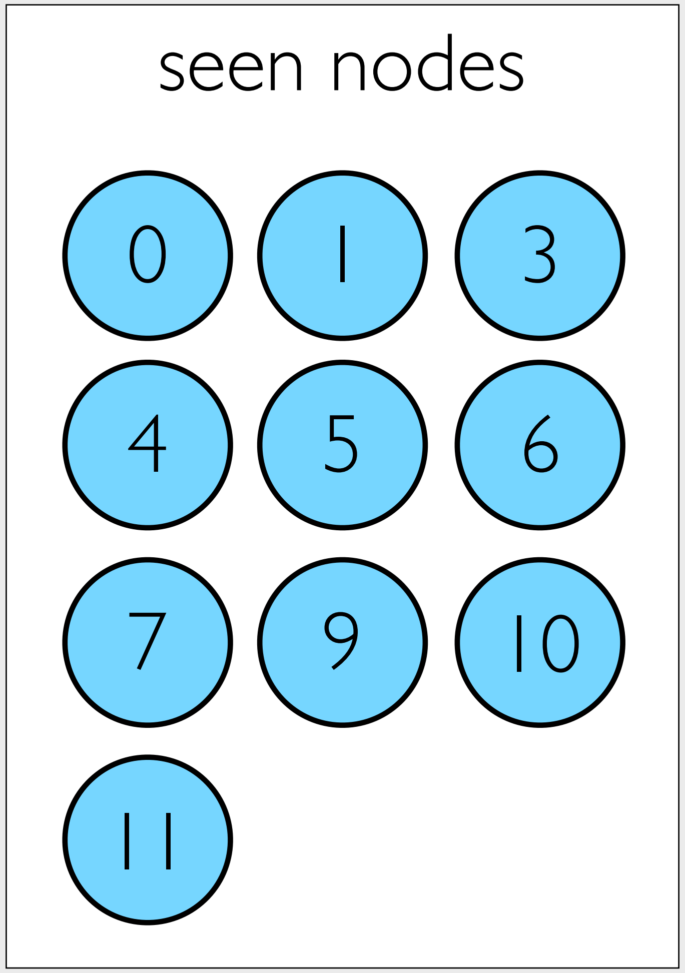

seen nodes



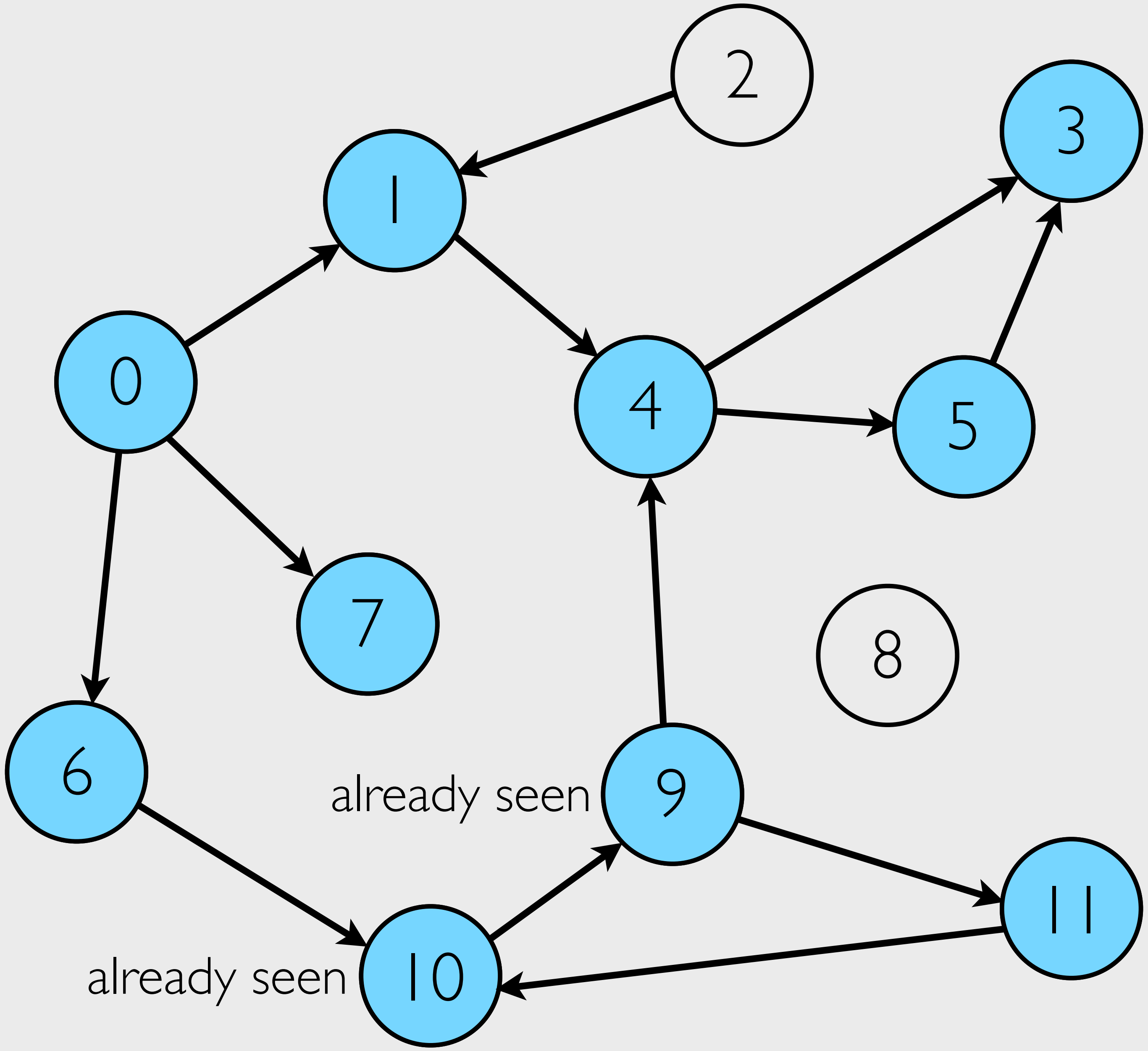

seen nodes



already seen

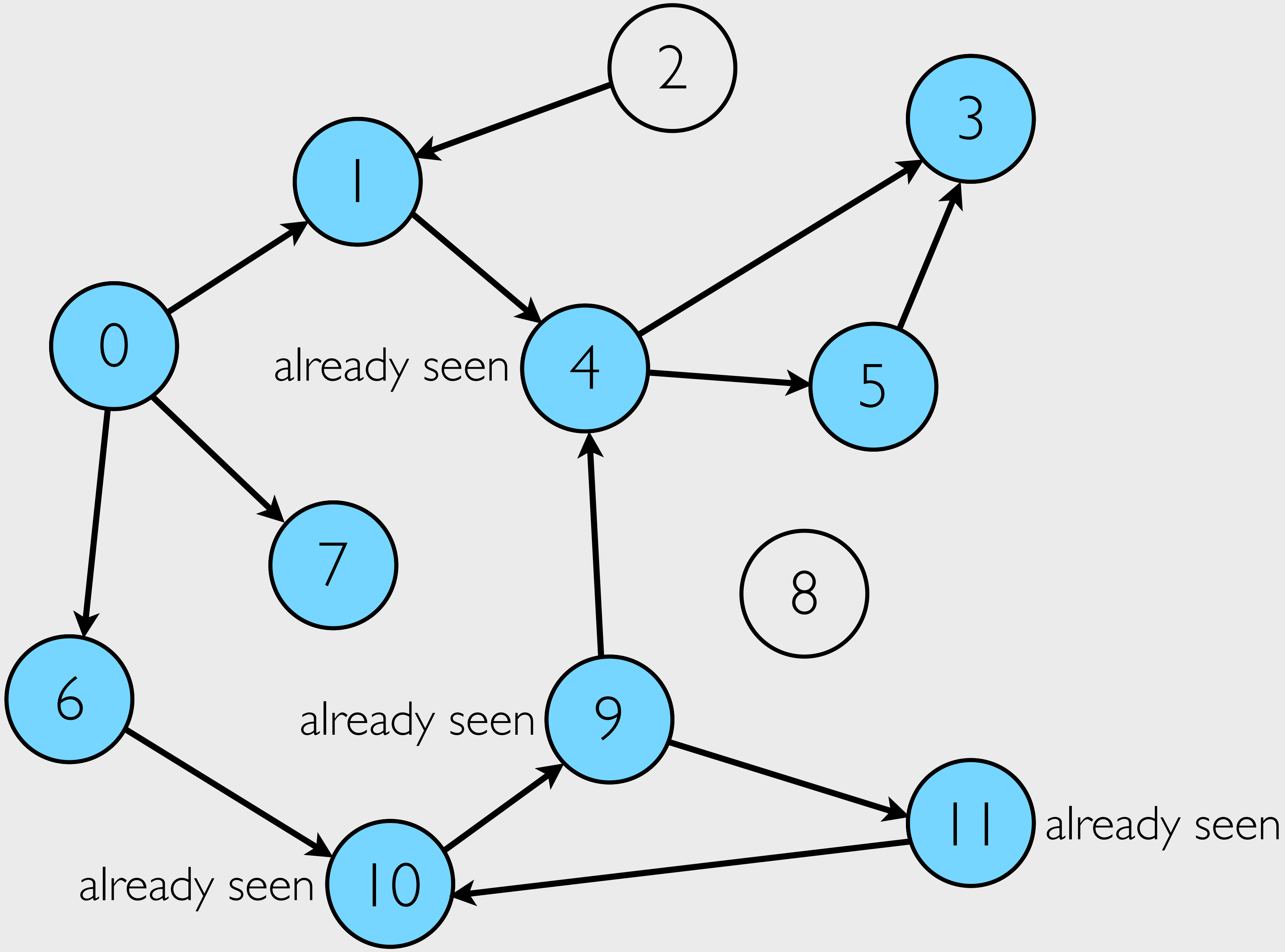


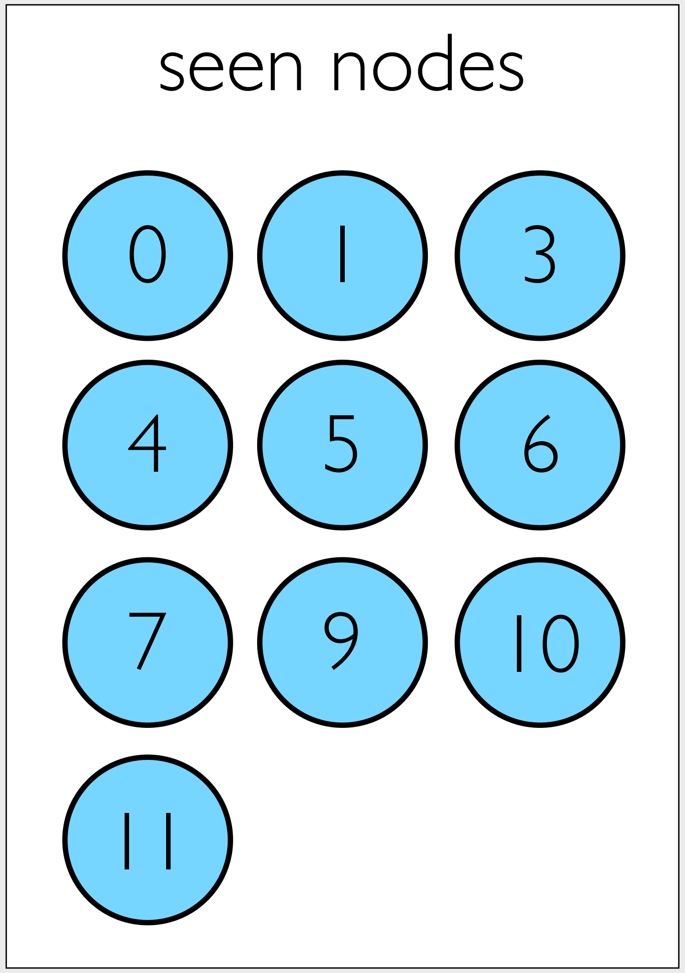
seen nodes

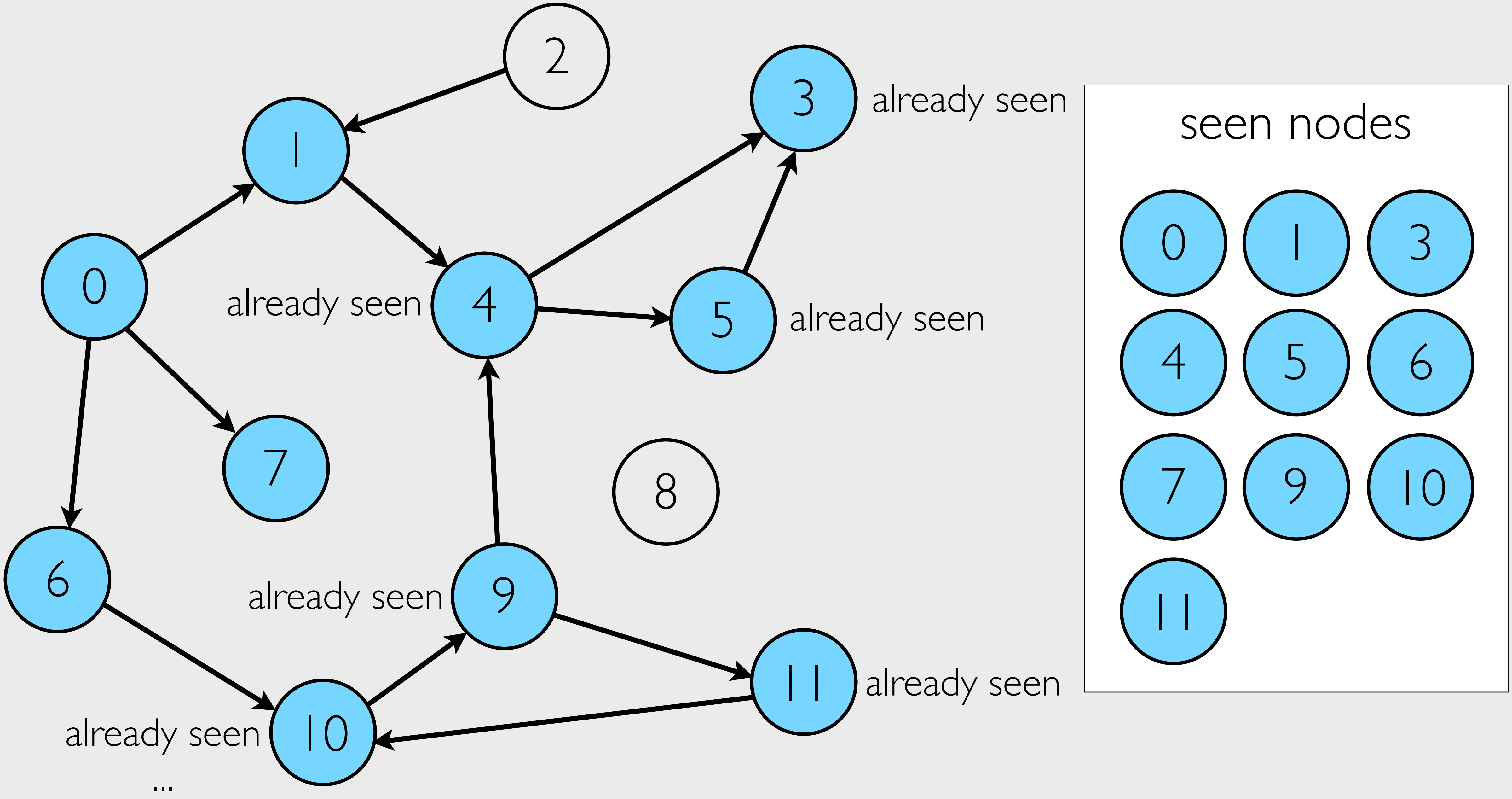


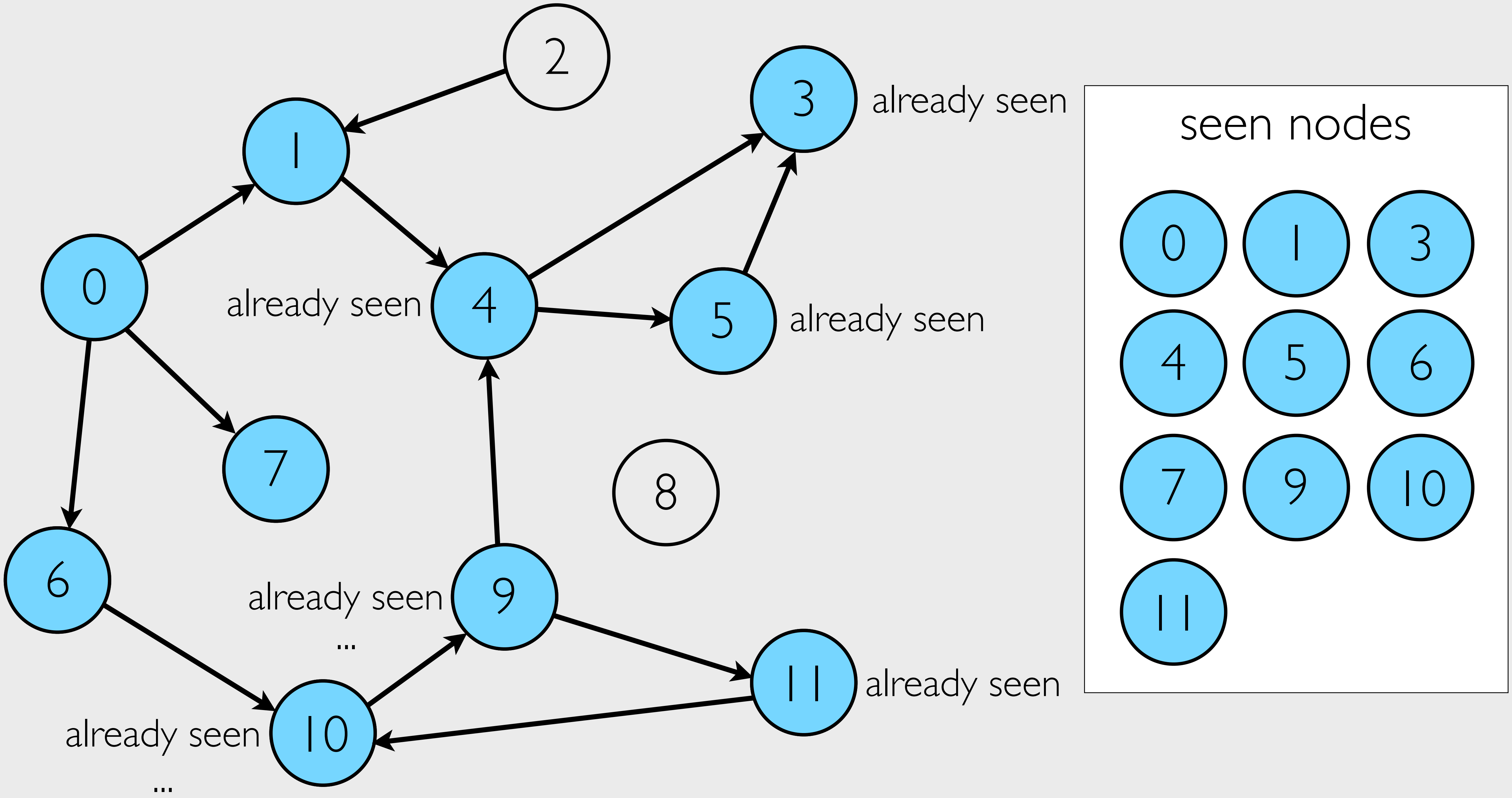
seen nodes

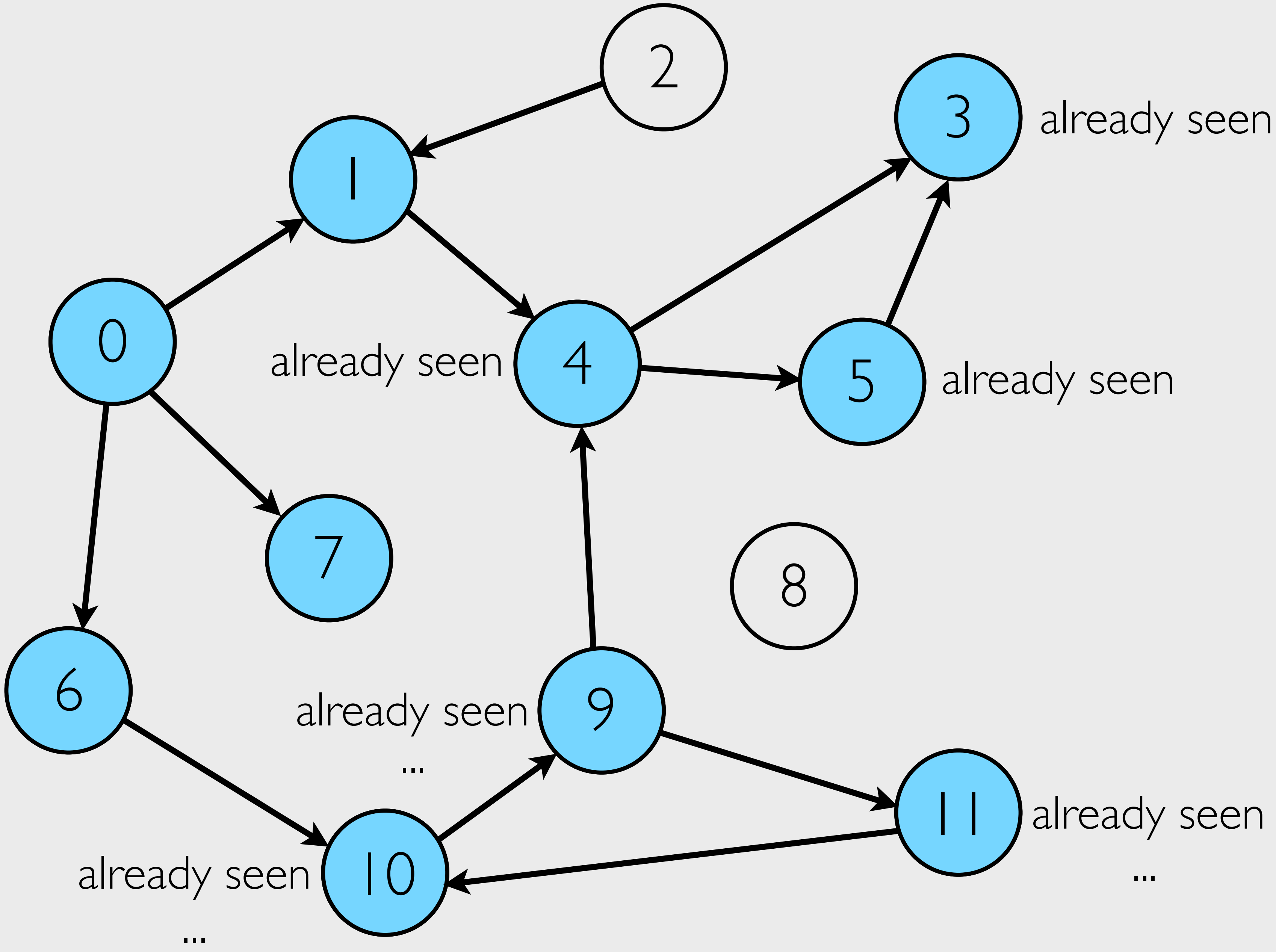
0 1 3
4 5 6
7 9 10
11

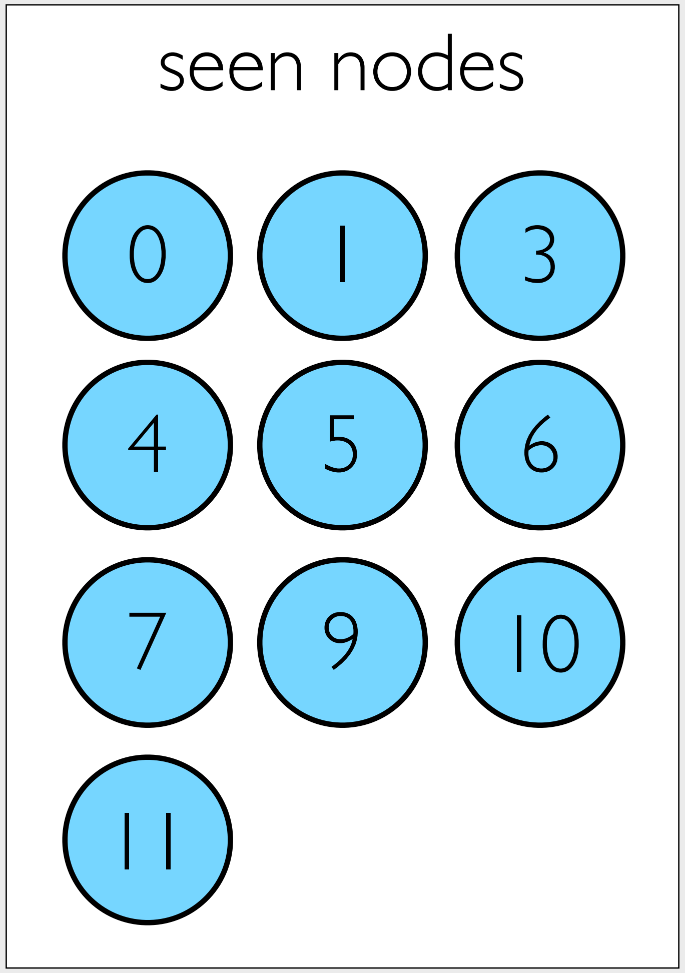

2
1
3
0
already seen
4
5
7
8
6
already seen
9
11
already seen
already seen
10


seen nodes
0
1
3
4
5
6
7
9
10
11

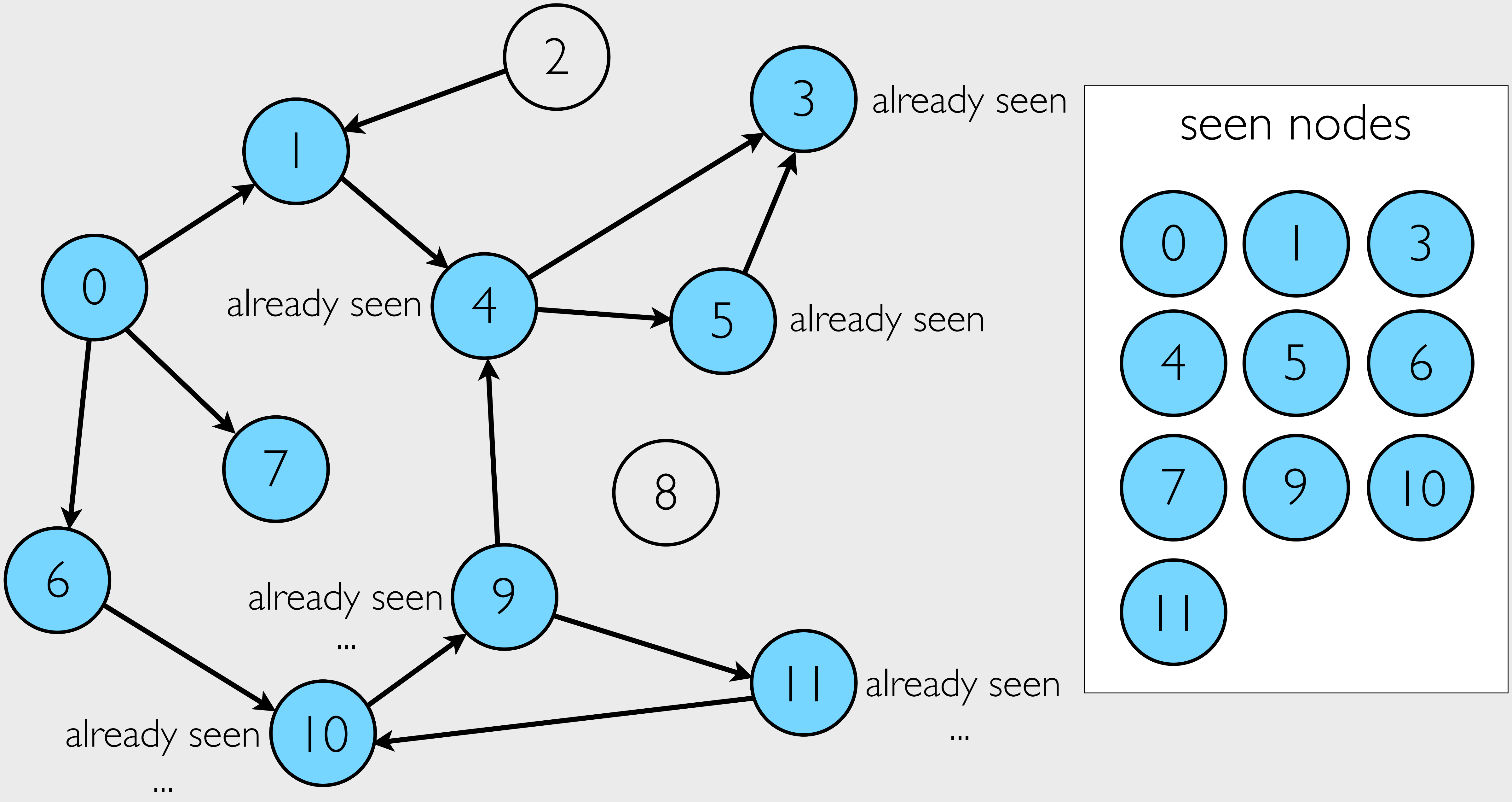

2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
9
11
already seen
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11


2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
9
...
11
already seen
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11


2
3
already seen
1
0
already seen
4
5
already seen
7
8
6
already seen
9
...
11
already seen
...
already seen
10
...

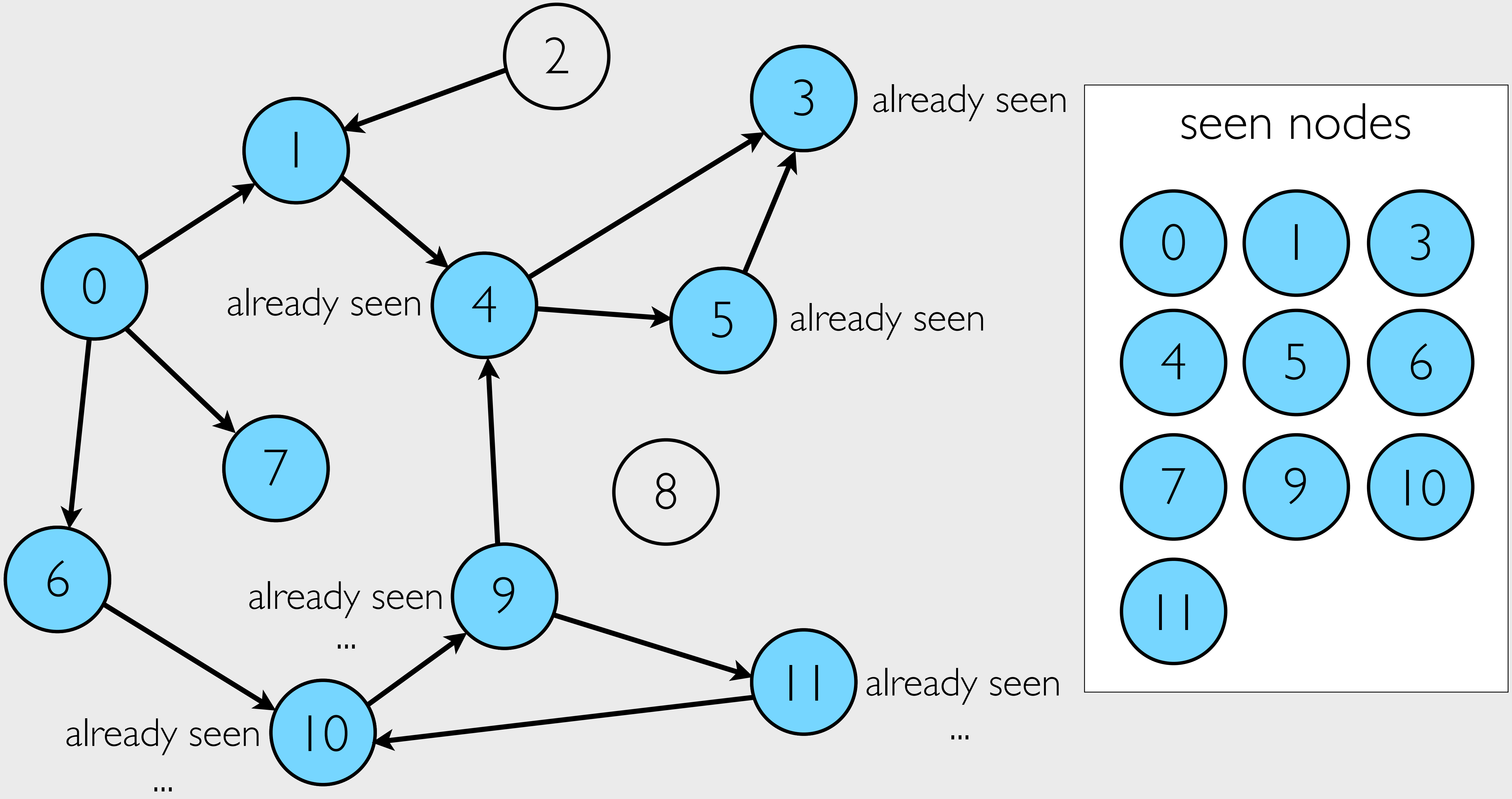

seen nodes
0
1
3
4
5
6
7
9
10
11

Events are updates that change an LVar's state



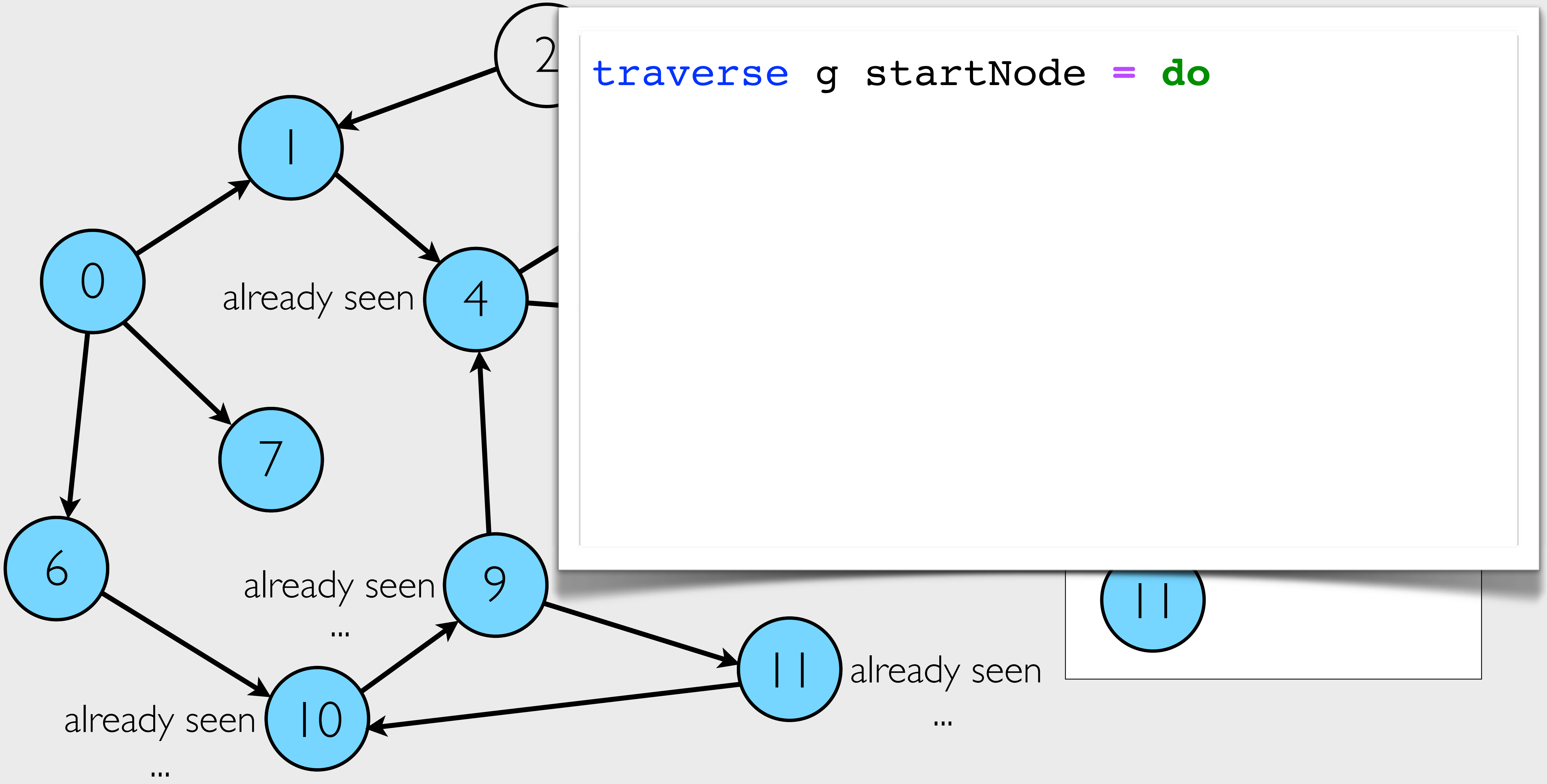
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response



Events are updates that change an LVar's state

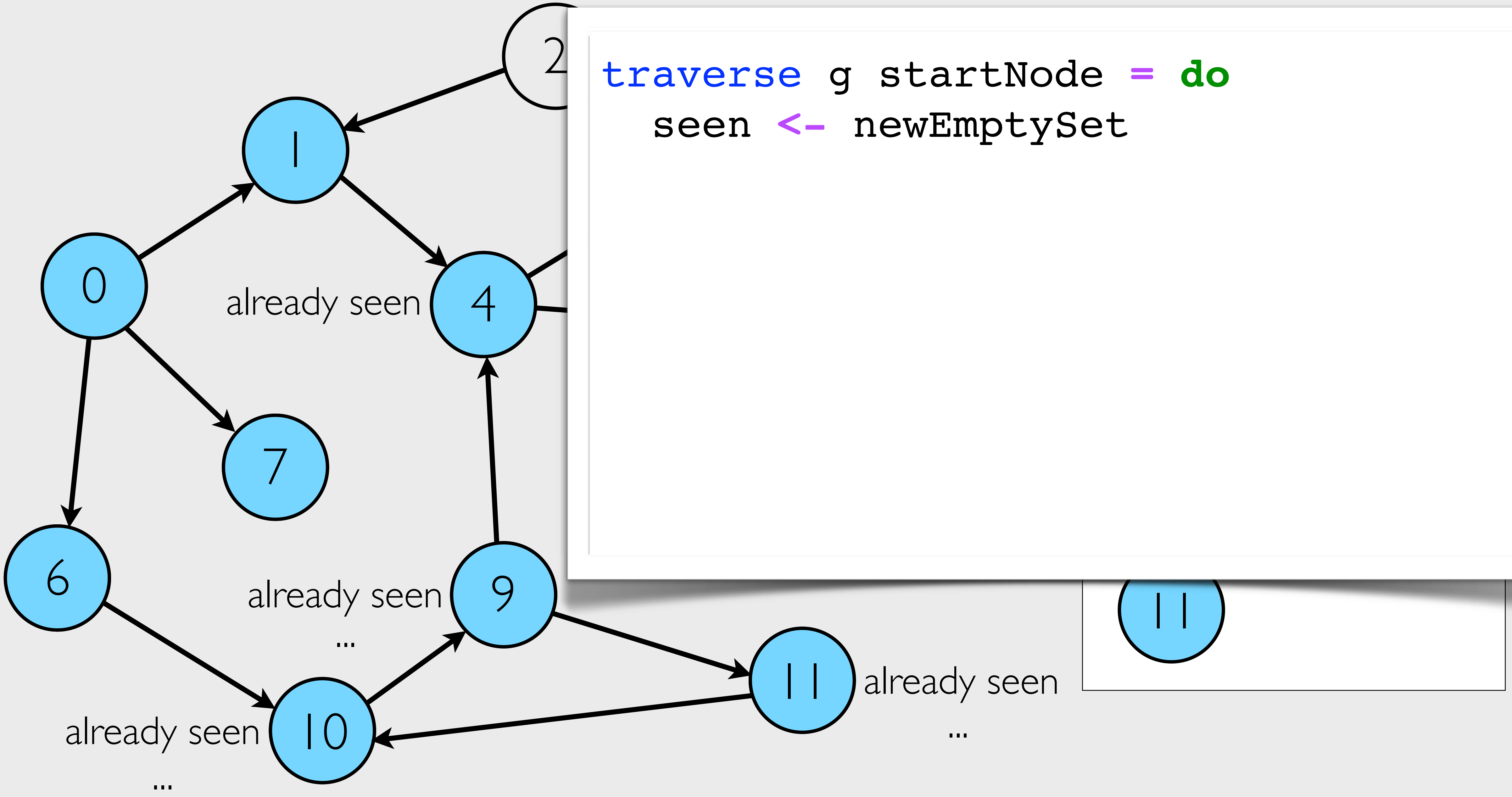
Event handlers listen for events and launch callbacks in response



```
traverse g startNode = do
```

Events are updates that change an LVar's state

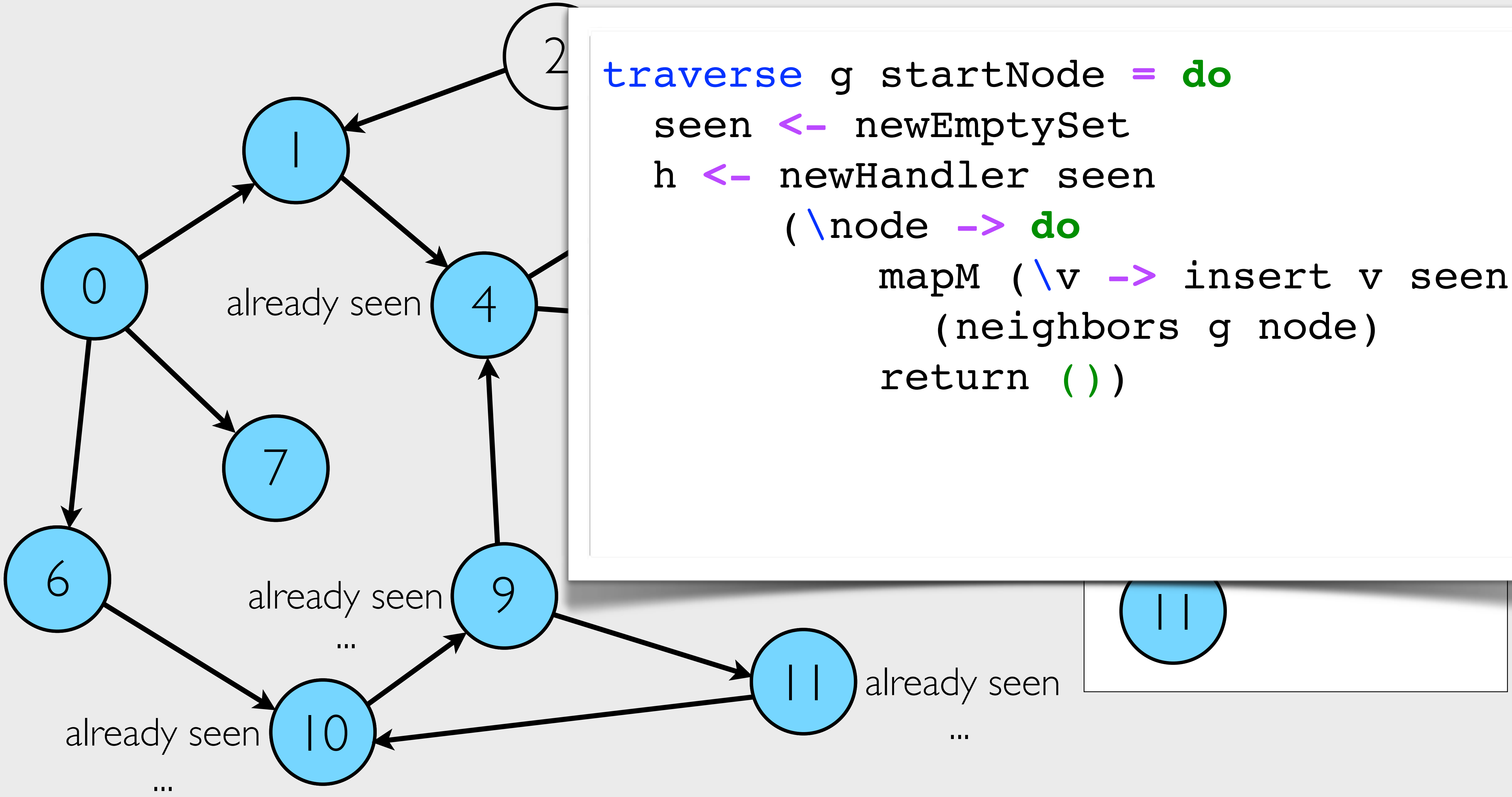
Event handlers listen for events and launch callbacks in response



```
traverse g startNode = do
  seen <- newEmptySet
```

Events are updates that change an LVar's state

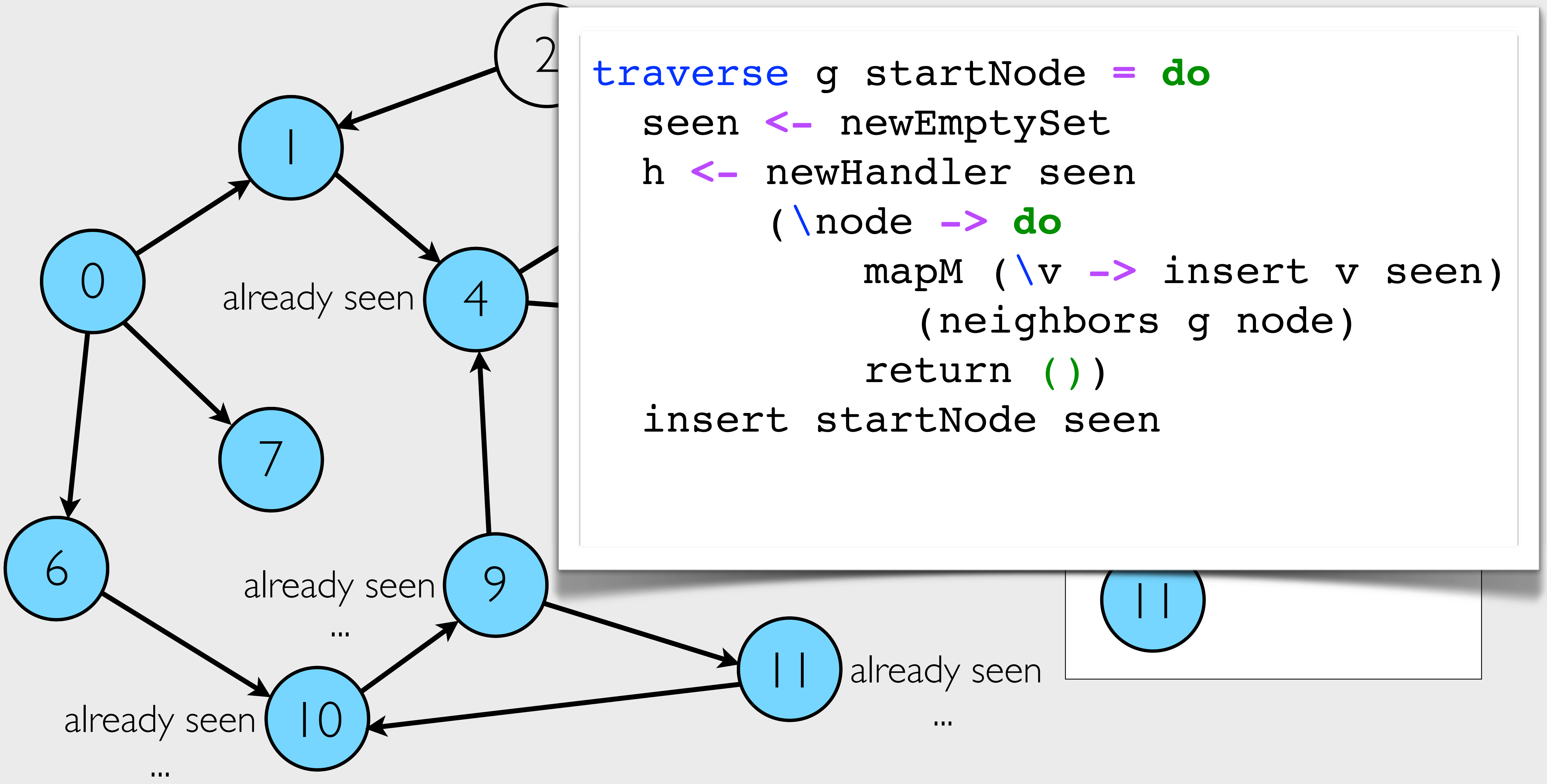
Event handlers listen for events and launch callbacks in response



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

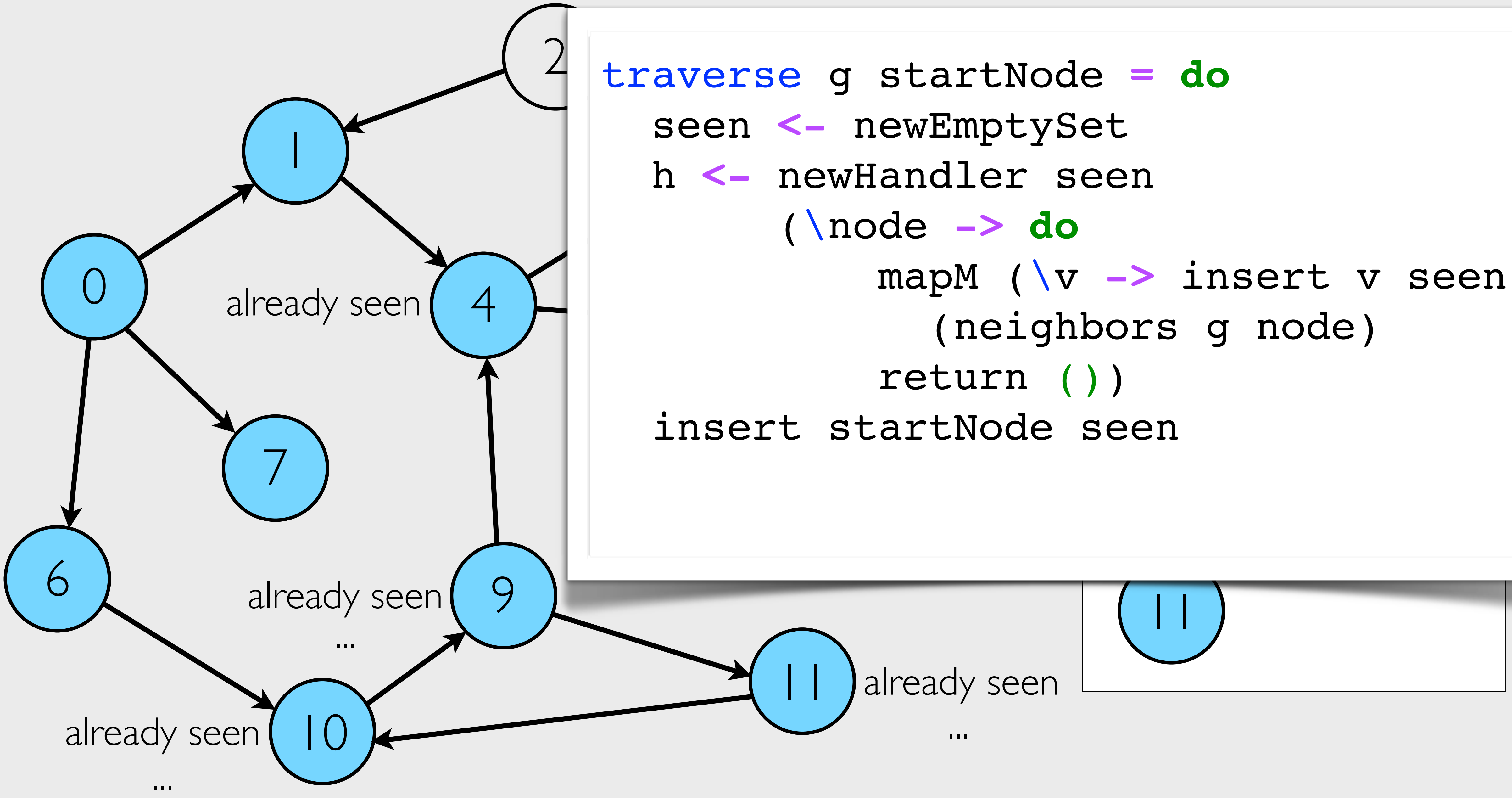


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

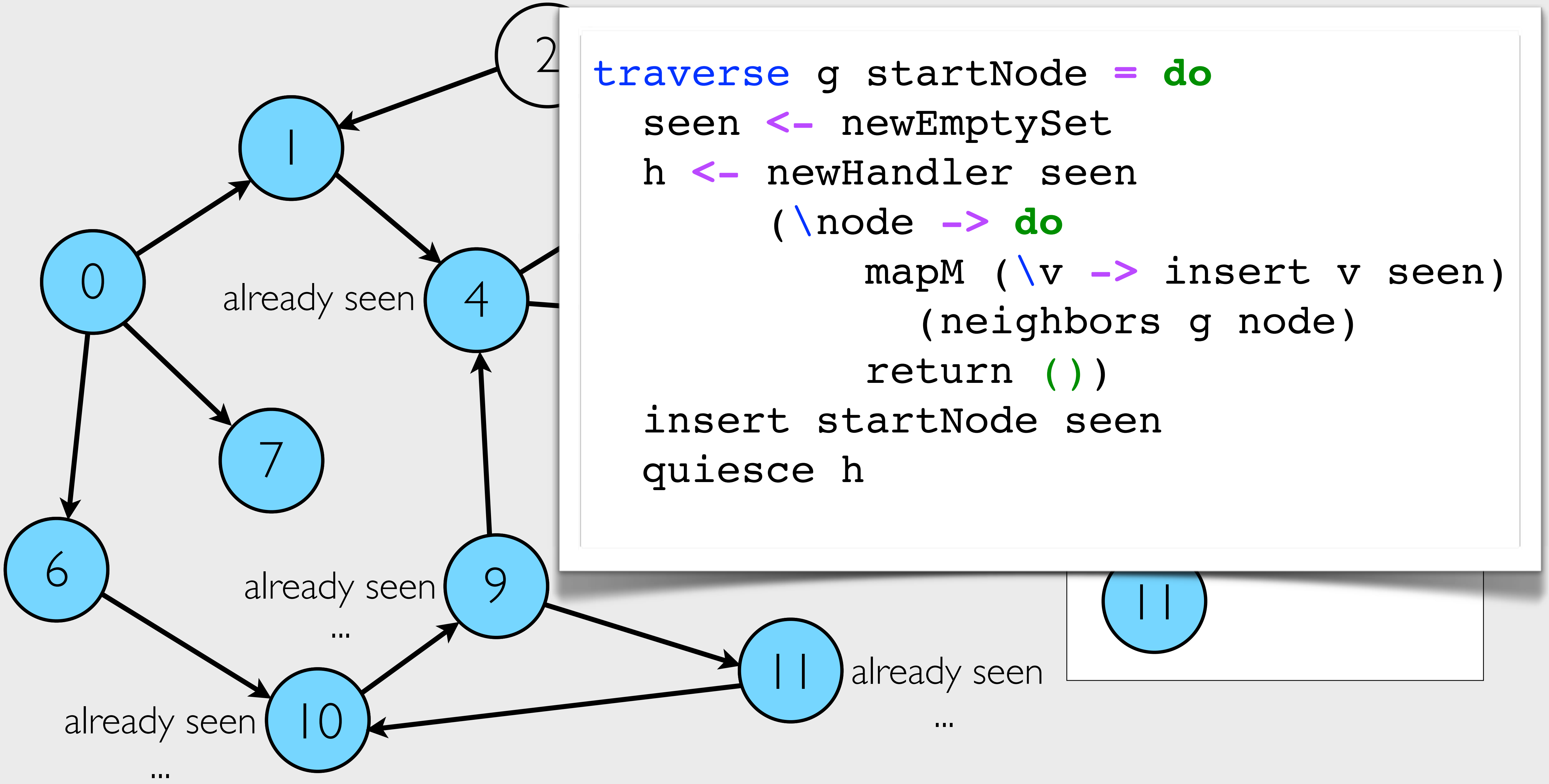


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
         (\node -> do
              mapM (\v -> insert v seen)
                (neighbors g node)
              return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

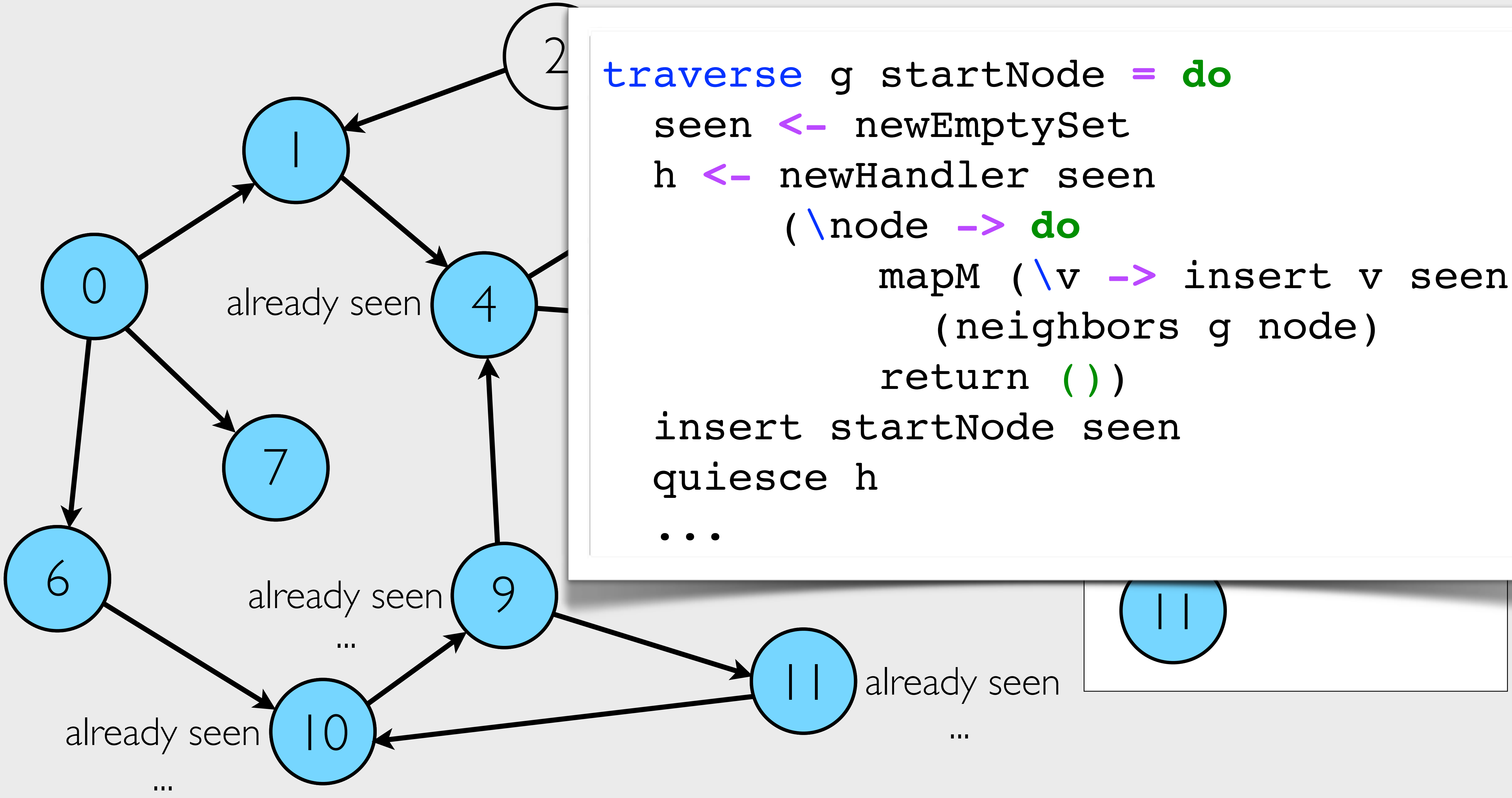


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
```

Events are updates that change an LVar's state

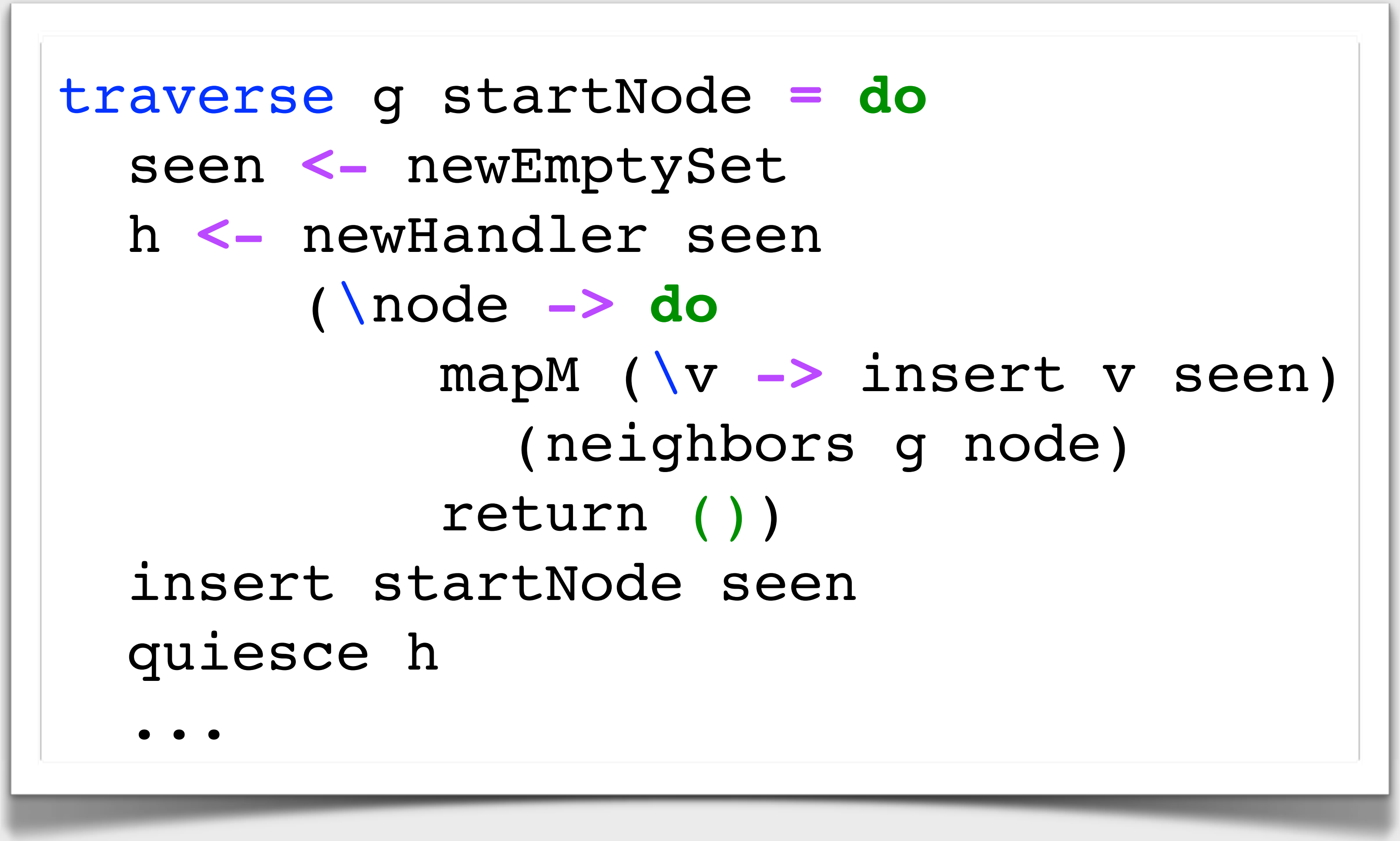
Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
  ...
```

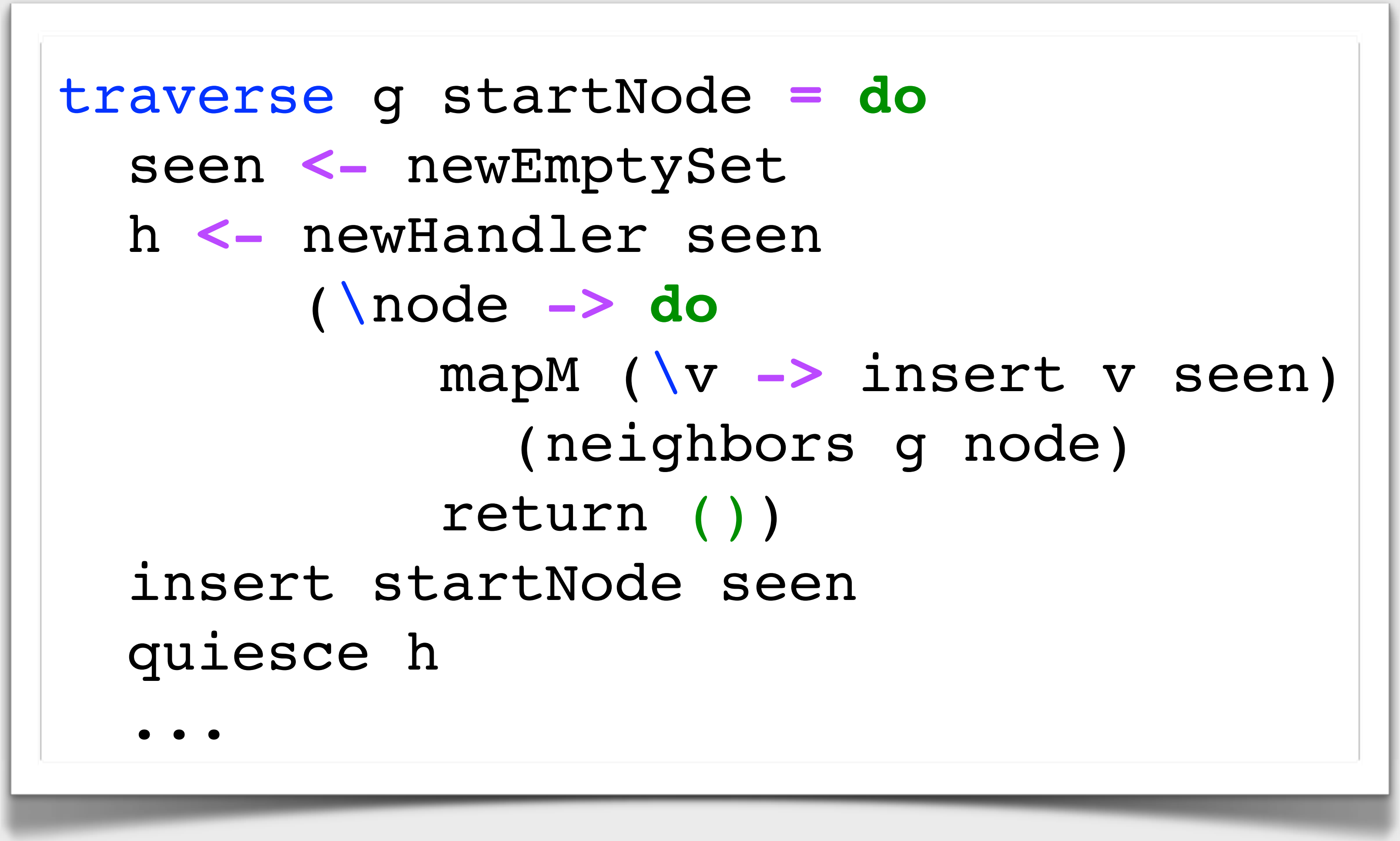
`freeze`: exact non-blocking read



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

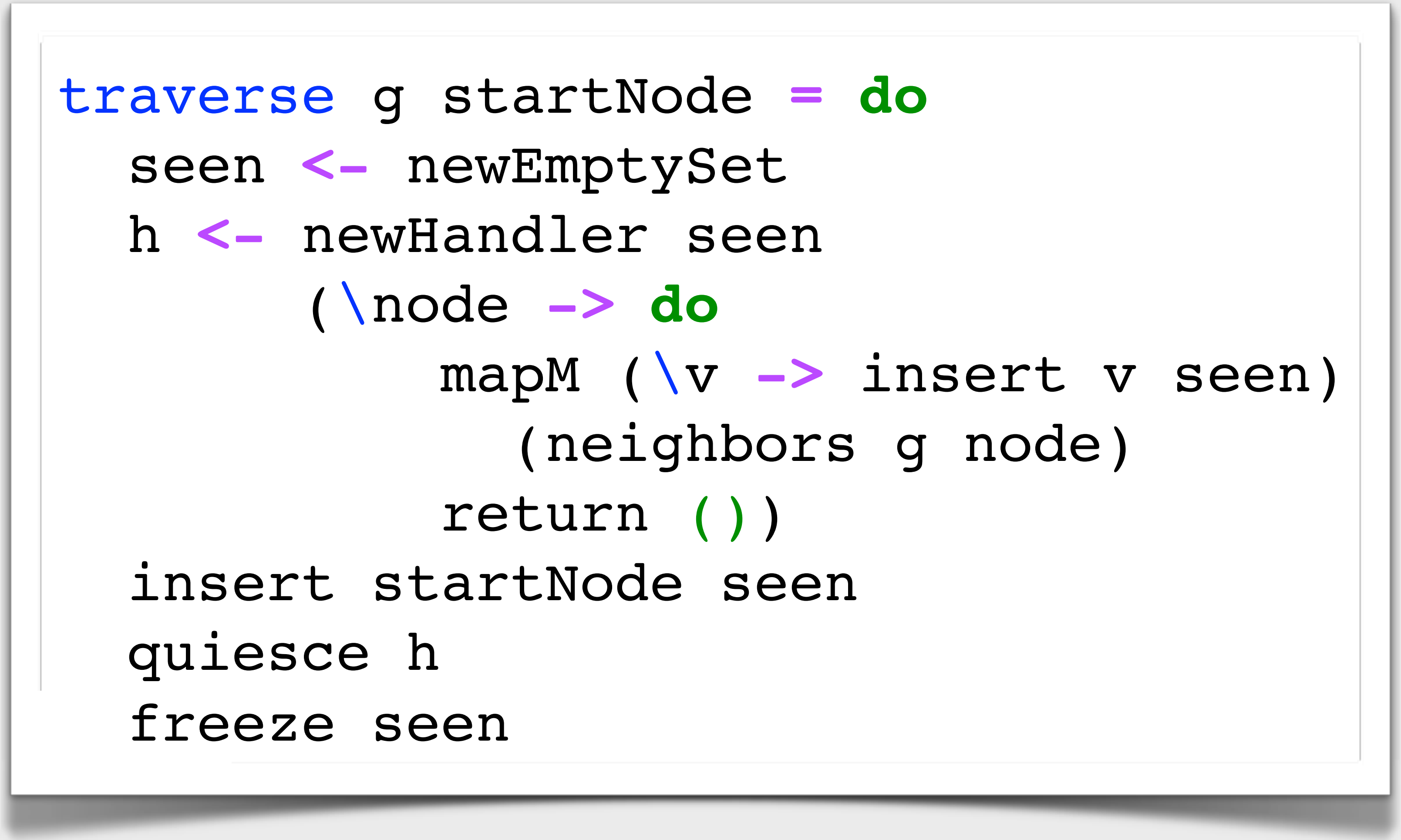
Attempts to write to a frozen LVar raise a write-after-`freeze` exception



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

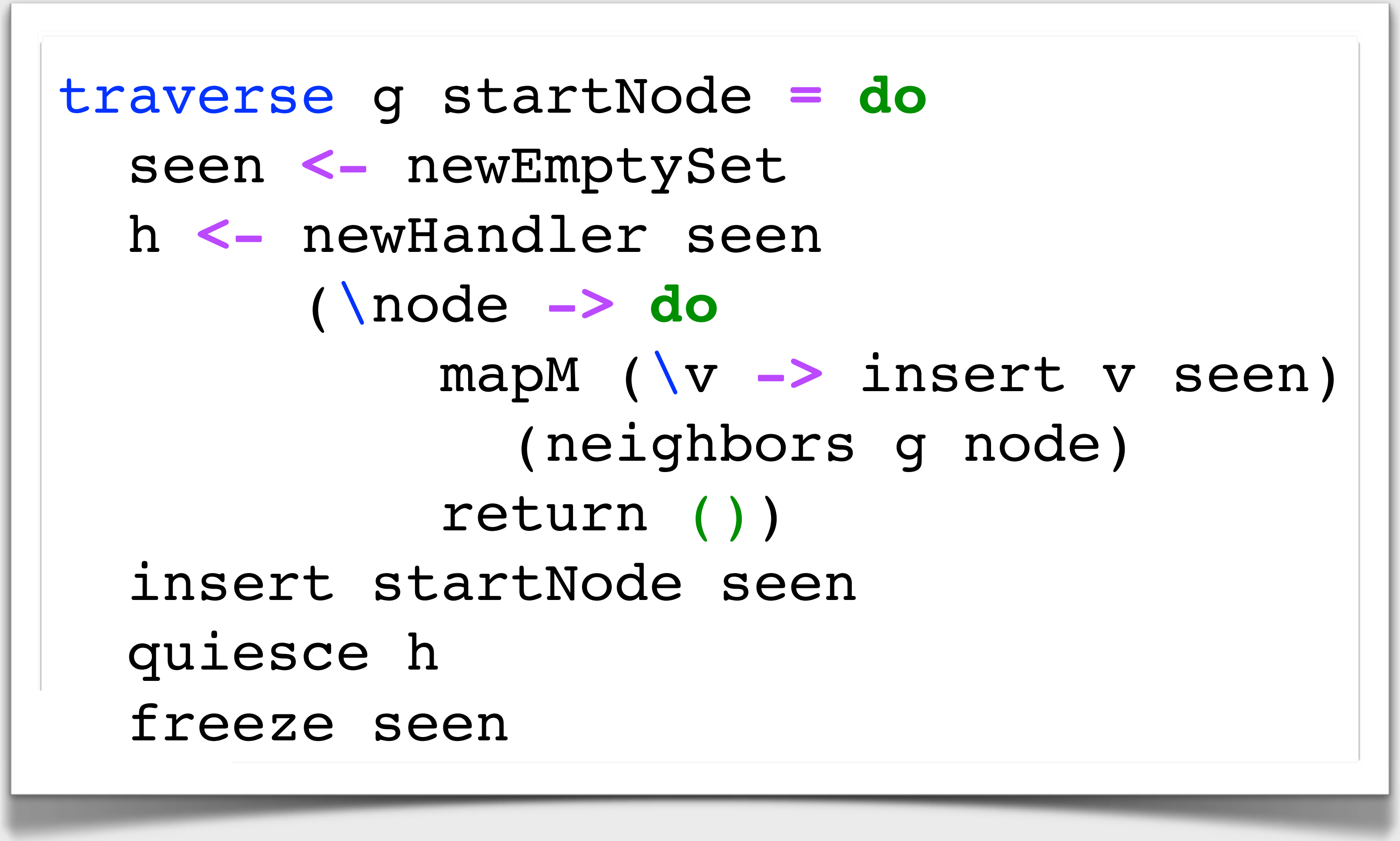


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception

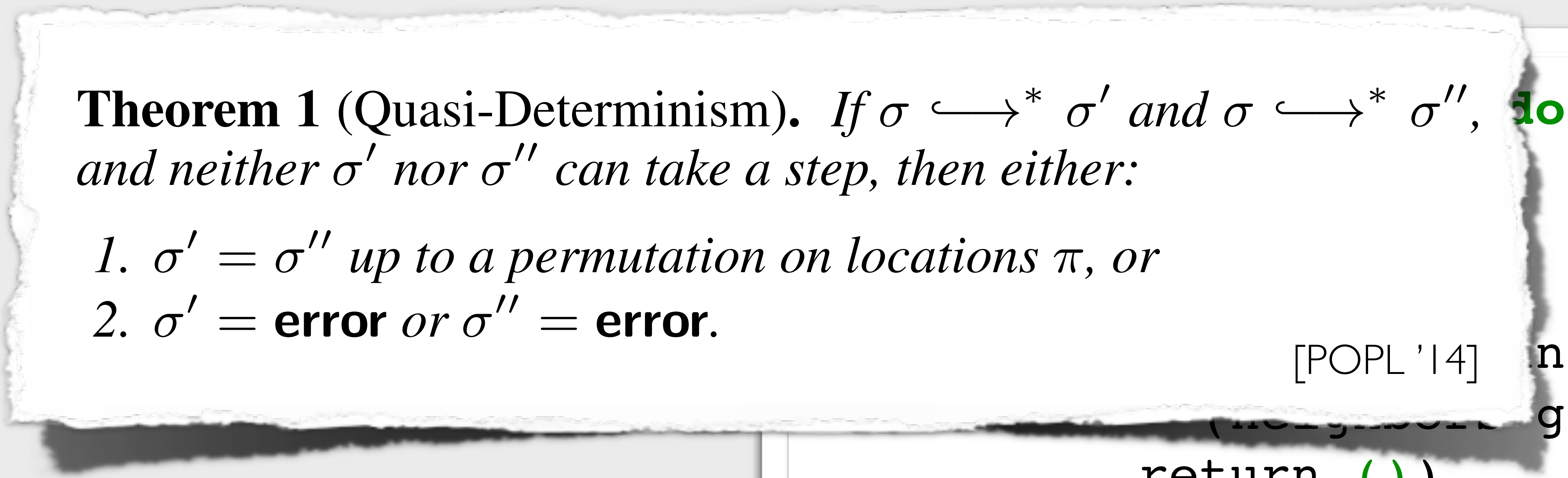


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception



Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

1. *$\sigma' = \sigma''$ up to a permutation on locations π, or*
2. *$\sigma' =$ **error** or $\sigma'' =$ **error**.*

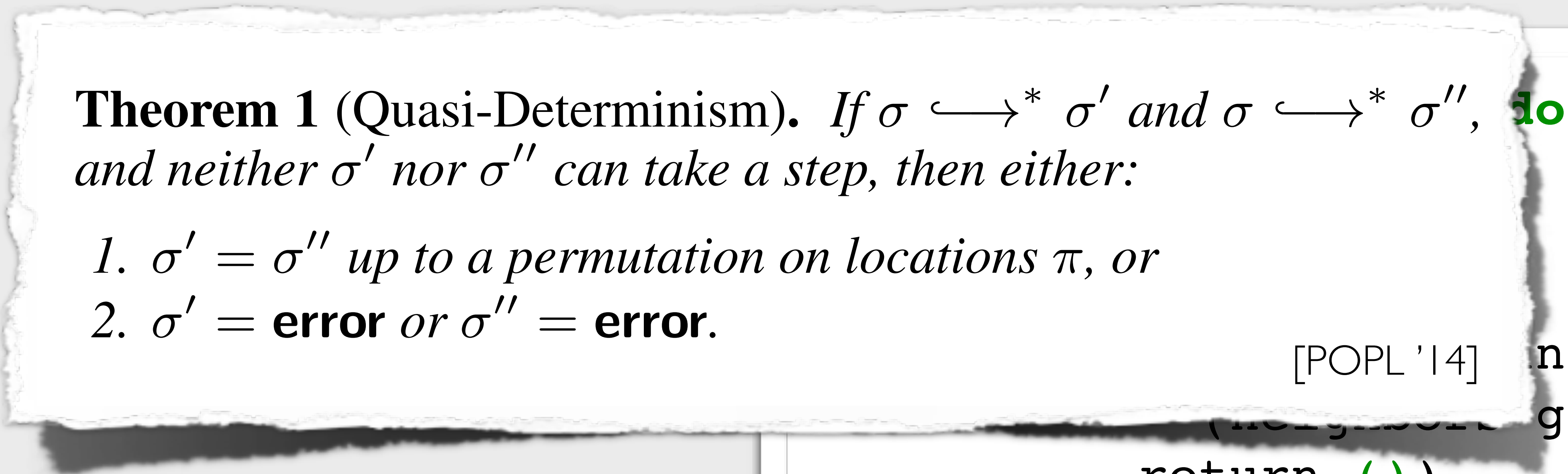
[POPL '14]

```
                                 do




                                 nsert v seen)
                                 g node)
                 return ())
     insert startNode seen
     quiesce h
     freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

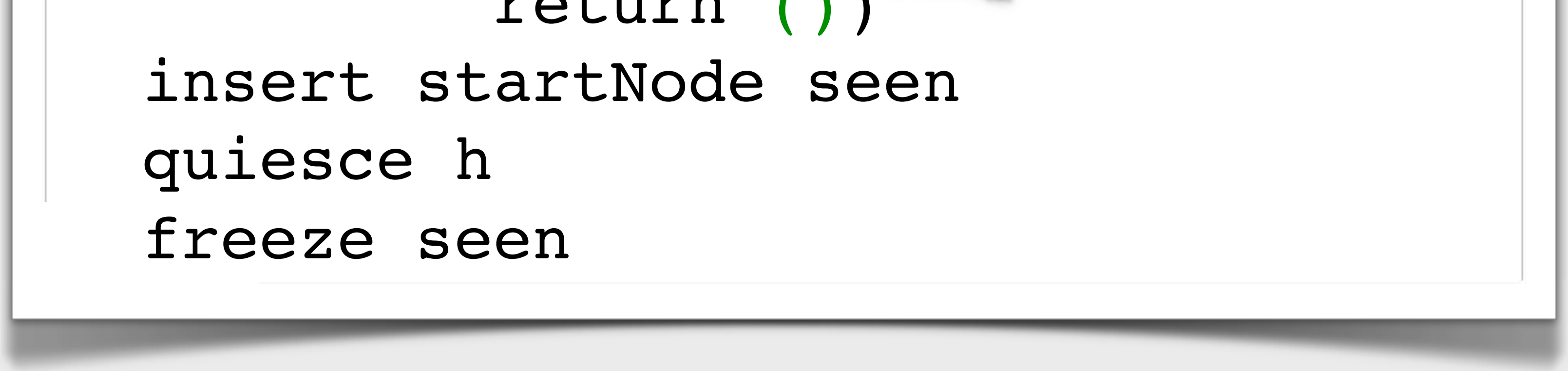
Two possible outcomes: either the same final value or an exception



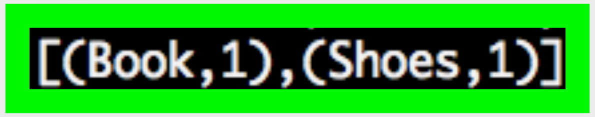
Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

1. *$\sigma' = \sigma''$ up to a permutation on locations π, or*
2. *$\sigma' =$ **error** or $\sigma'' =$ **error**.*

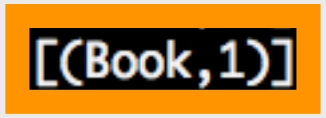
[POPL '14]



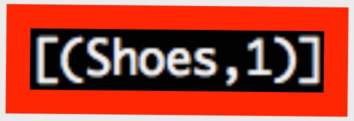
```
                                  do



                                  nsert v seen)
                                  g node)
         return ())
insert startNode seen
quiesce h
freeze seen
```



```
[(Book,1),(Shoes,1)]
```



```
[(Book,1)]
```

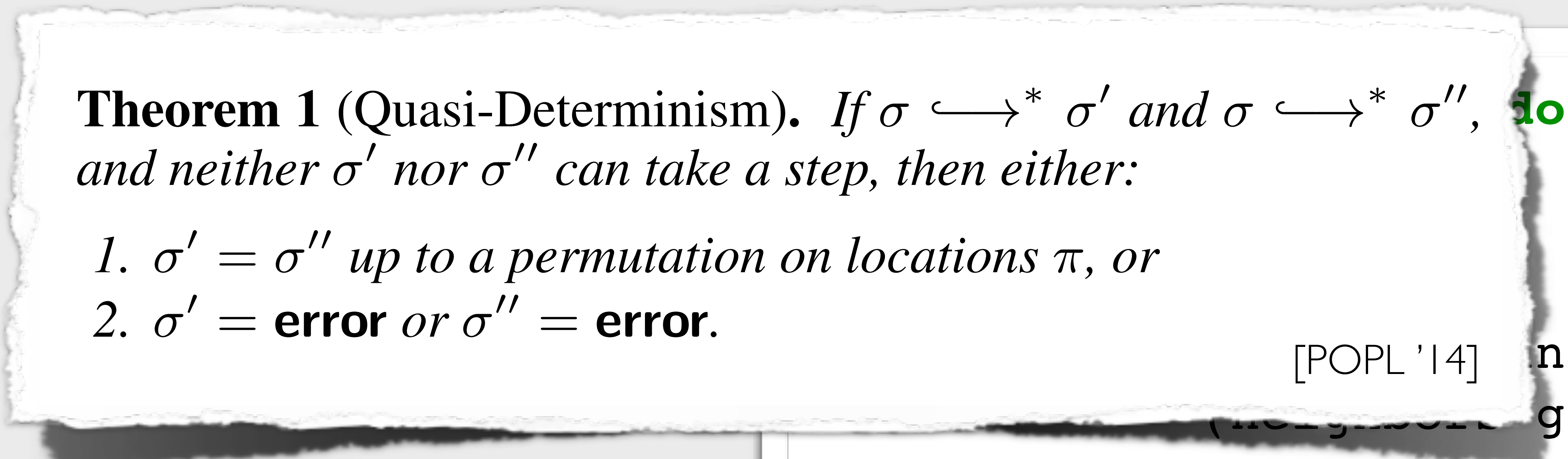


```
[(Shoes,1)]
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

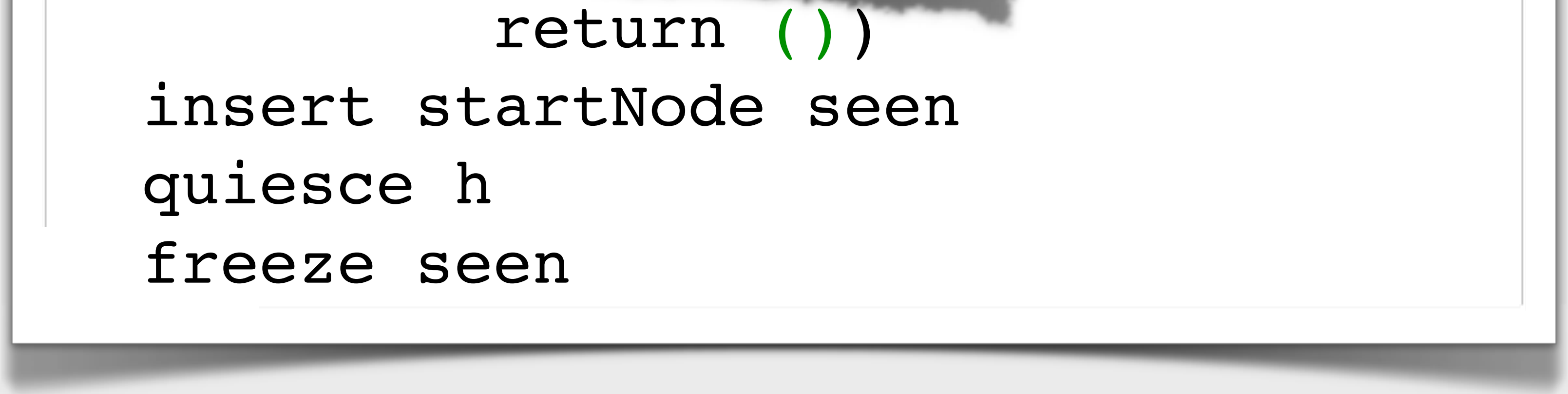
Two possible outcomes: either the same final value or an exception



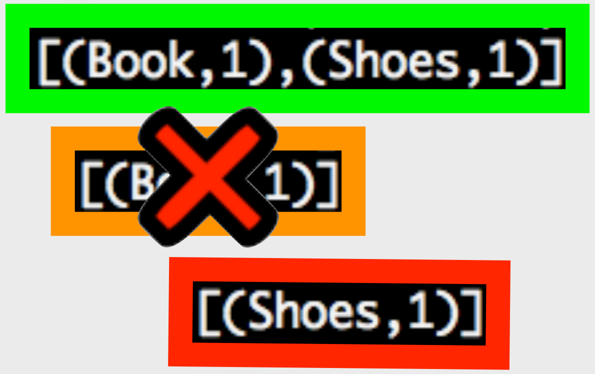
Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

1. *$\sigma' = \sigma''$ up to a permutation on locations π, or*
2. *$\sigma' = \textbf{error}$ or $\sigma'' = \textbf{error}$.*

[POPL '14]



```
                                                      do




                                                      nsert v seen)
                                                      g node)
                 return ())
  insert startNode seen
  quiesce h
  freeze seen
```



```
[(Book,1),(Shoes,1)]
```

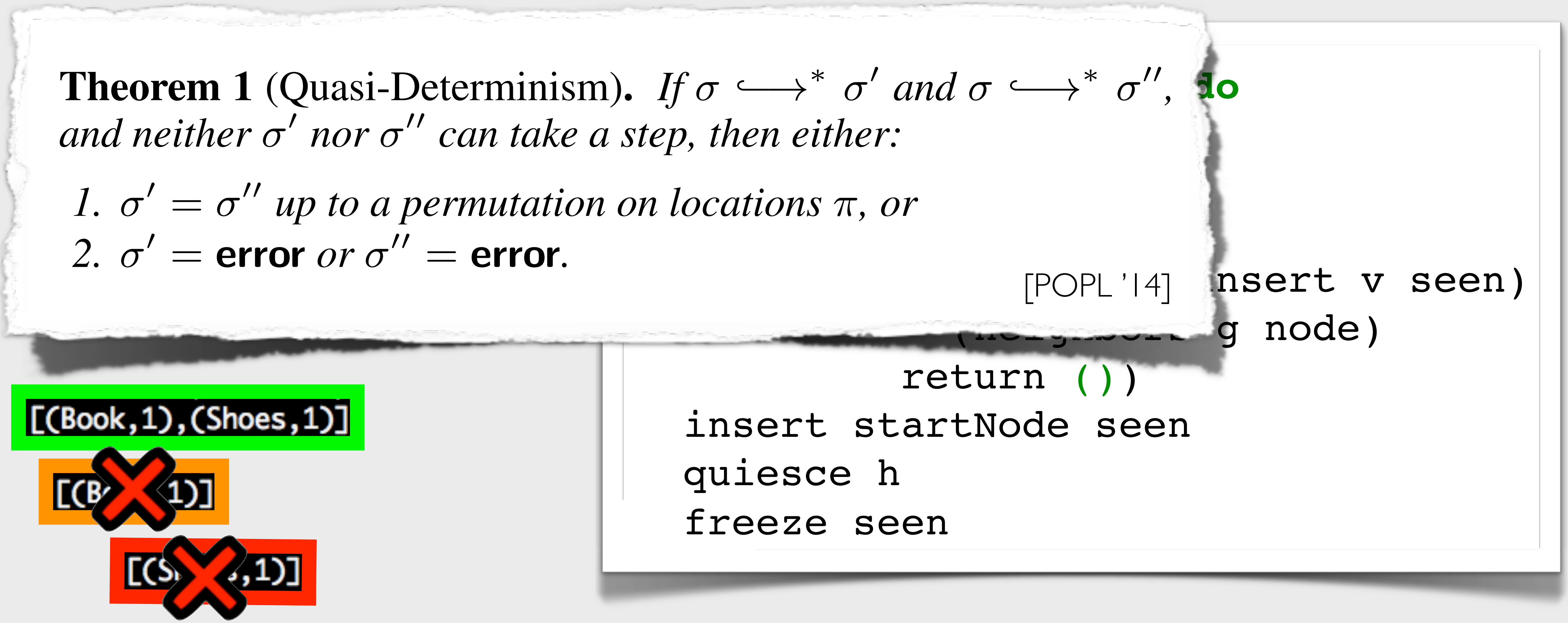
```
[(B 1)]
```

```
[(Shoes,1)]
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception



Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

1. *$\sigma' = \sigma''$ up to a permutation on locations π, or*
2. *$\sigma' =$ **error** or $\sigma'' =$ **error**.*

[POPL '14]

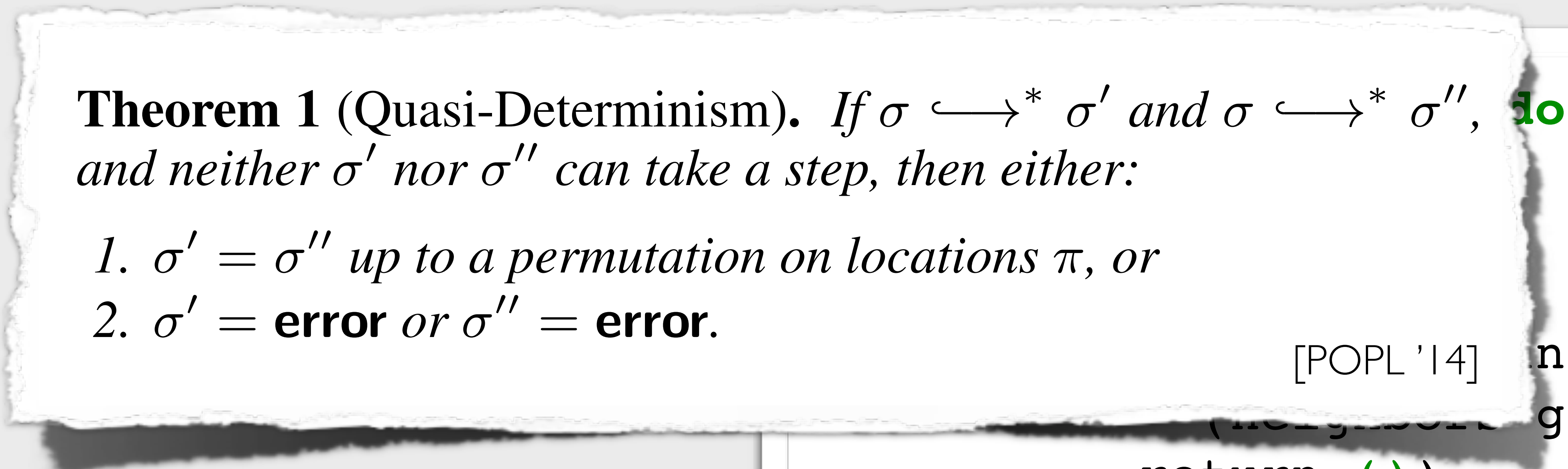
```
do
nsert v seen)
g node)
                return ())
  insert startNode seen
  quiesce h
  freeze seen
```

```
[(Book,1),(Shoes,1)]
[(B    1)]
[(S   ,1)]
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

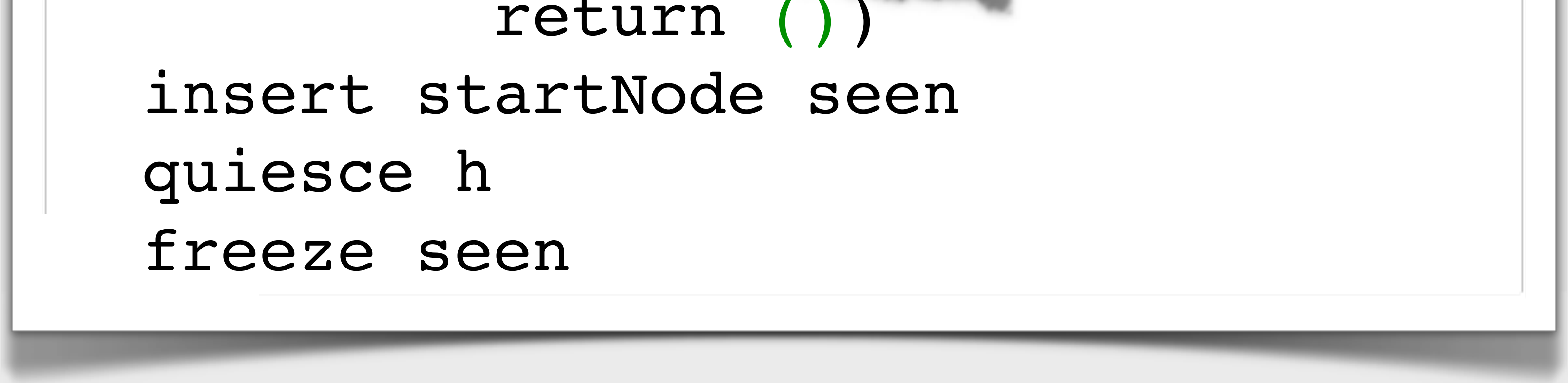
Two possible outcomes: either the same final value or an exception



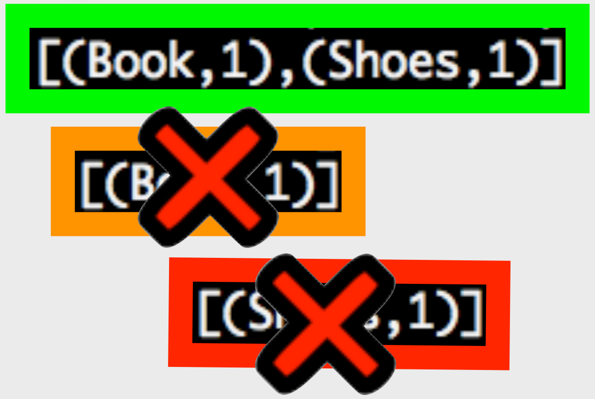
Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

1. *$\sigma' = \sigma''$ up to a permutation on locations π, or*
2. *$\sigma' =$ **error** or $\sigma'' =$ **error**.*

[POPL '14]



```
                                   do
                      nsert v seen)
                      g node)
         return ())
insert startNode seen
quiesce h
freeze seen
```



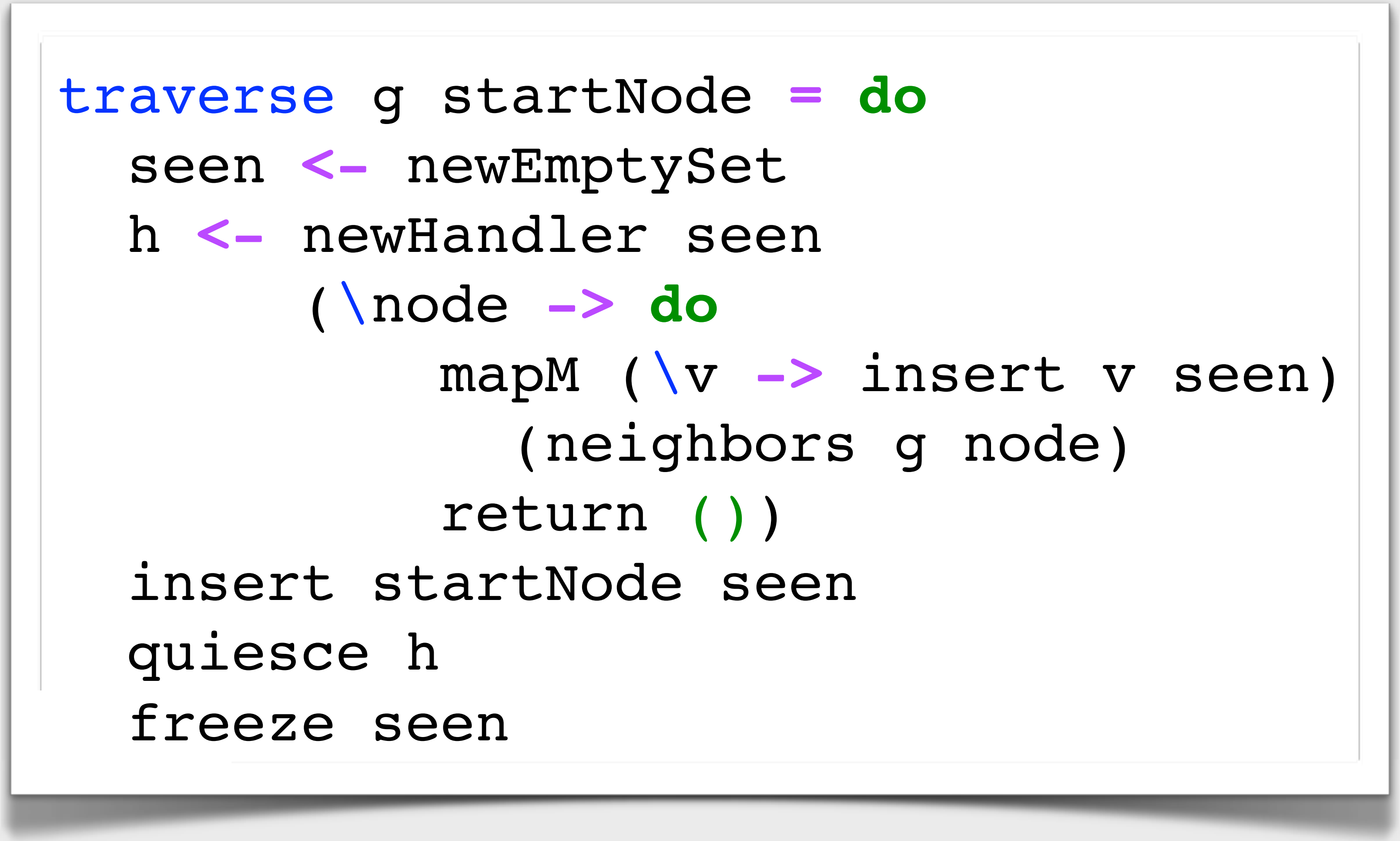
```
[(Book,1),(Shoes,1)]
[(B 1)]
[(S ,1)]
```

or error.

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception

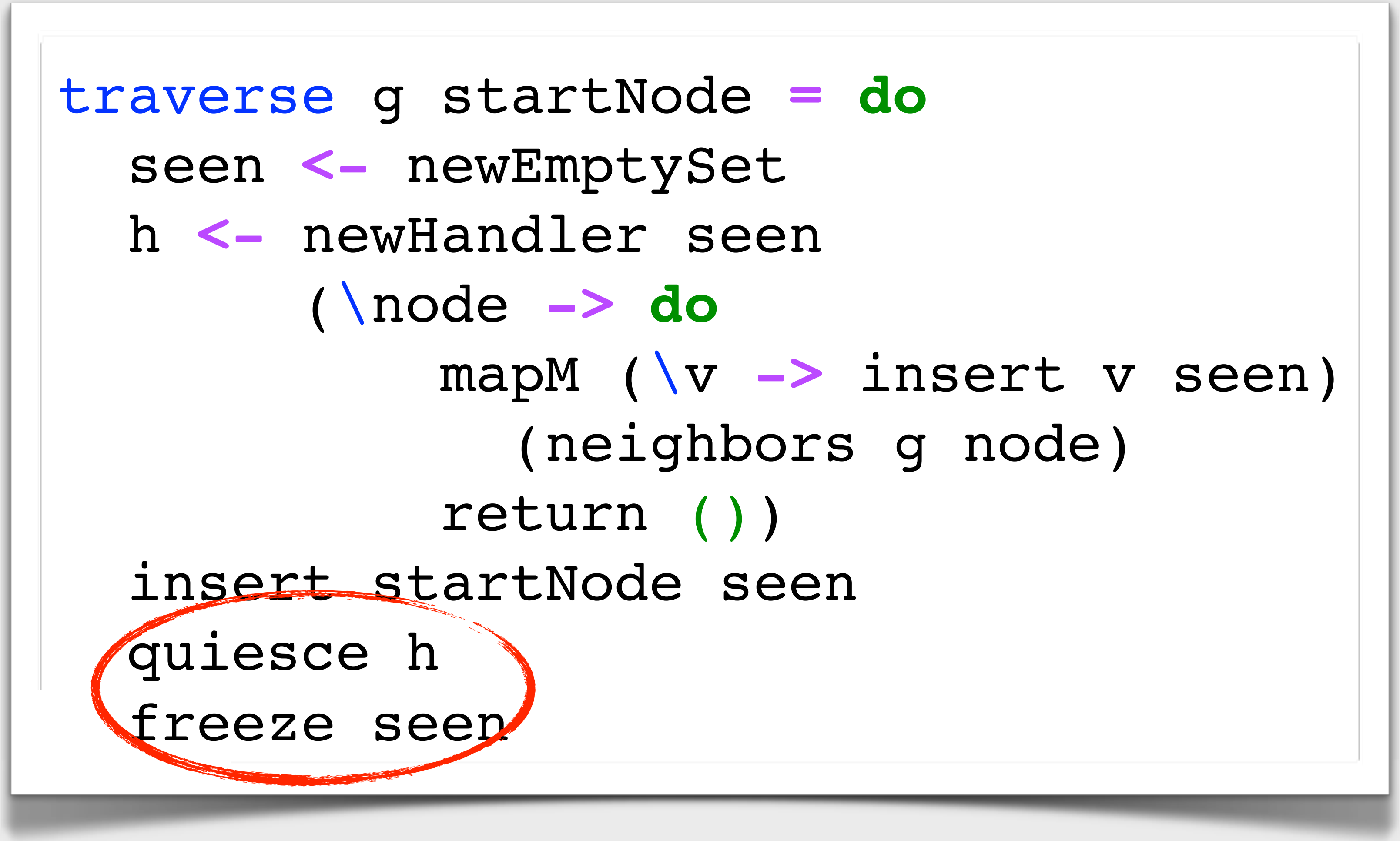


```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception



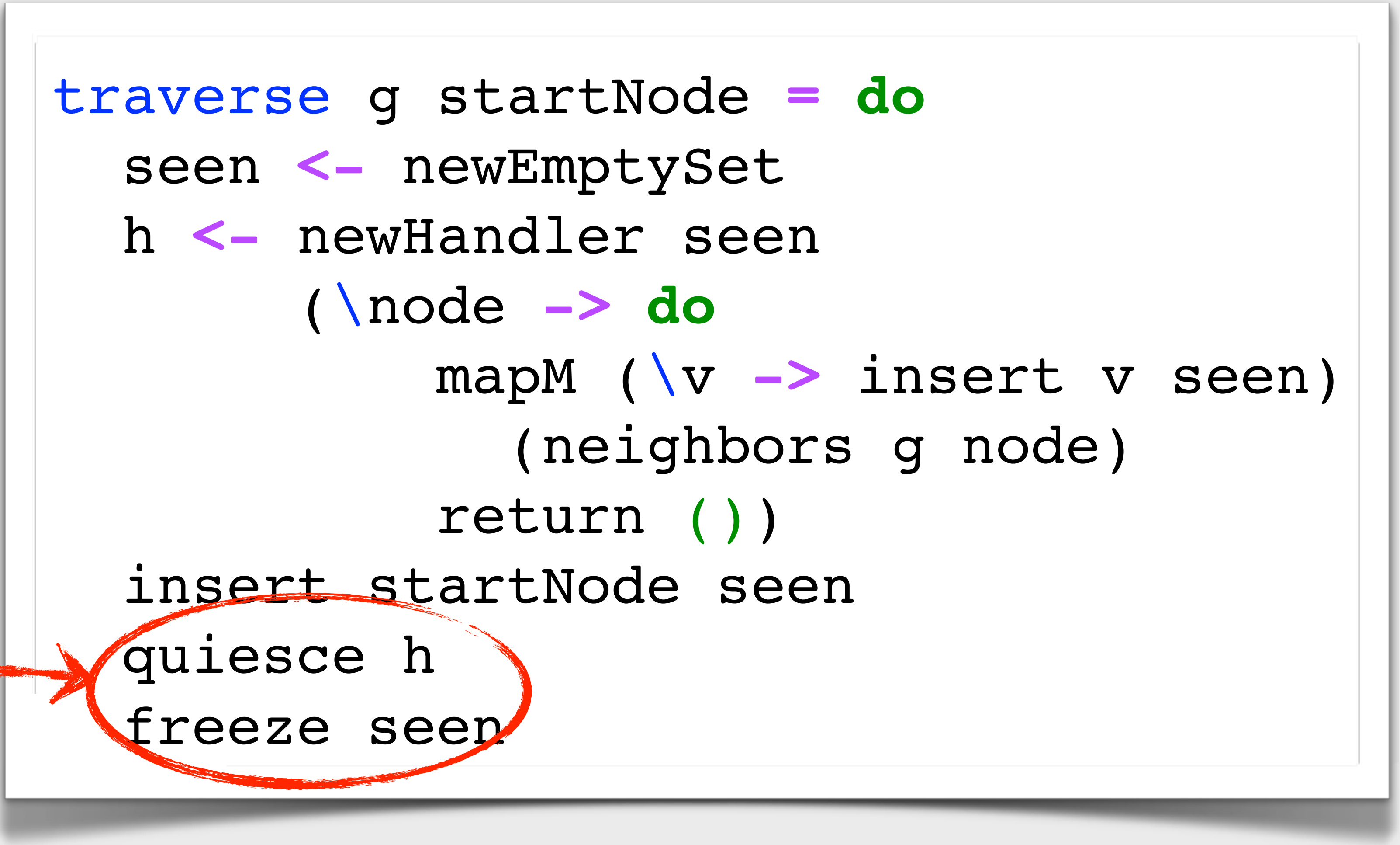
```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception

Let the system handle this for us:



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

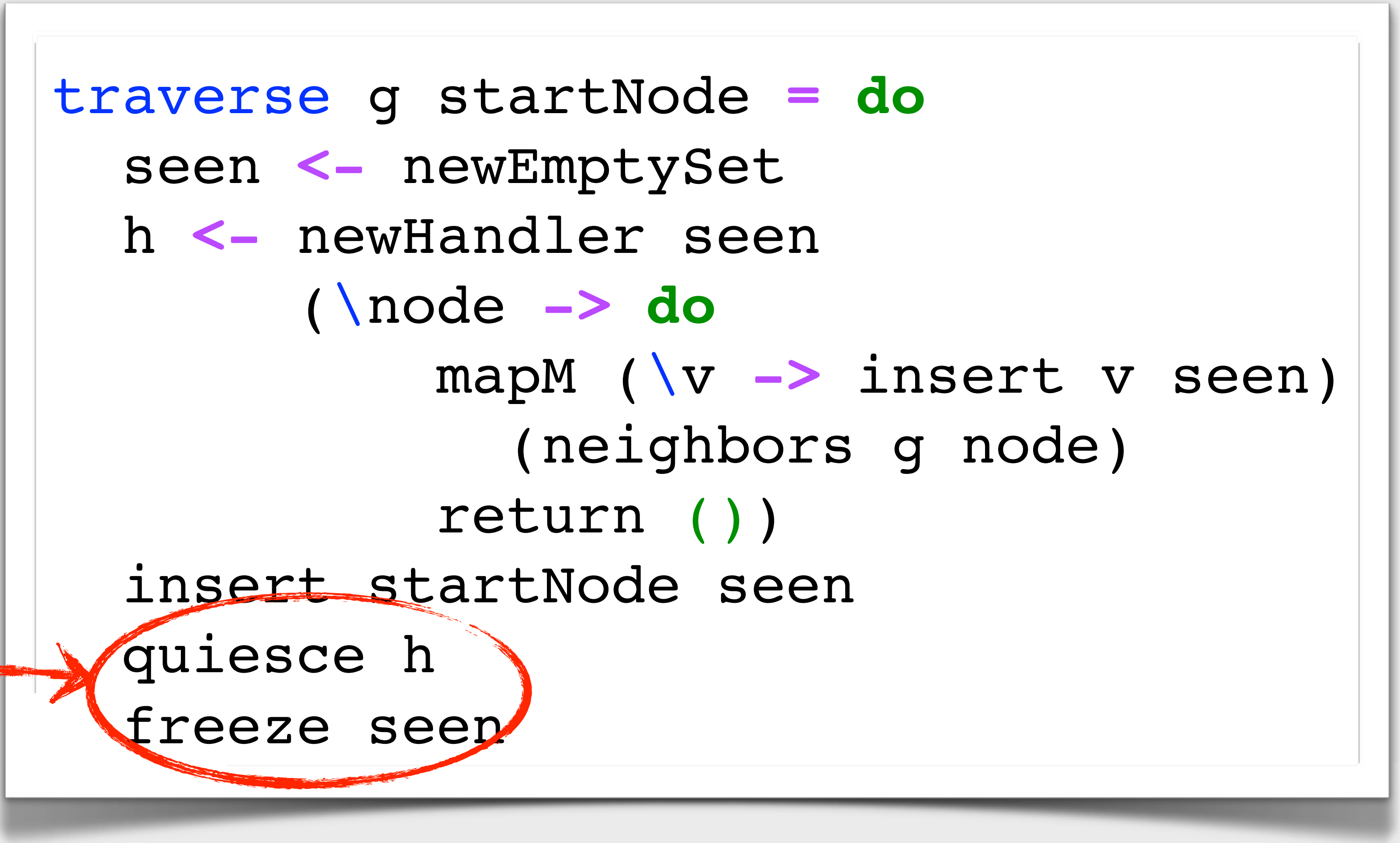
`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

Two possible outcomes: either the same final value or an exception

Let the system handle this for us:

`runParThenFreeze`



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

LVish

a Haskell library for parallel programming with LVars

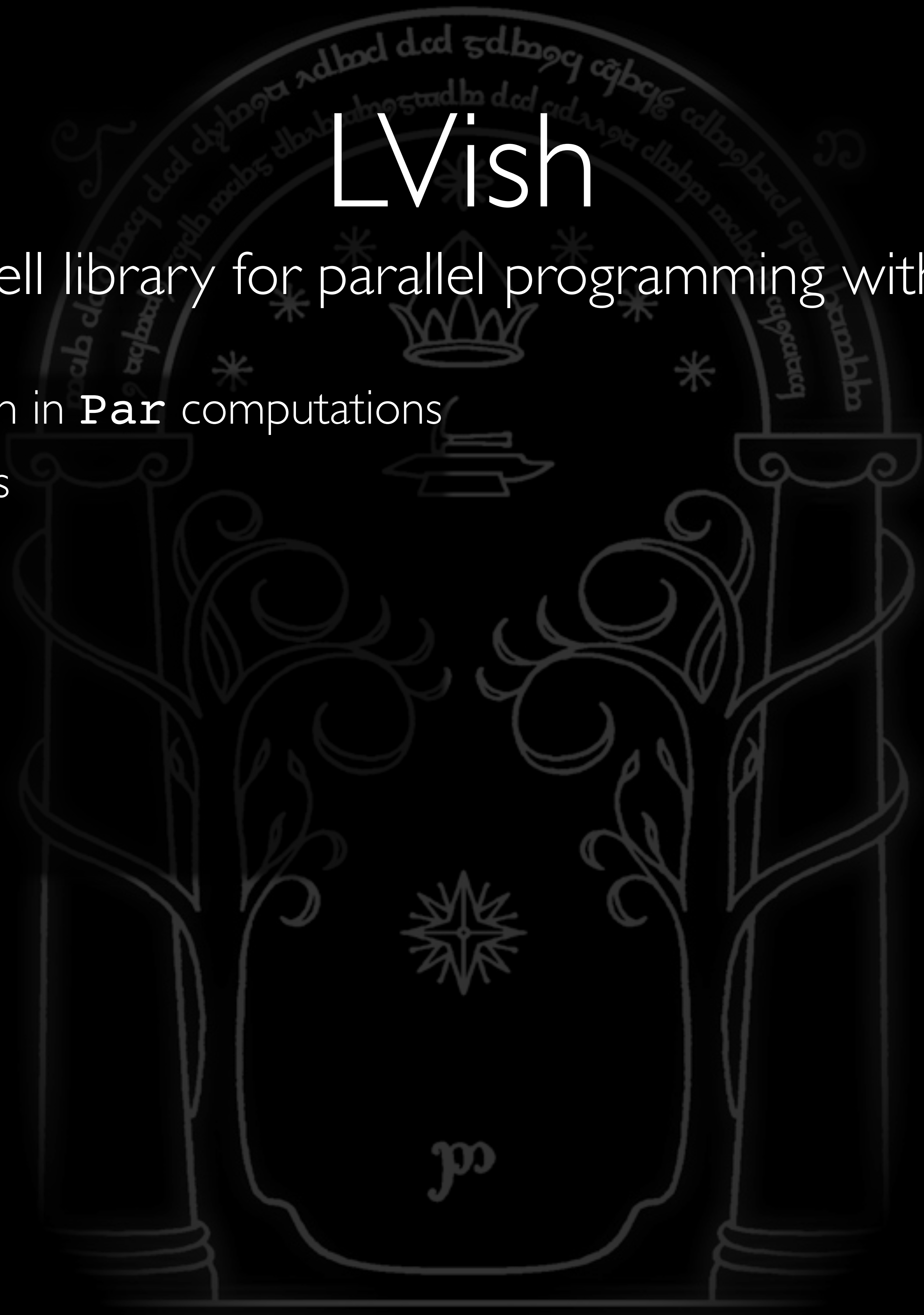
LVish

a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

LVish

a Haskell library for parallel programming with LVars



LVar operations run in `Par` computations

Lightweight threads

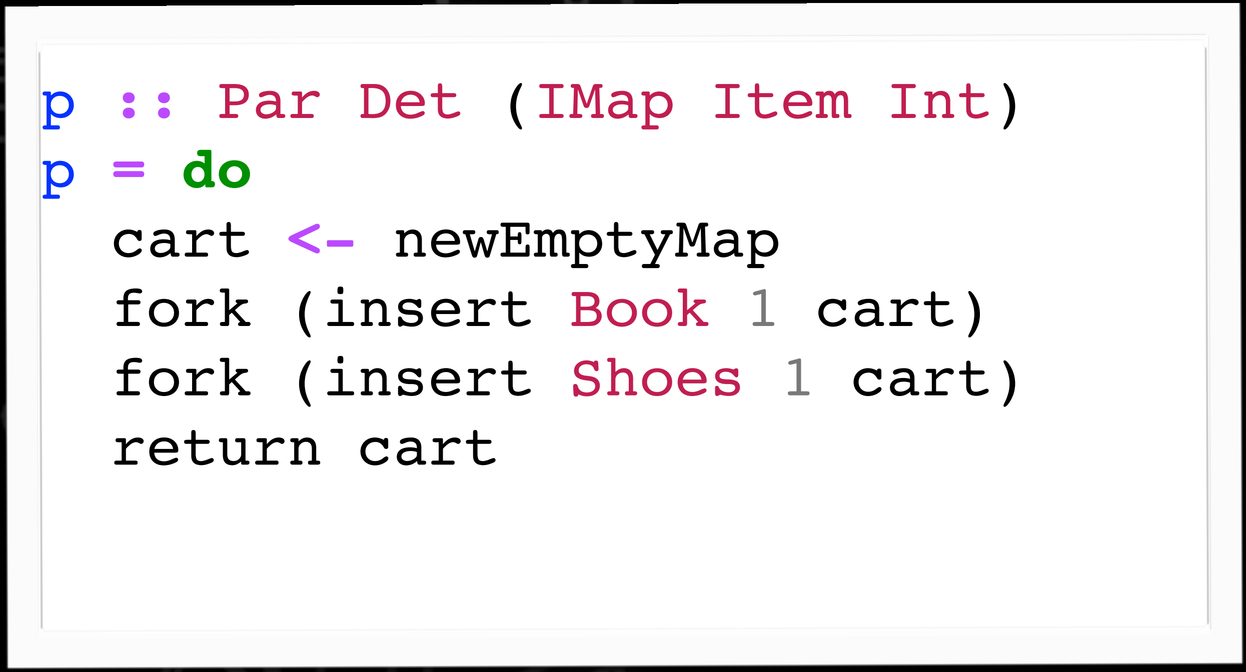
LVish

a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

Lightweight threads

`Par` computations indexed by *effect level*



```
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart
```

LVish

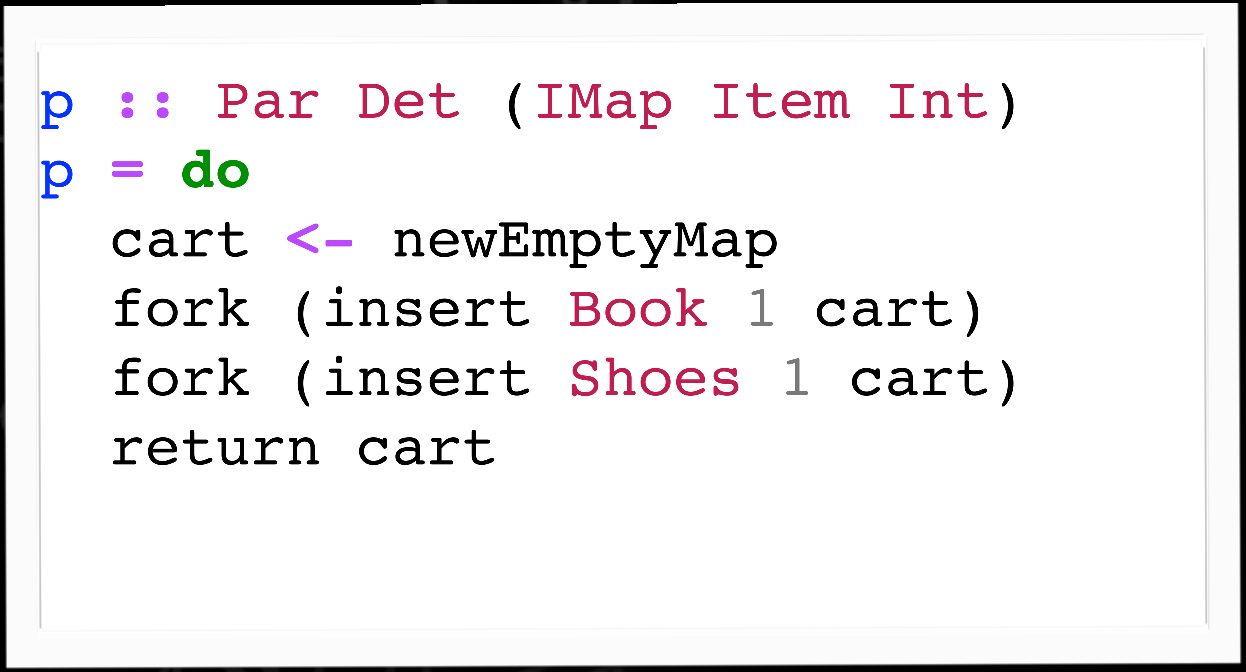
a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

Lightweight threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom



```haskell
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart
```

LVish

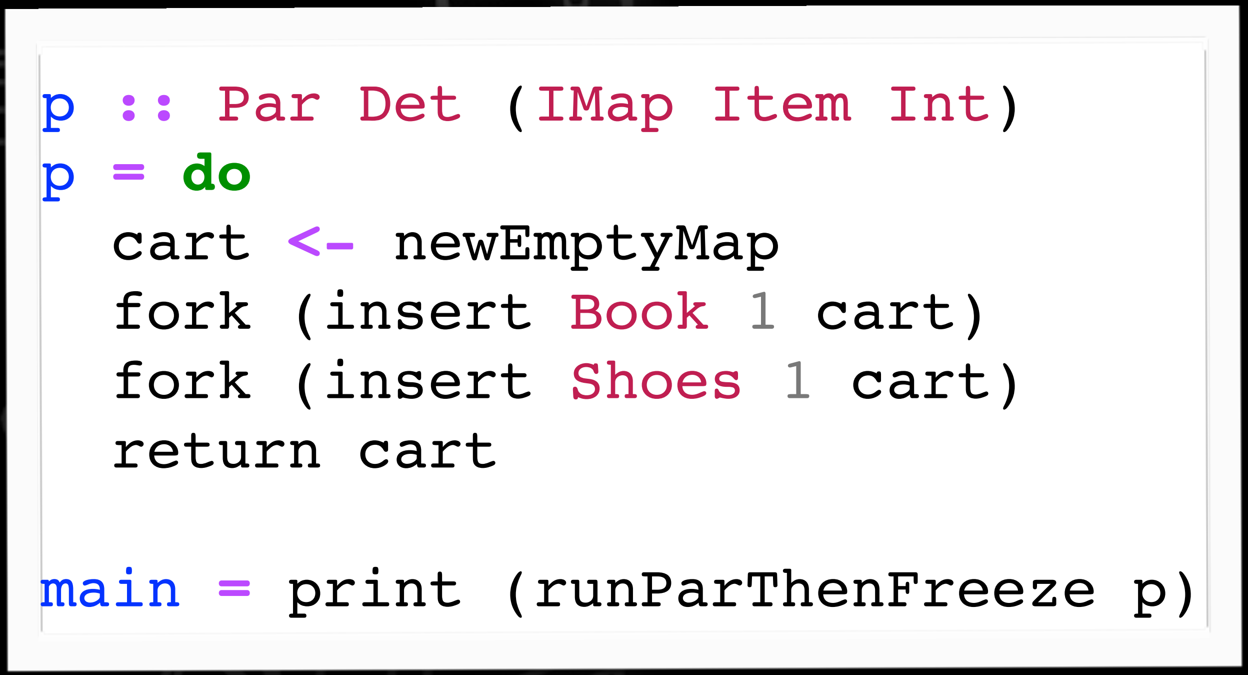
a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

Lightweight threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom



```haskell
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

LVish

a Haskell library for parallel programming with LVars

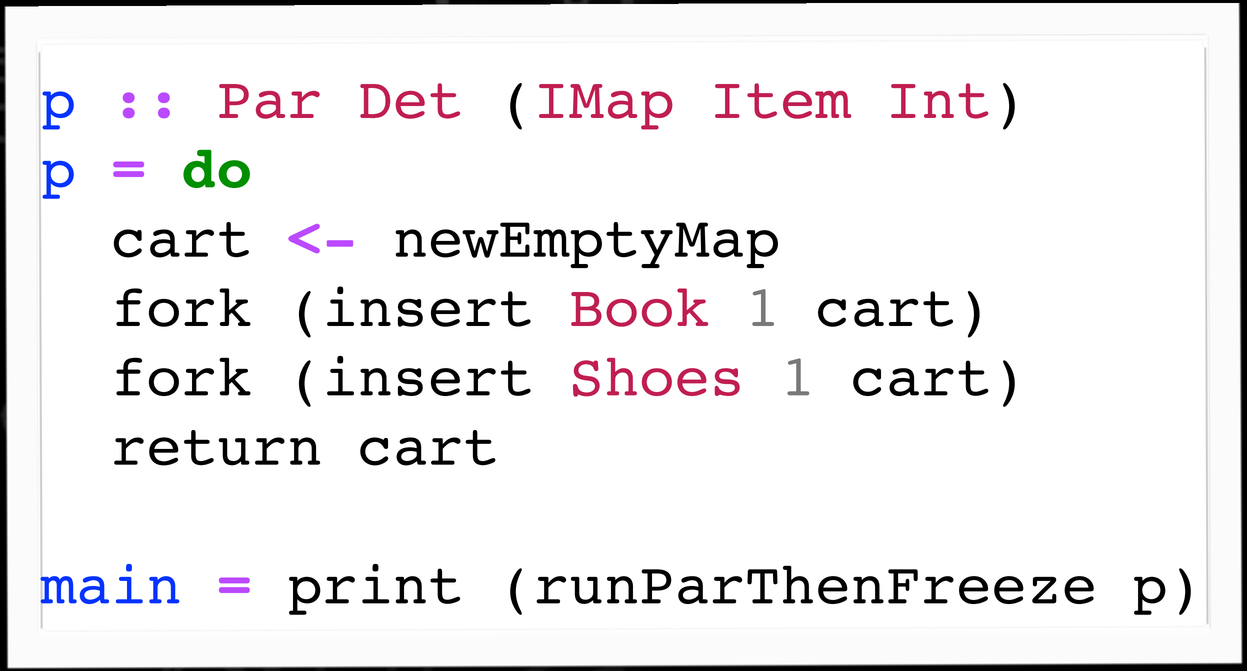
LVar operations run in `Par` computations

Lightweight threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

Efficient lock-free sets, maps, *etc.*



```haskell
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

LVish

a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

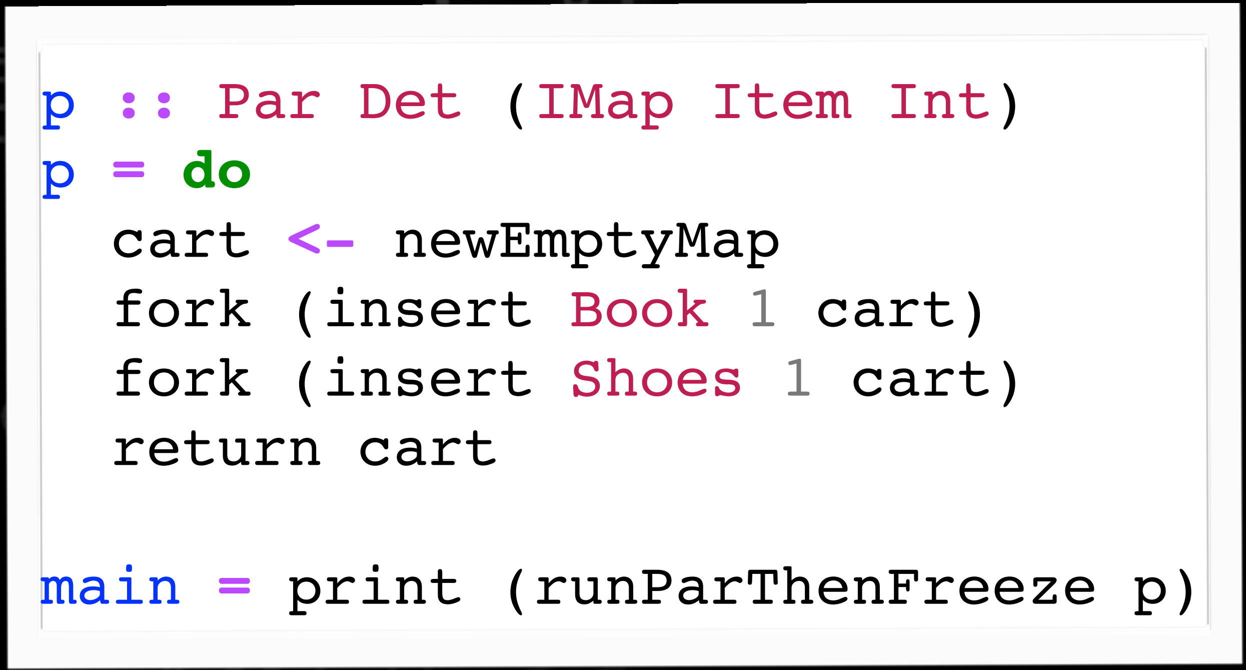
Lightweight threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

Efficient lock-free sets, maps, *etc.*

Implement your own LVars, too



```
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

LVish

a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations

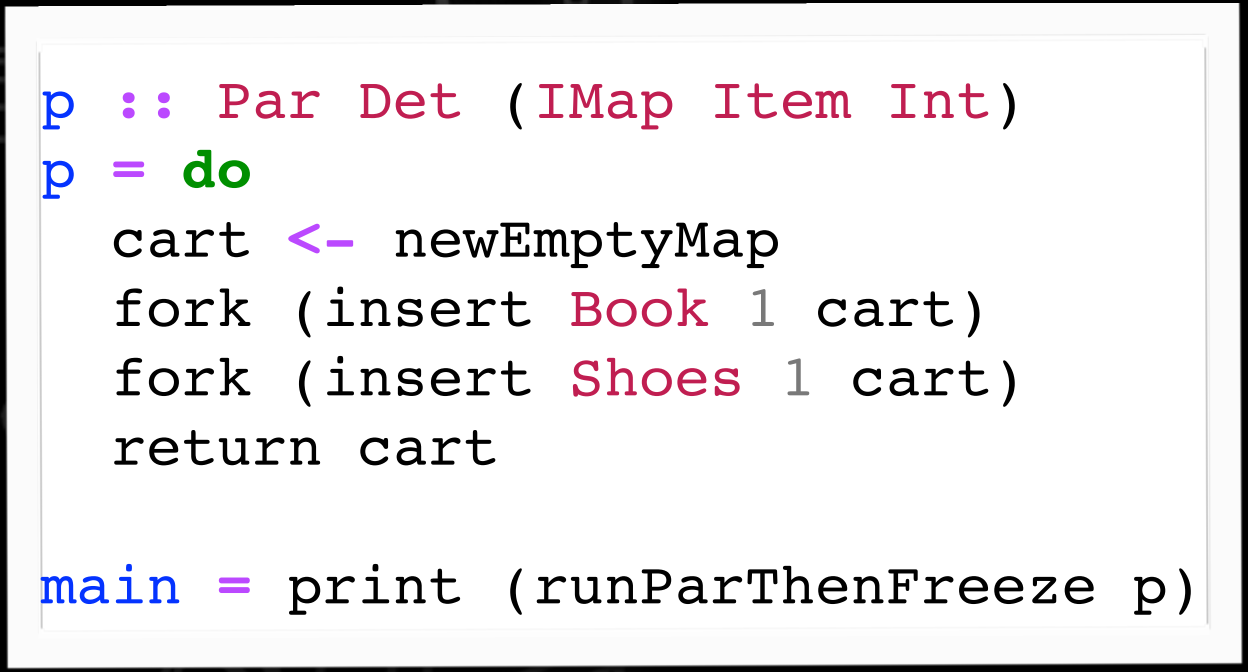
Lightweight threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

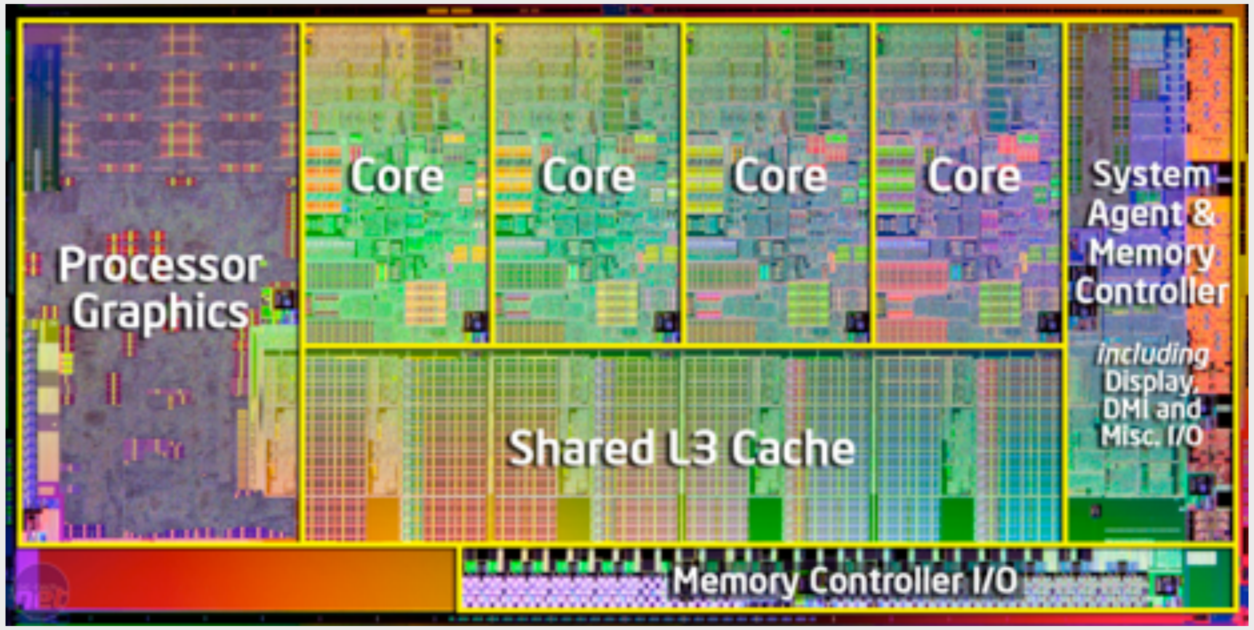
Efficient lock-free sets, maps, *etc.*

Implement your own LVars, too

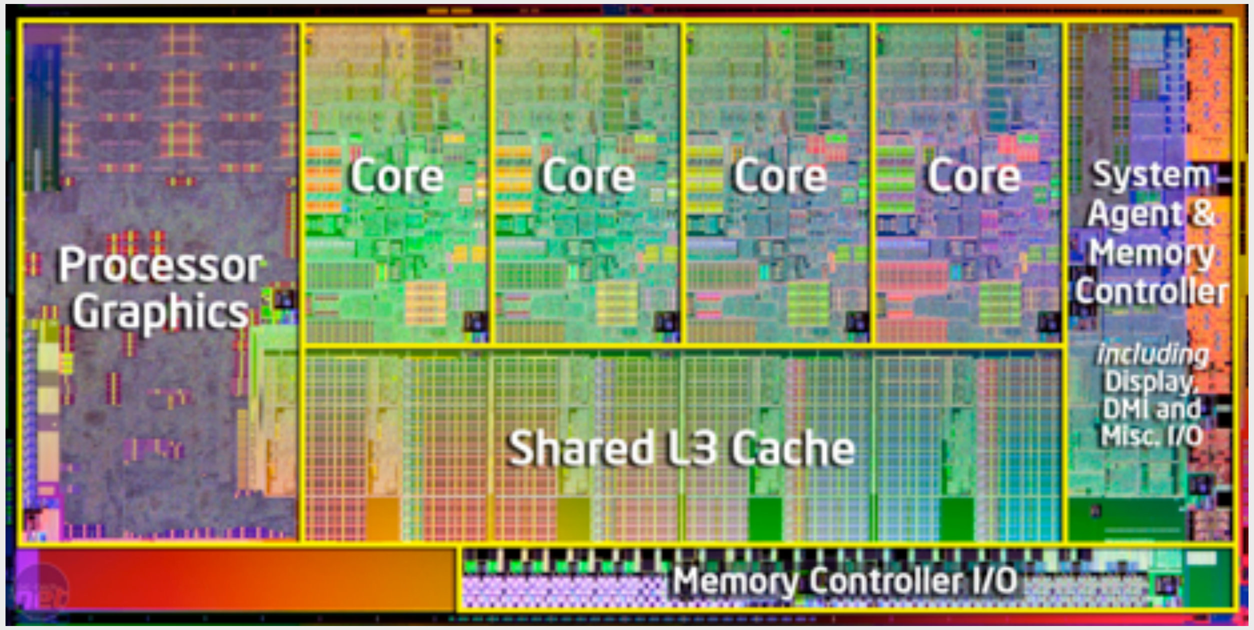


```haskell
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

`hackage.haskell.org/package/lvish`

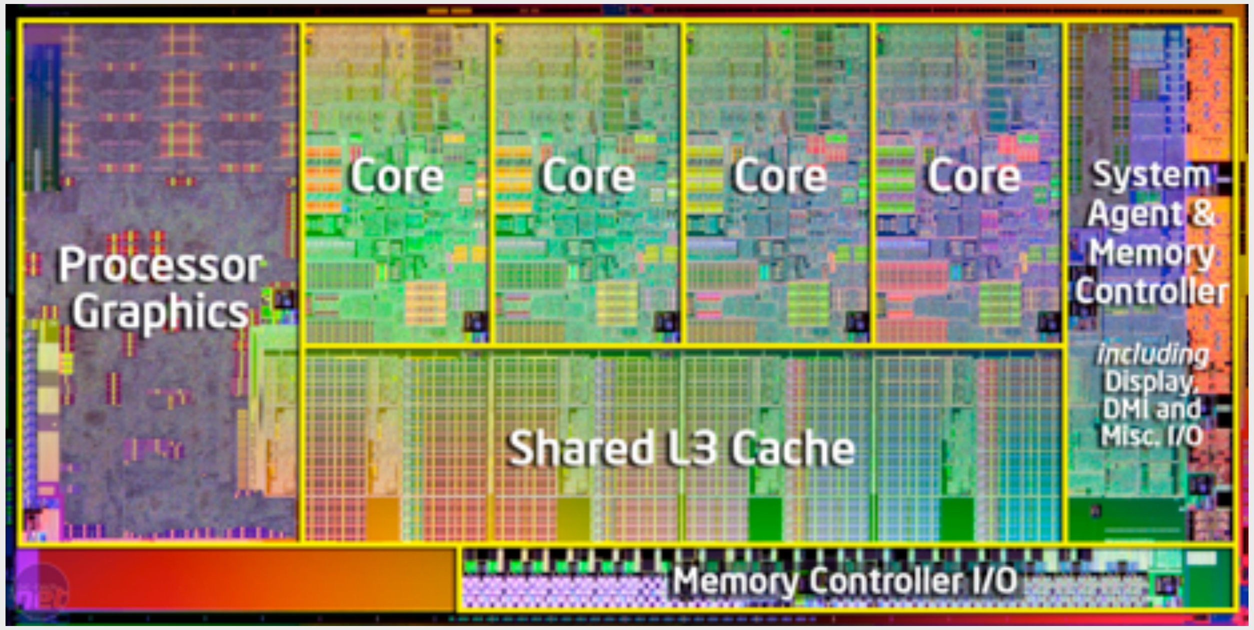


Deterministic Parallel Programming



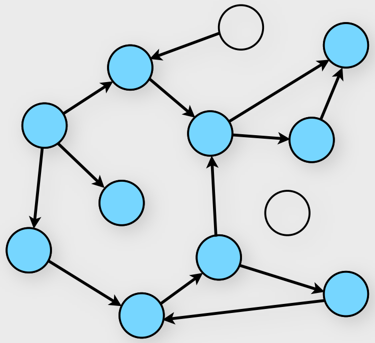
(observably)

Deterministic Parallel Programming

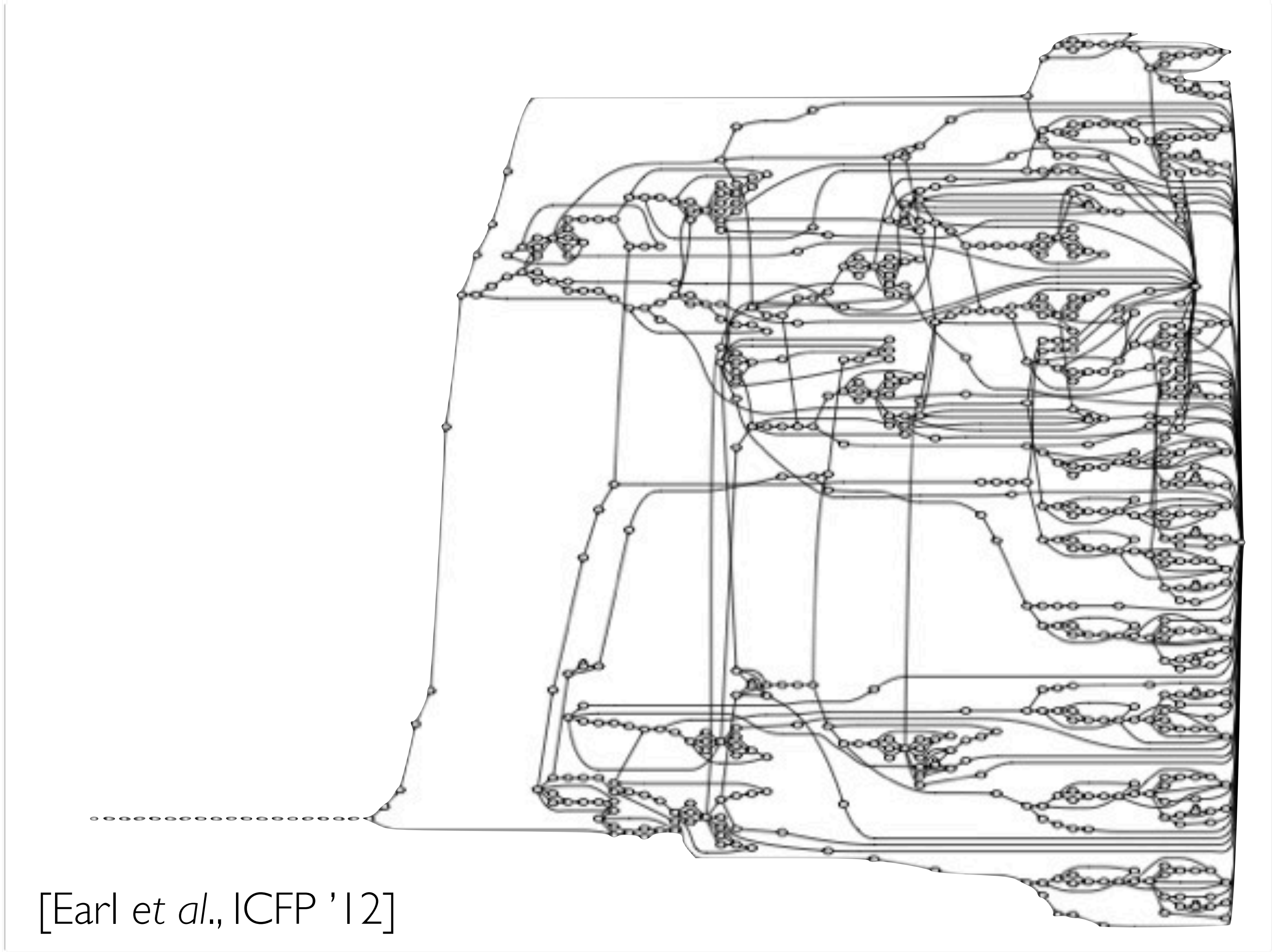


(observably) *(irregular)*

Deterministic Parallel Programming



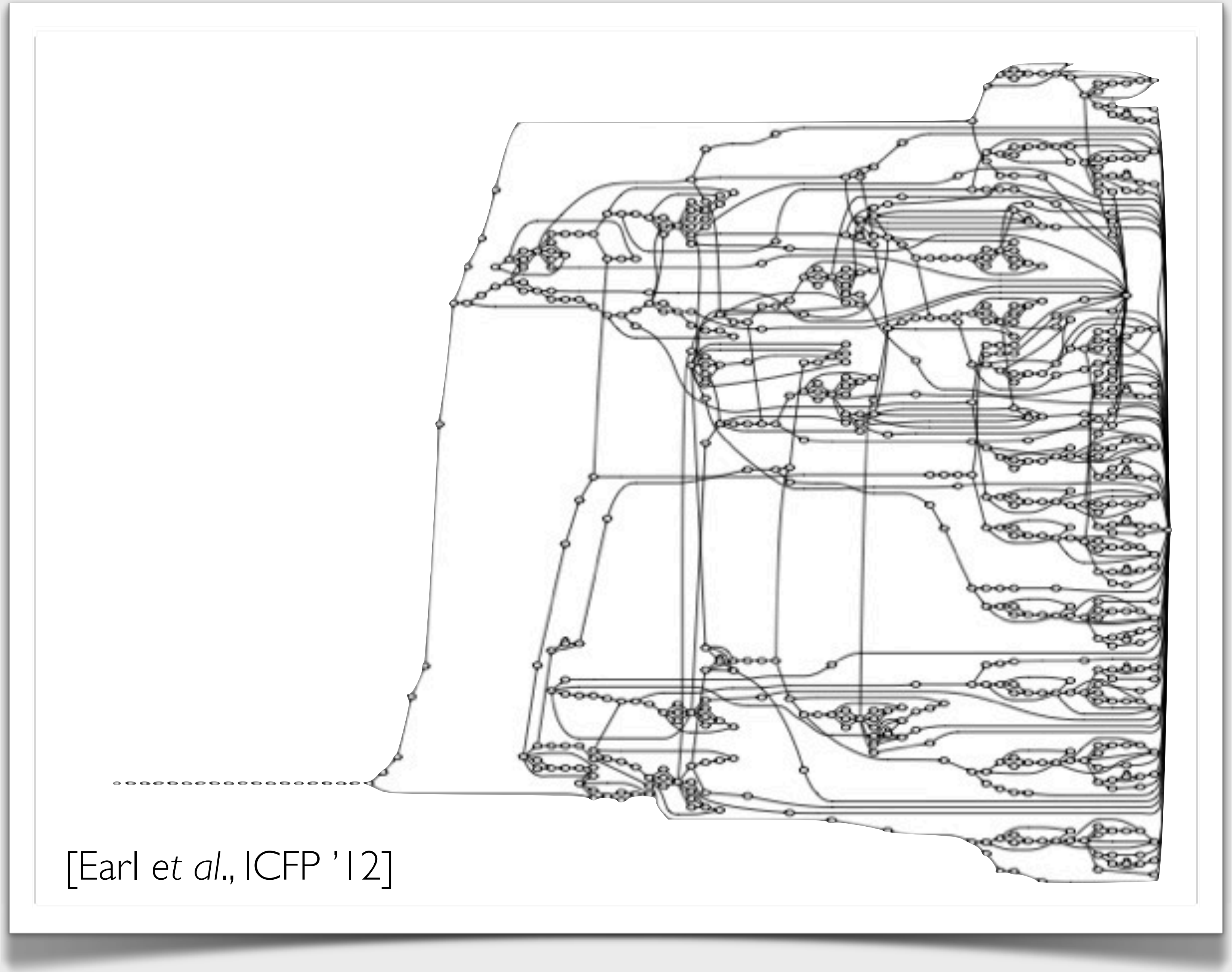
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]



[Earl *et al.*, ICFP '12]

Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

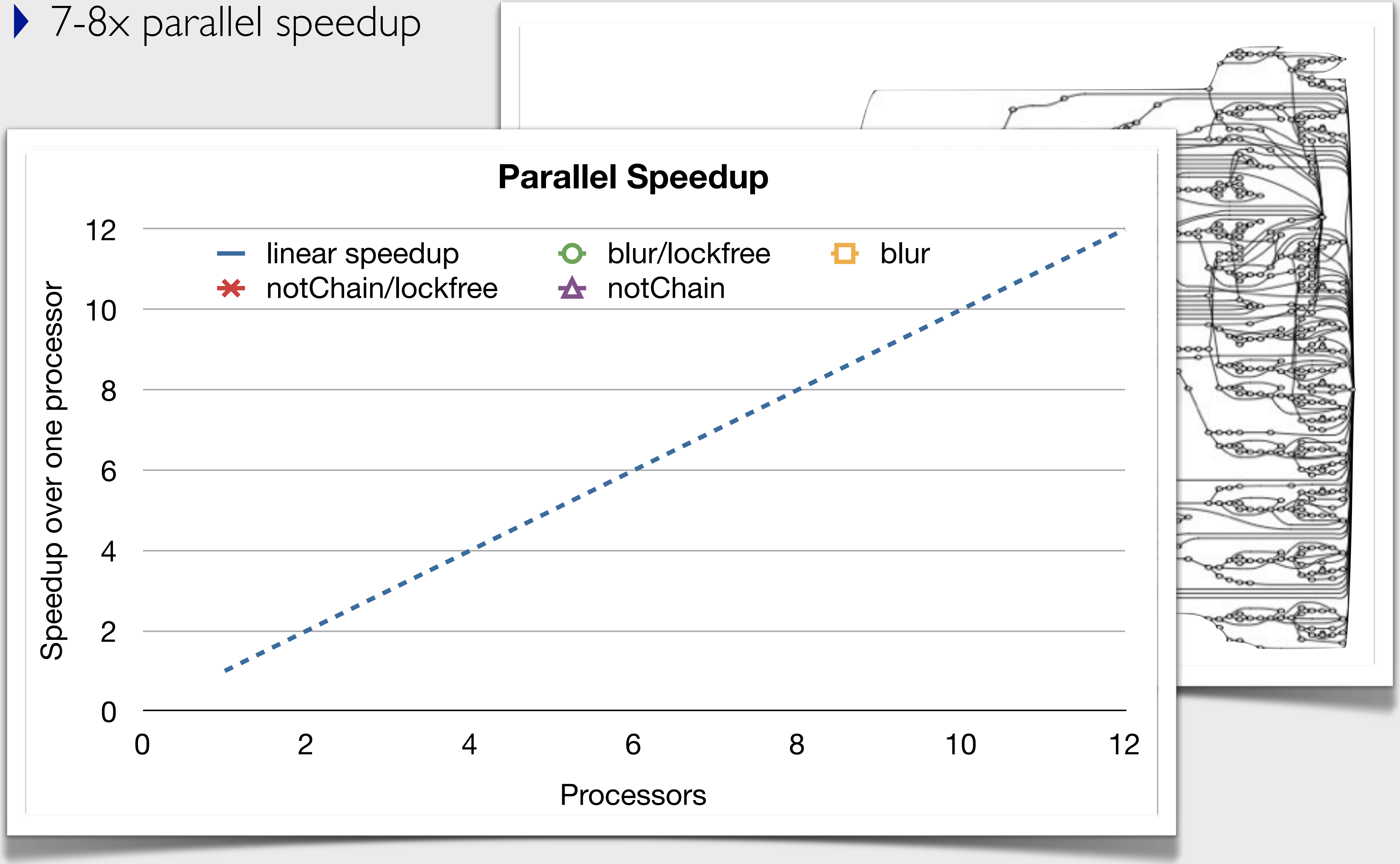
- up to 20x speedup, *even on one core*, from not having to copy data



[Earl *et al.*, ICFP '12]

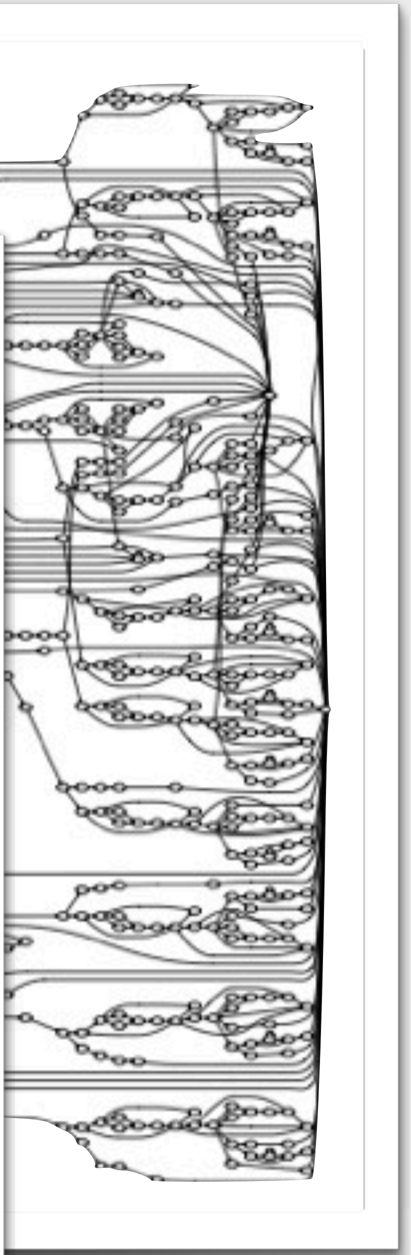
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

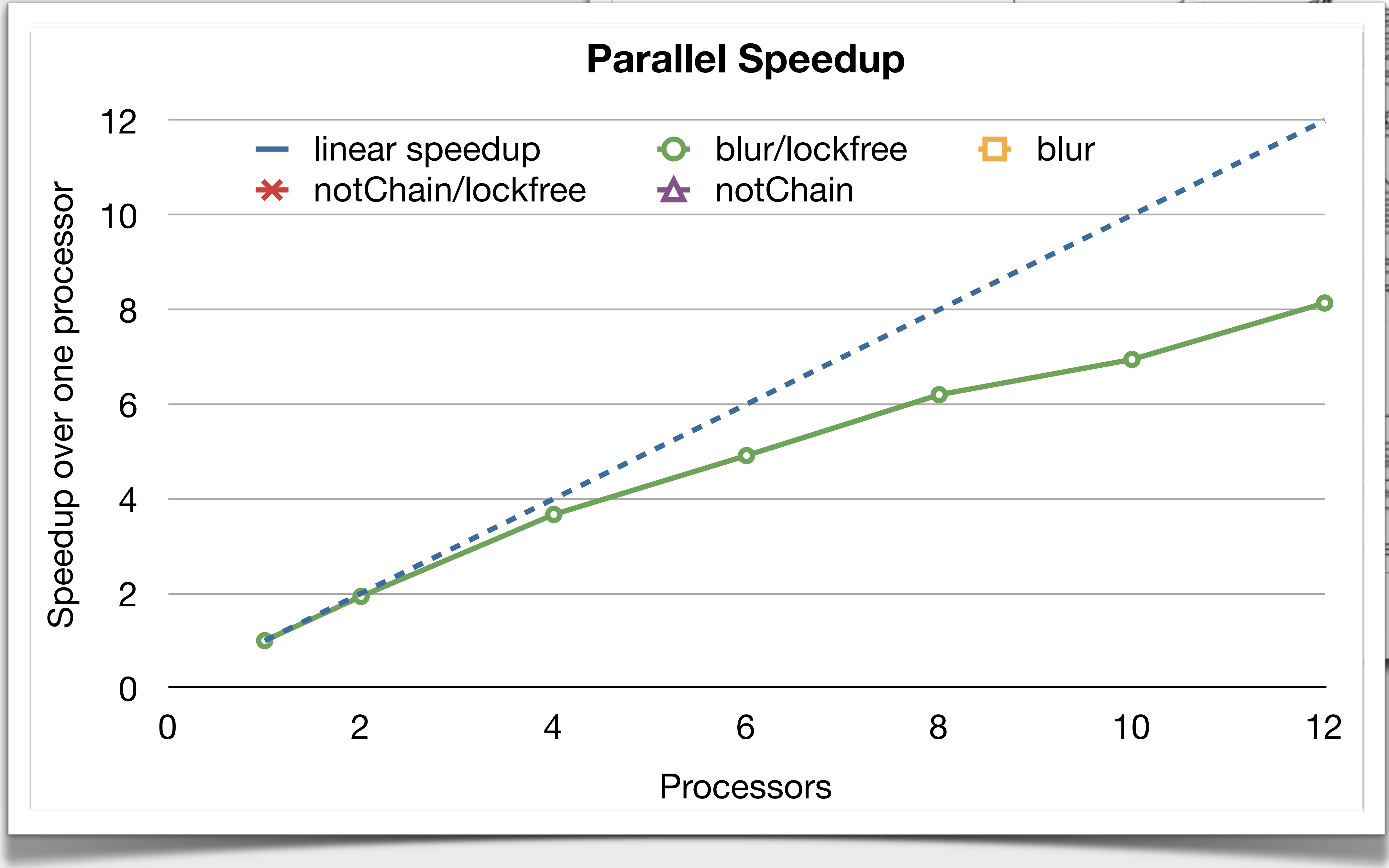
- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup



Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

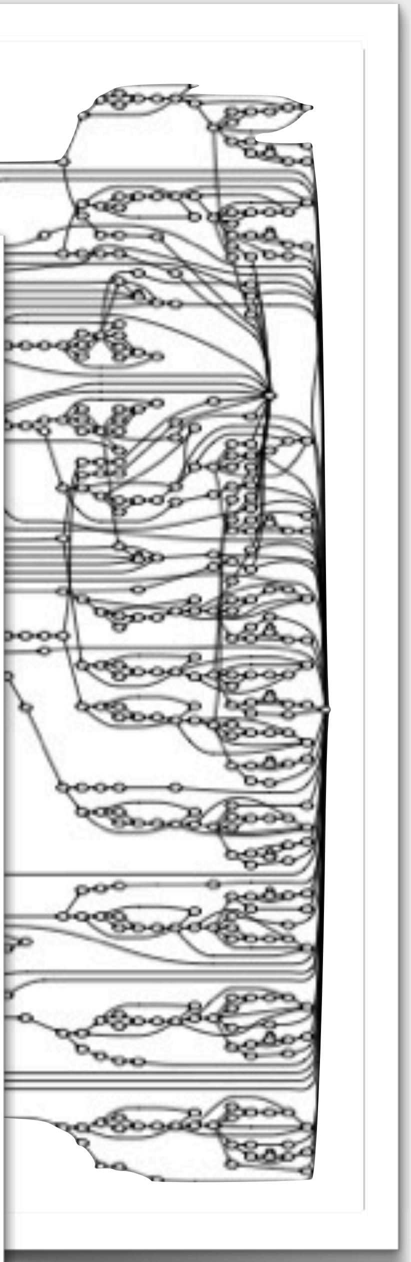
- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup

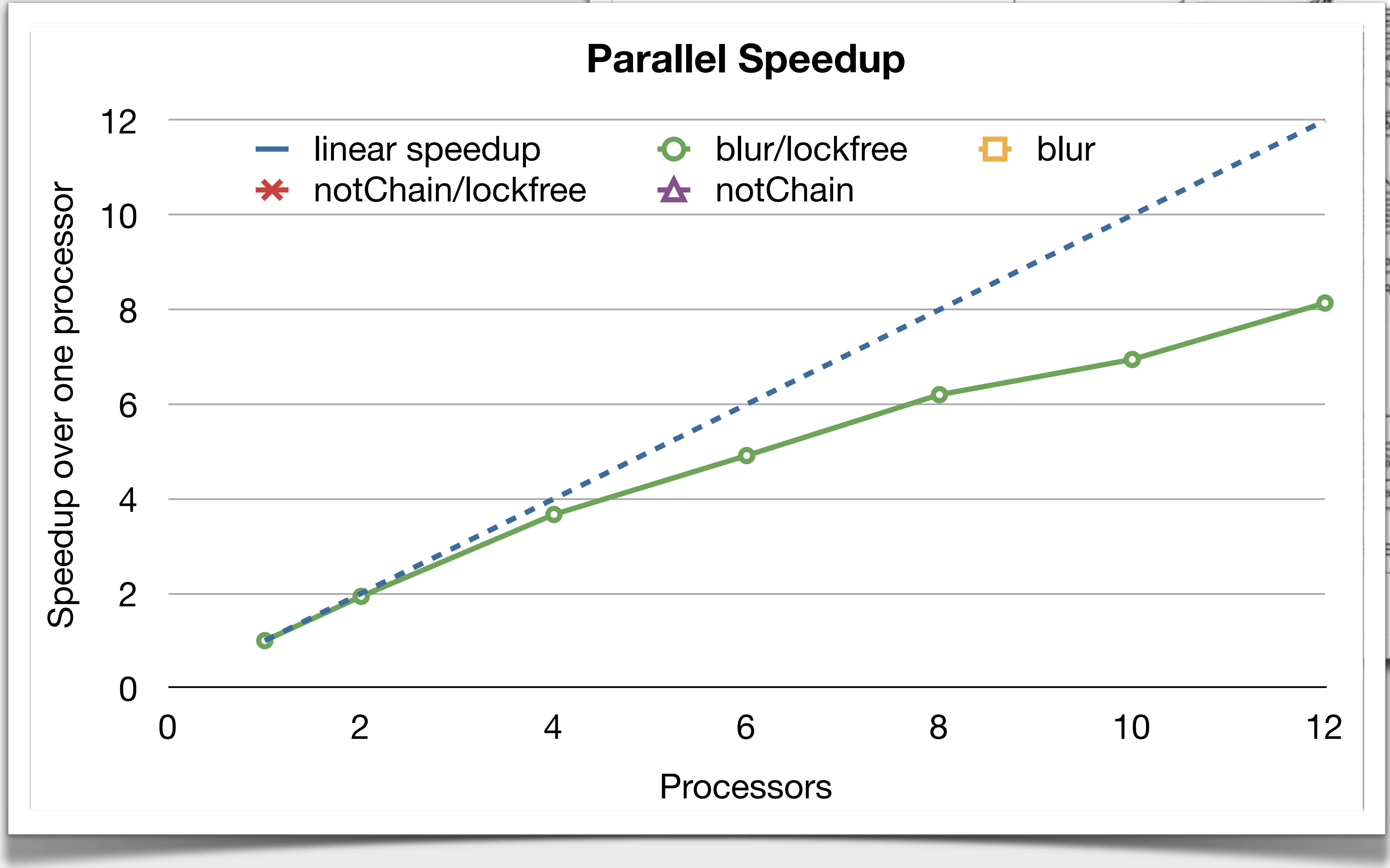




Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

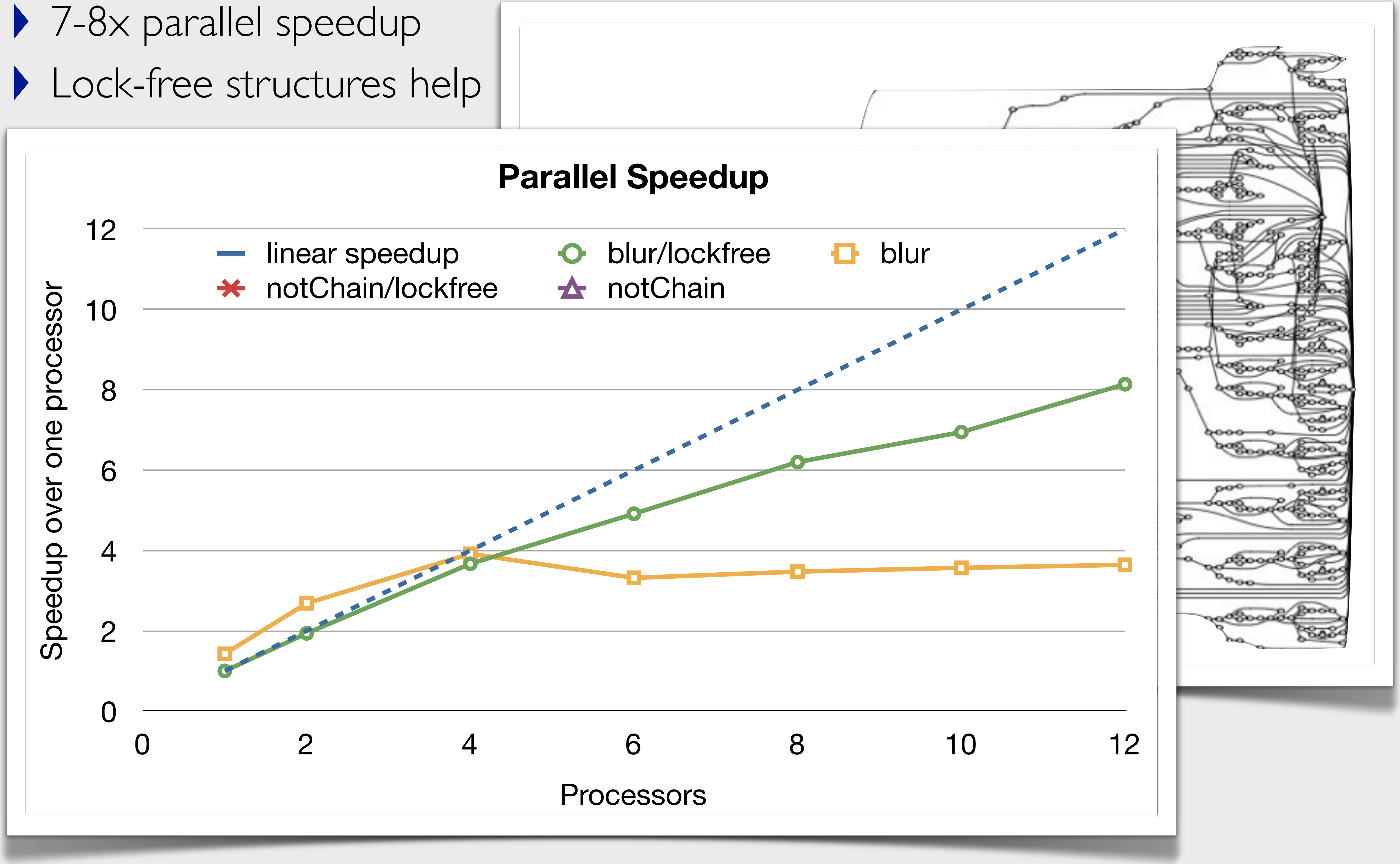
- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup
- Lock-free structures help





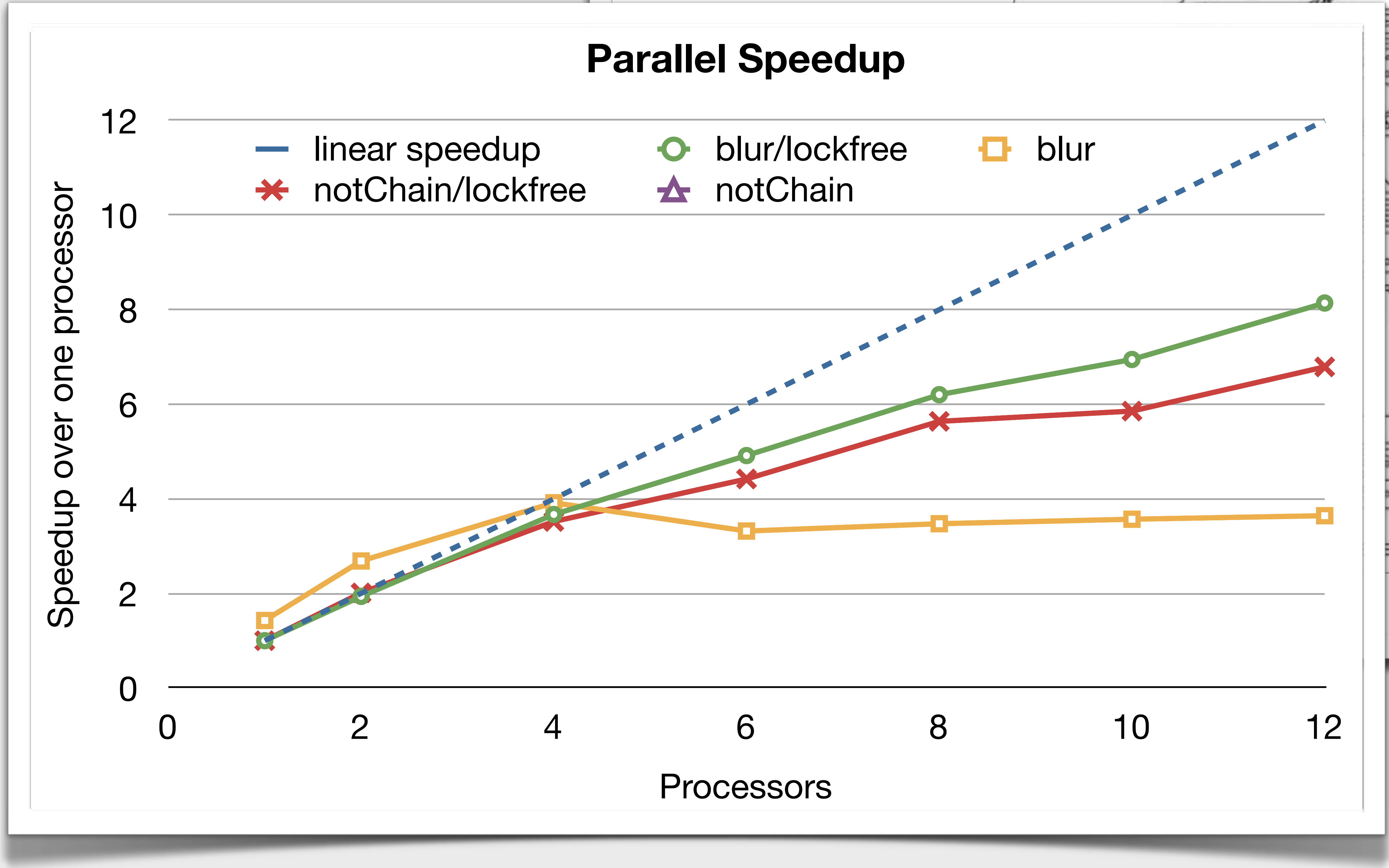
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup
- Lock-free structures help



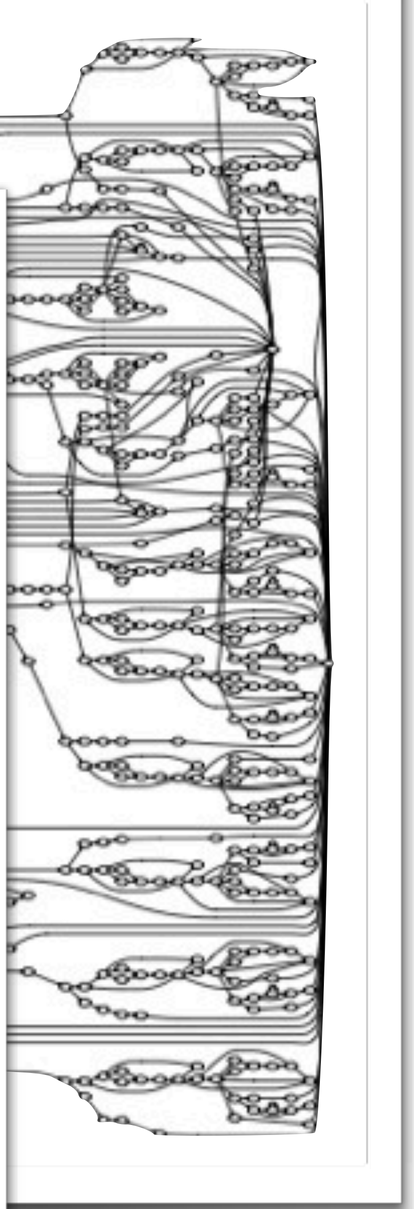
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

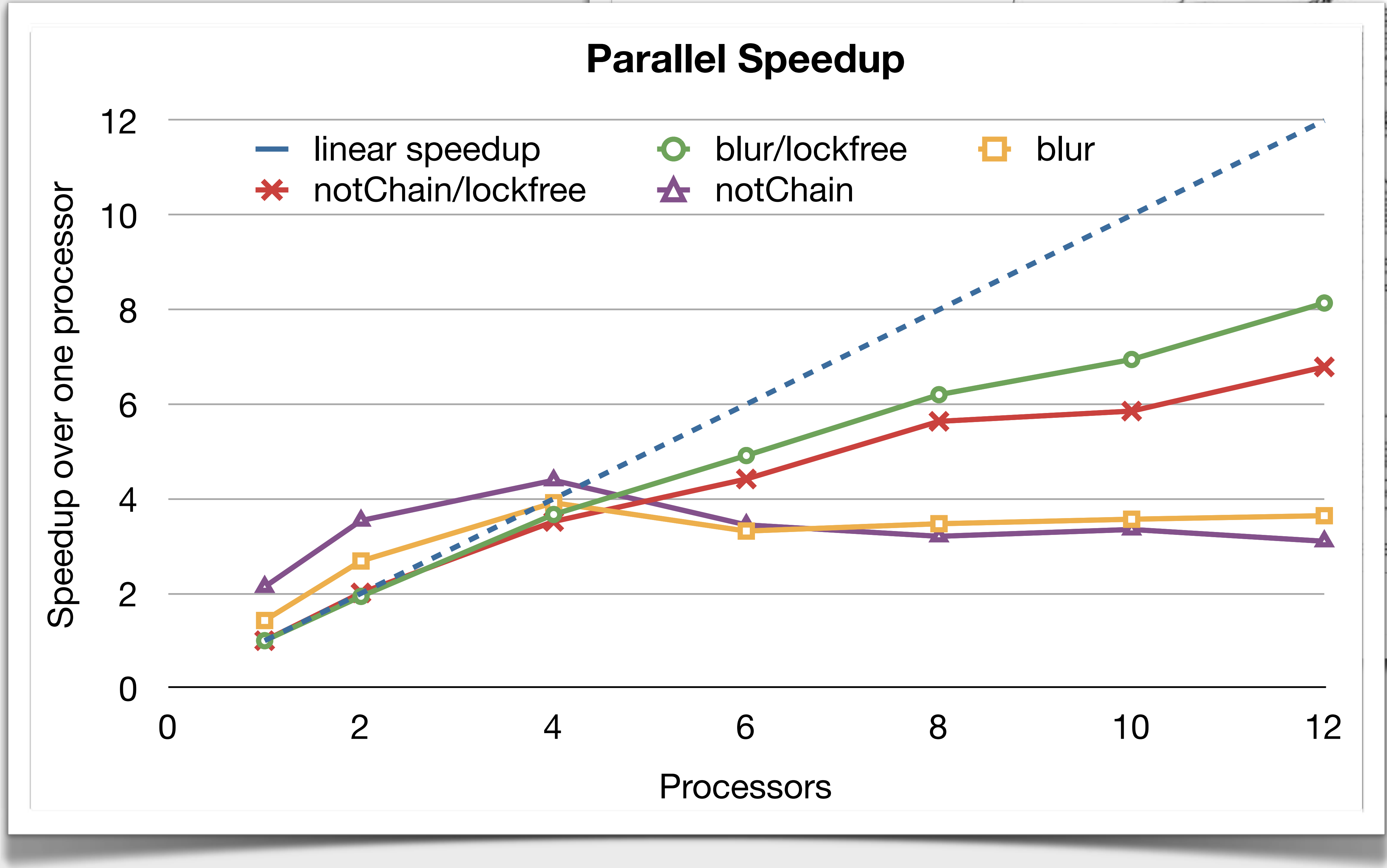
- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup
- Lock-free structures help



Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

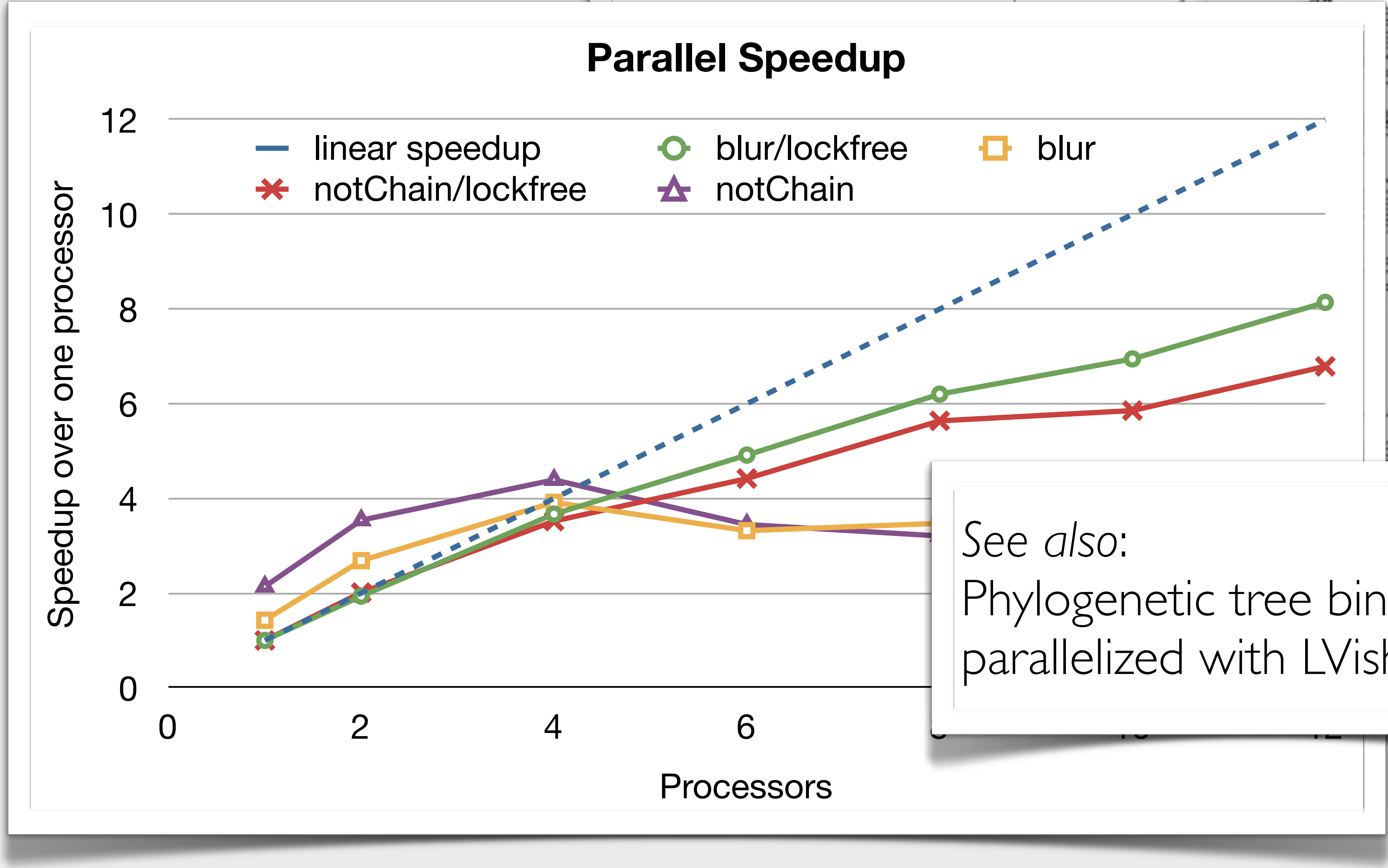
- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup
- Lock-free structures help

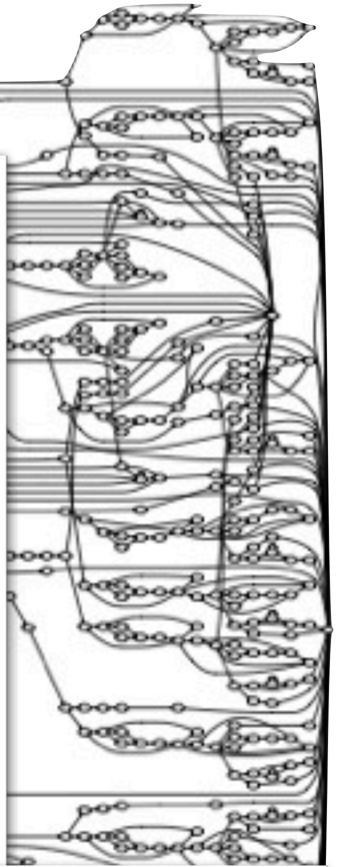




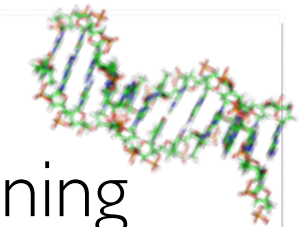
Case study: *k*-CFA static analysis parallelized with LVish [POPL '14]

- up to 20x speedup, *even on one core*, from not having to copy data
- 7-8x parallel speedup
- Lock-free structures help

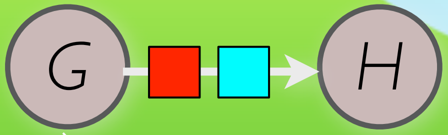




See *also*:
Phylogenetic tree binning parallelized with LVish [PLDI '14]



LVars and LVish across the landscape:

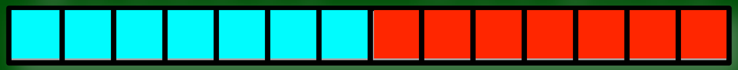


Kahn process networks

single-assignment

imperative disjoint

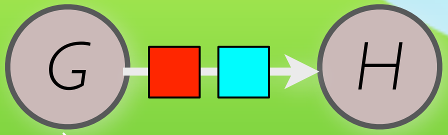
```
g(left)   h(rght)
```



λ-calculus

```
f  (g  x)
   (h  y)
```

LVars and LVish across the landscape:

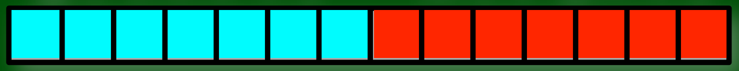


Kahn process networks

single-assignment

imperative disjoint

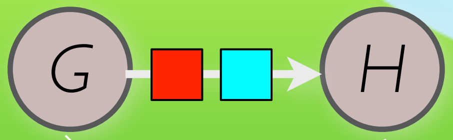
```
g(left)  h(rght)
```



λ-calculus ✓

```
f  (g  x)
   (h  y)
```

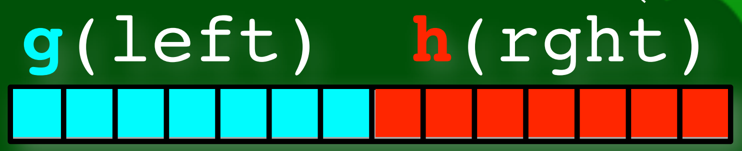
LVars and LVish across the landscape:



Kahn process networks

single-assignment ✓✓

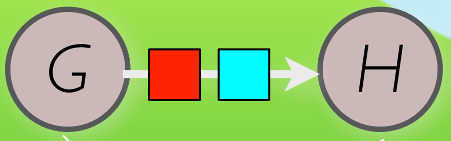
imperative disjoint



λ-calculus ✓

```
f (g x)
  (h y)
```

LVars and LVish across the landscape:

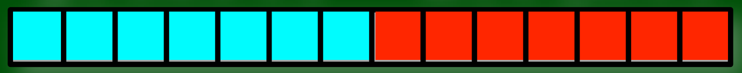


Kahn process networks ✓

single-assignment ✓✓

imperative disjoint

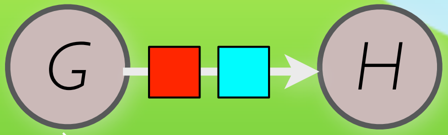
```
g(left)   h(rght)
```



λ-calculus ✓

```
f  (g  x)
   (h  y)
```

LVars and LVish across the landscape:

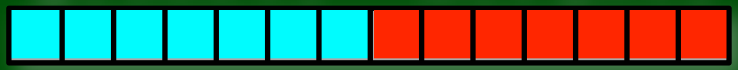


✓ Kahn process networks

✓✓ single-assignment

imperative disjoint

```
g(left)   h(rght)
```

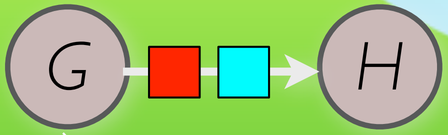


✓ λ-calculus

```
f  (g  x)
   (h  y)
```

✓ quasi-det.

LVars and LVish across the landscape:



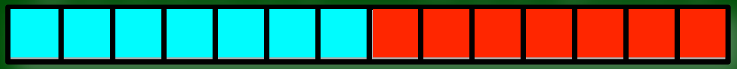
Kahn process networks ✓

single-assignment ✓✓

quasi-det. ✓

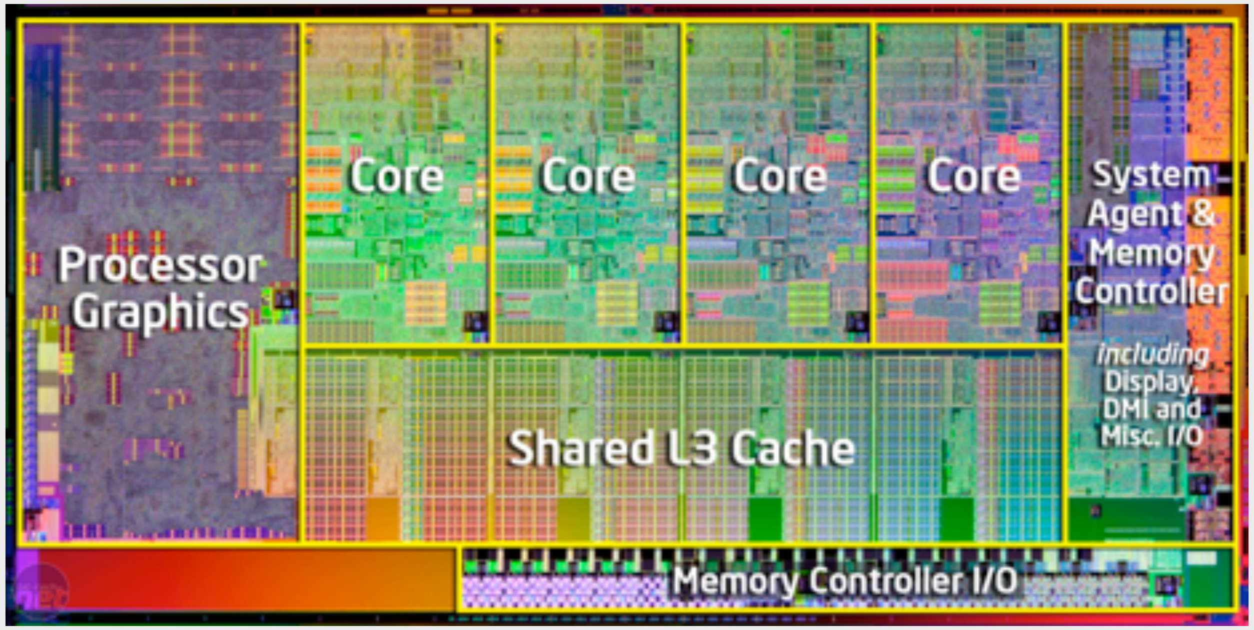
imperative disjoint ✓ [PLDI '14]

```
g(left)  h(rght)
```



λ-calculus ✓

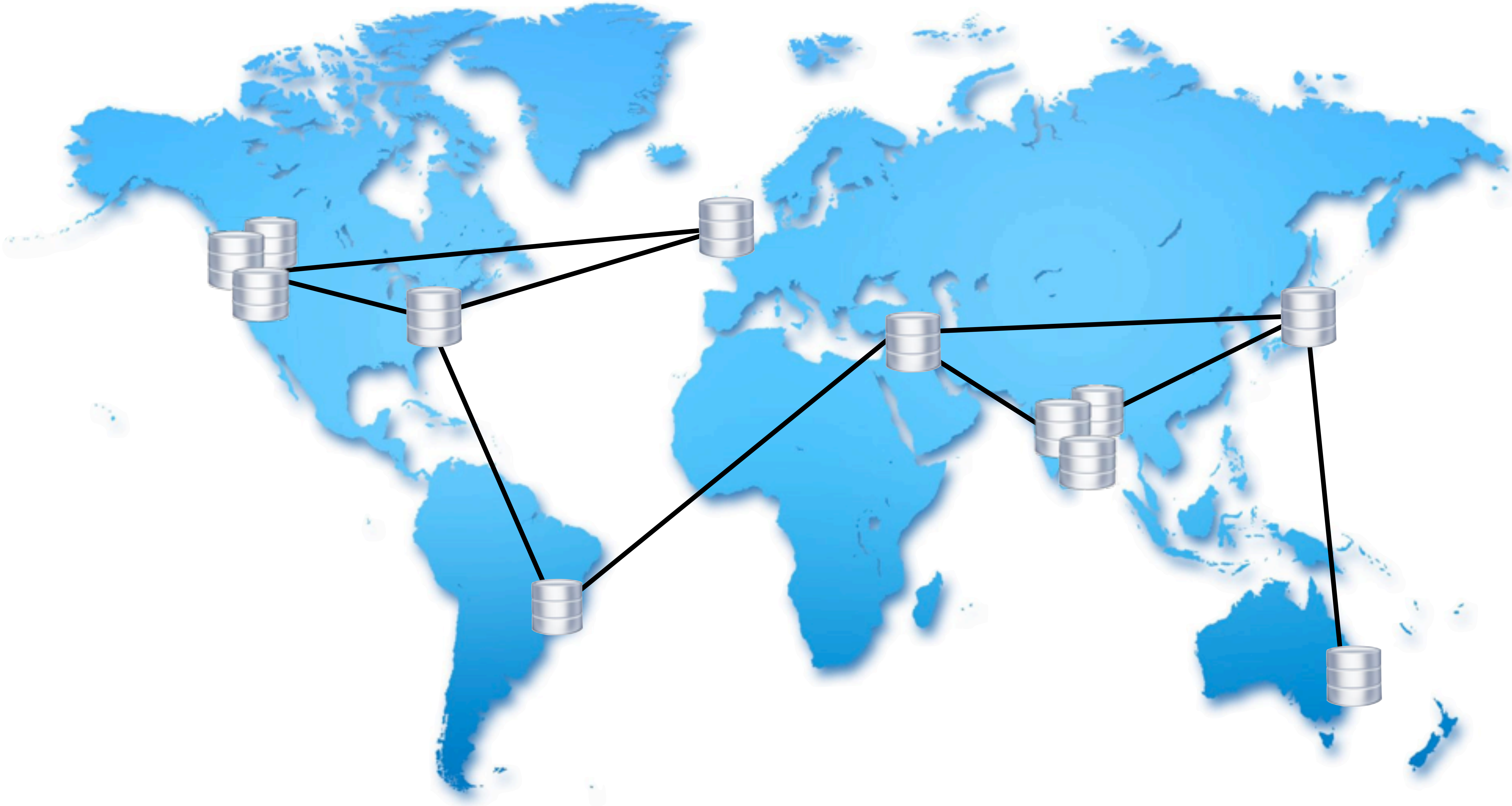
```
f (g x)
  (h y)
```

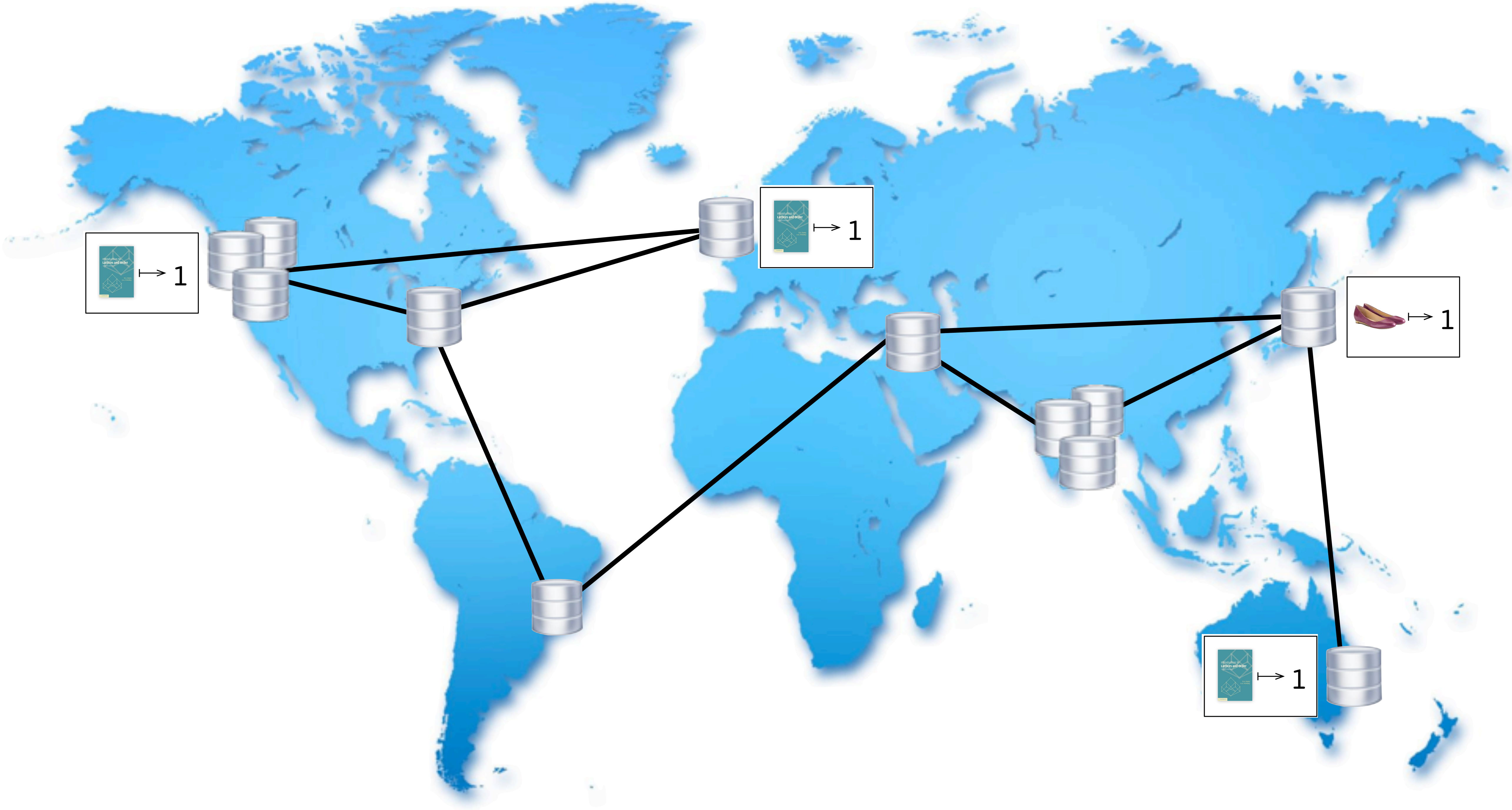


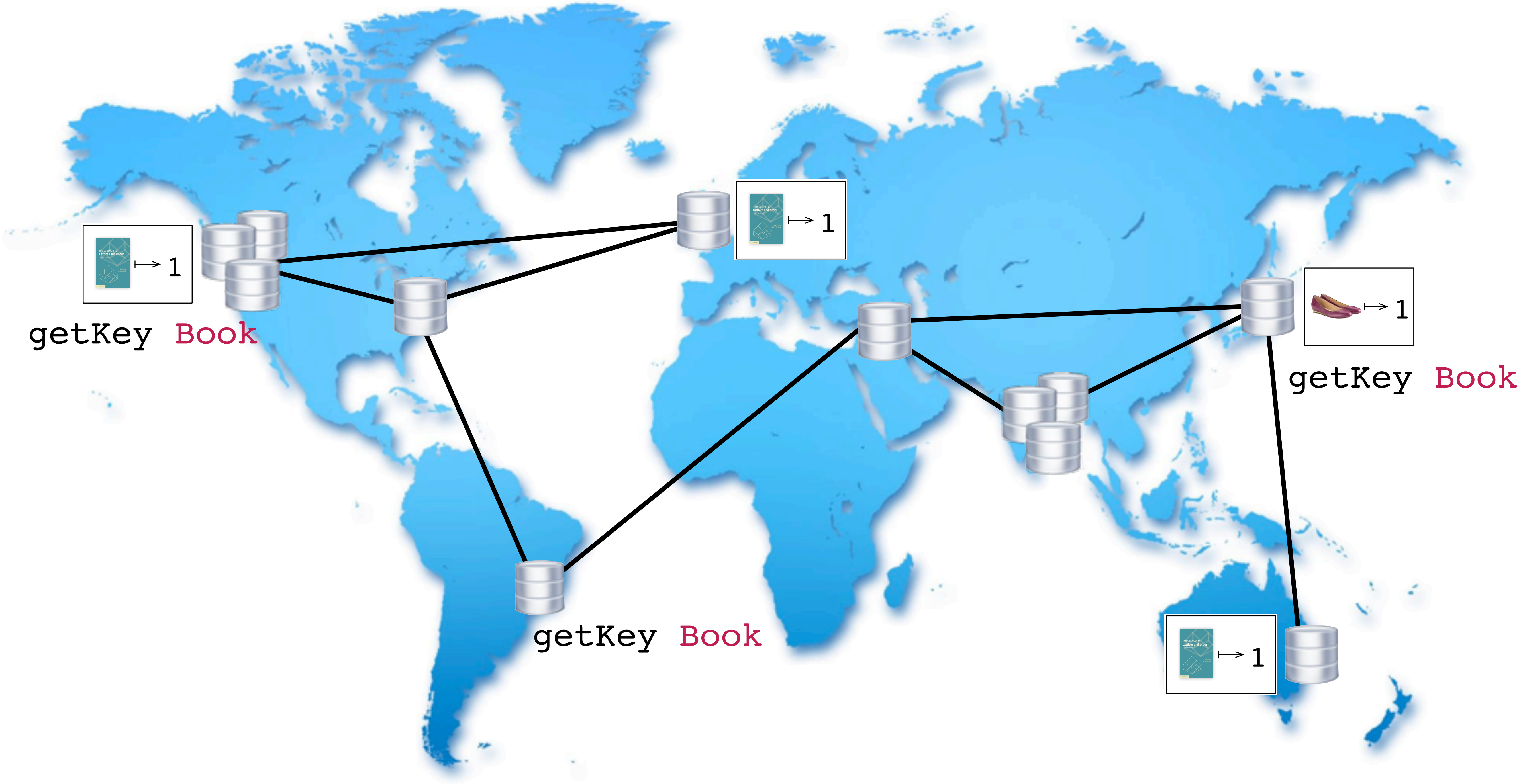
Parallel systems

Distributed systems

Distributed systems



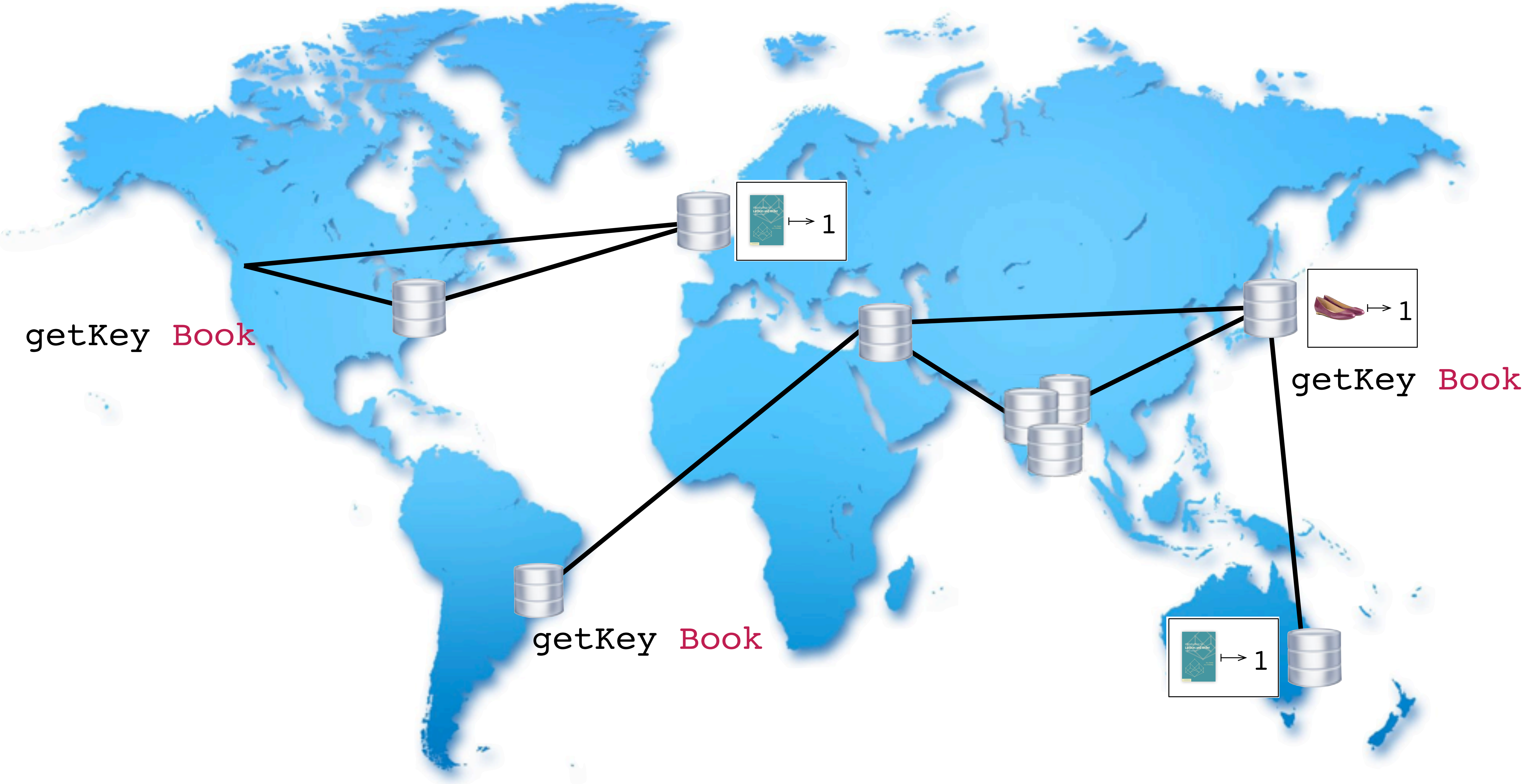

↦ 1
↦ 1
↦ 1
↦ 1



getKey Book

getKey Book

getKey Book

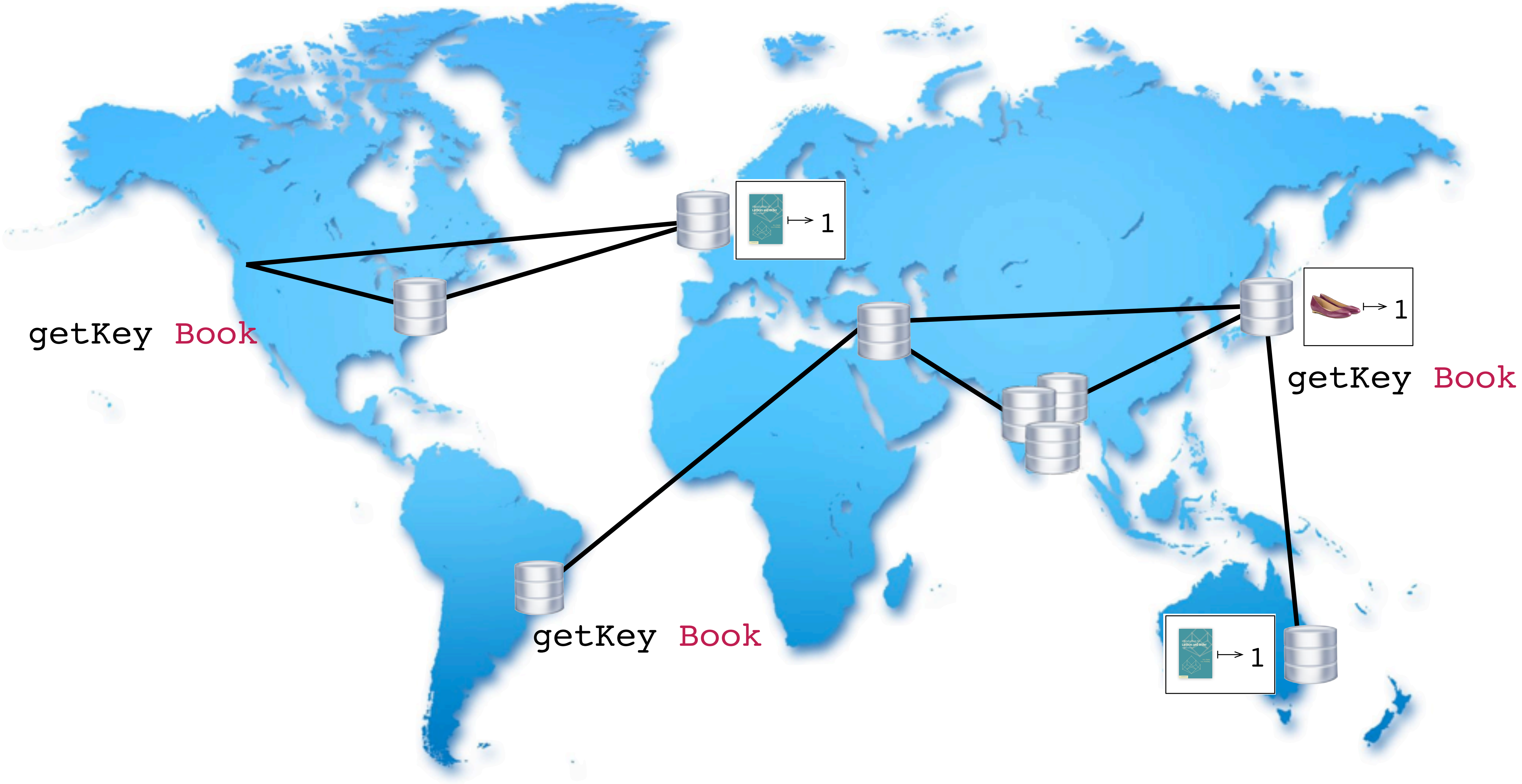


getKey Book

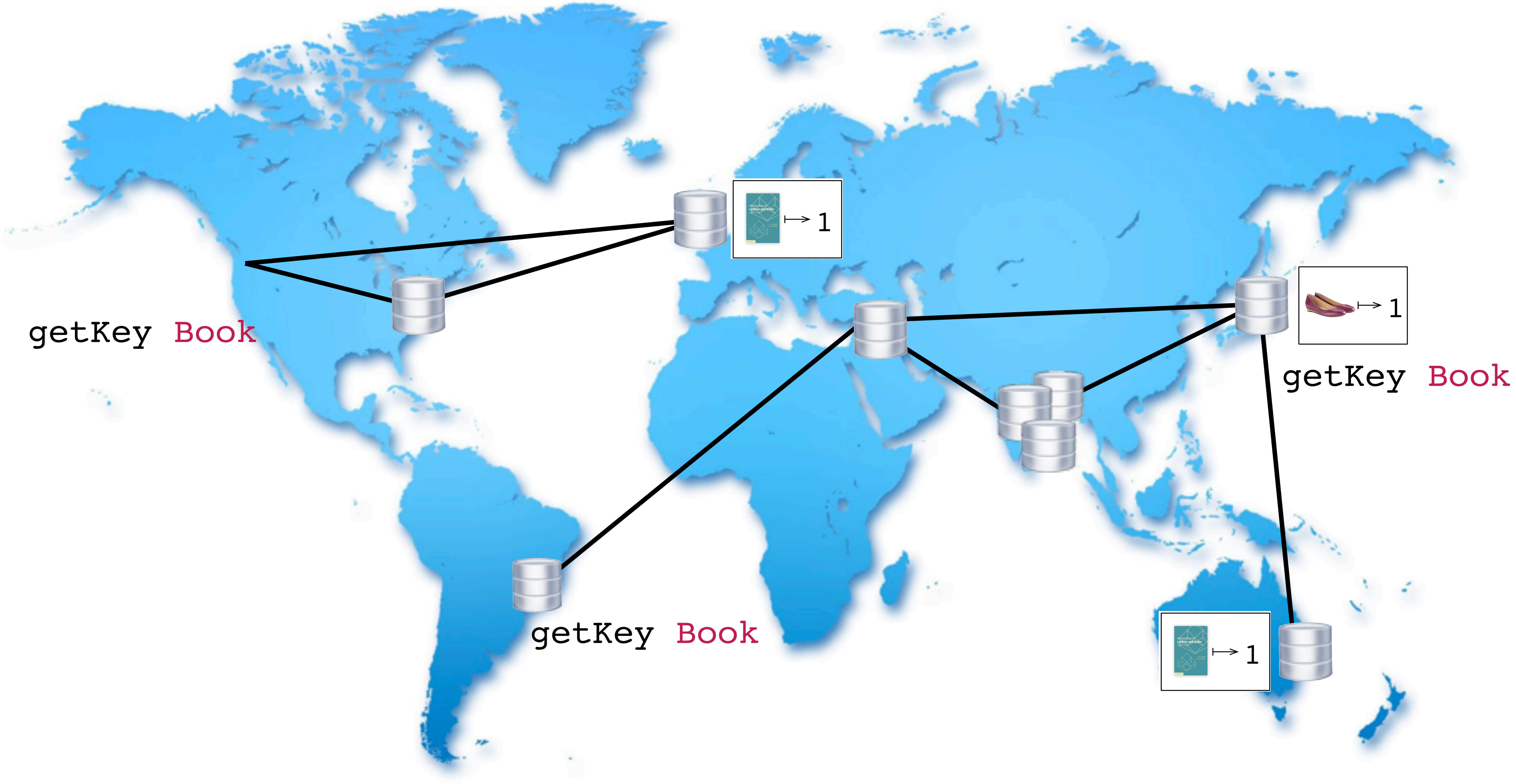
getKey Book

getKey Book

Eventual consistency.



Eventual consistency. *How?*



`getKey Book`

`getKey Book`

`getKey Book`

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

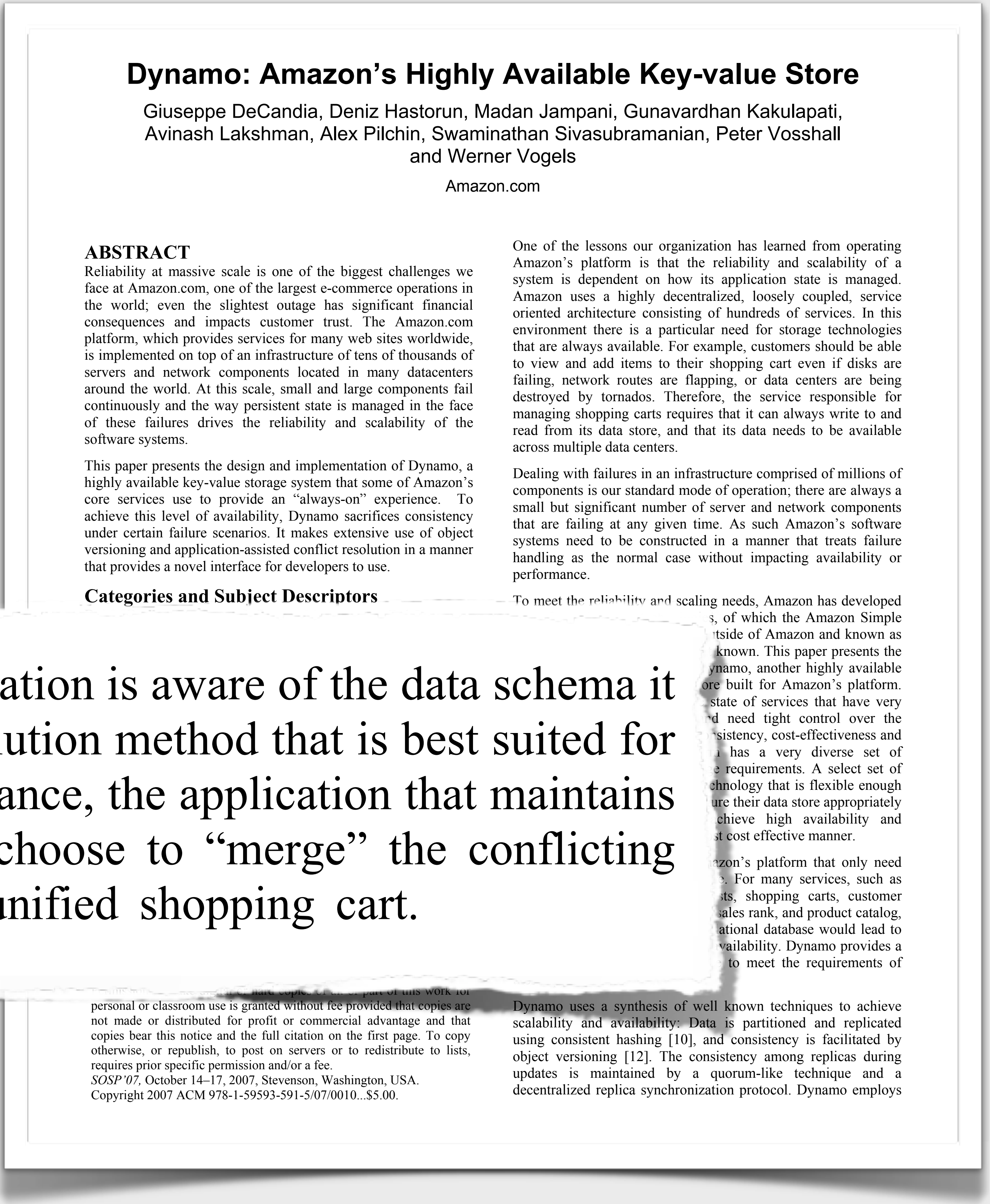
To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

[DeCandia *et al*., SOSP '07]



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed
s, of which the Amazon Simple
tside of Amazon and known as
known. This paper presents the
ynamo, another highly available
ore built for Amazon's platform.
state of services that have very
d need tight control over the
sistency, cost-effectiveness and
has a very diverse set of
requirements. A select set of
chnology that is flexible enough
ure their data store appropriately
chieve high availability and
st cost effective manner.

zon's platform that only need
. For many services, such as
sts, shopping carts, customer
sales rank, and product catalog,
ational database would lead to
vailability. Dynamo provides a
to meet the requirements of

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

> since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

[DeCandia *et al*., SOSP '07]

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

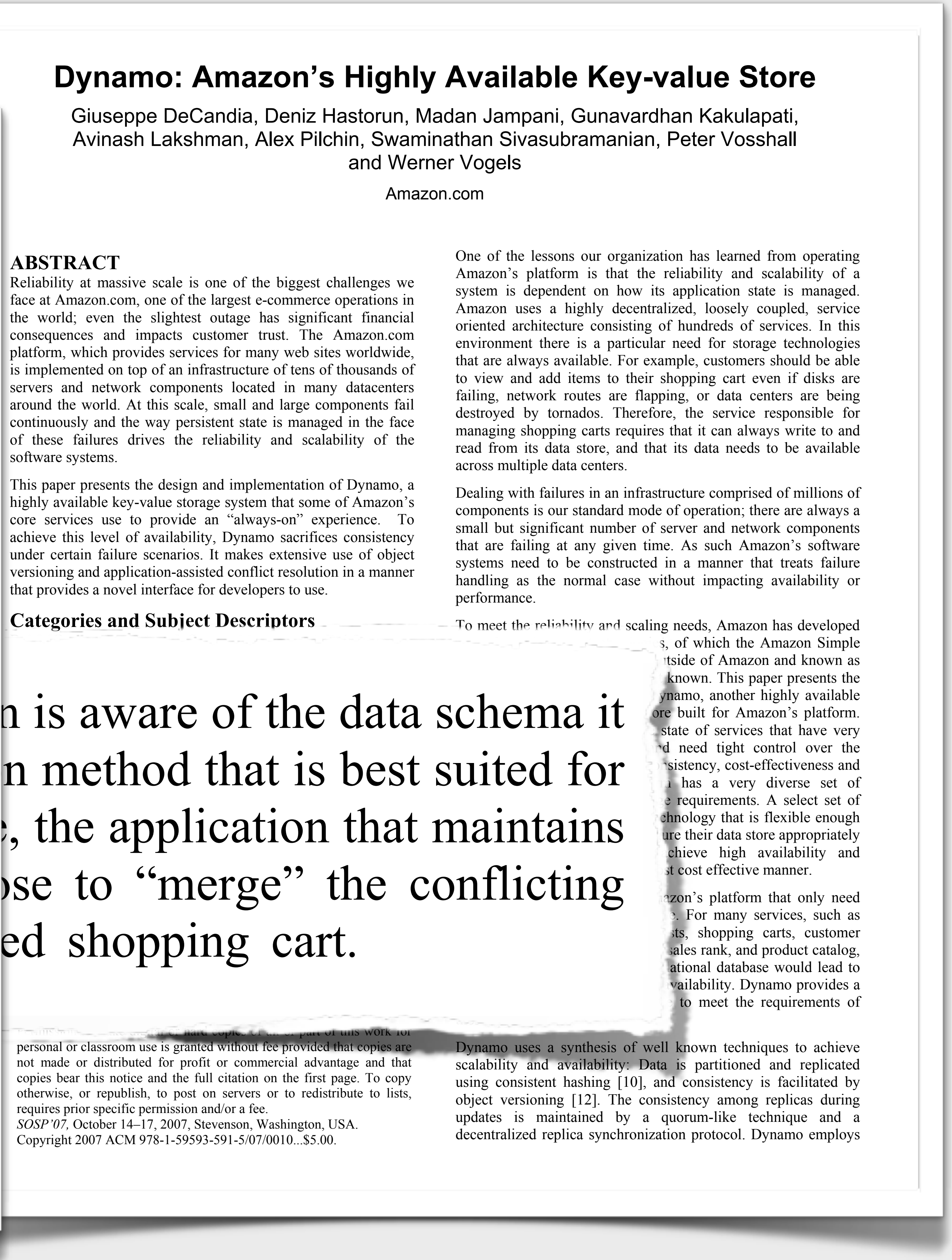
Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard "strong consistency" approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

[Shapiro *et al*., SSS '11]



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

n is aware of the data schema it
n method that is best suited for
, the application that maintains
ose to "merge" the conflicting
ed shopping cart.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed
s, of which the Amazon Simple
tside of Amazon and known as
known. This paper presents the
ynamo, another highly available
ore built for Amazon's platform.
state of services that have very
d need tight control over the
sistency, cost-effectiveness and
has a very diverse set of
requirements. A select set of
chnology that is flexible enough
ure their data store appropriately
chieve high availability and
st cost effective manner.

zon's platform that only need
. For many services, such as
sts, shopping carts, customer
sales rank, and product catalog,
ational database would lead to
vailability. Dynamo provides a
to meet the requirements of

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

[DeCandia *et al*., SOSP '07]

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard "strong consistency" approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

[Shapiro *et al.*, SSS '11]

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels
Amazon.com

ABSTRACT

personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

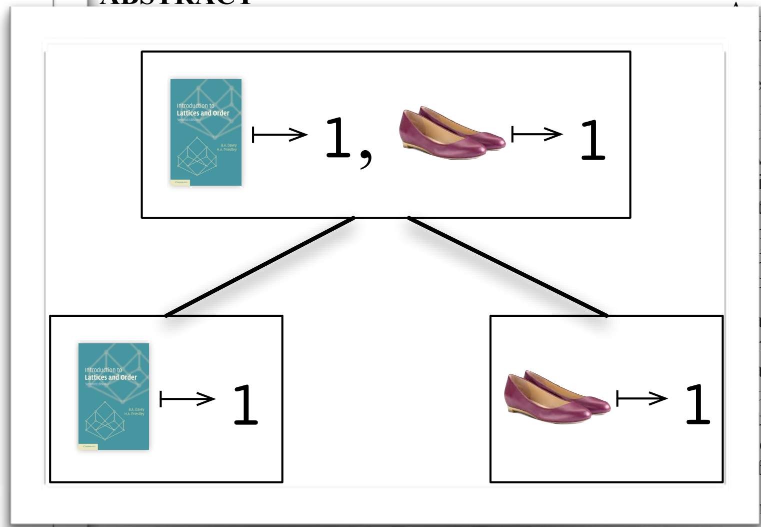
One of the lessons our organization has learned from operating
azon's platform is that the reliability and scalability of a
em is dependent on how its application state is managed.
azon uses a highly decentralized, loosely coupled, service
nted architecture consisting of hundreds of services. In this
ironment there is a particular need for storage technologies
are always available. For example, customers should be able
view and add items to their shopping cart even if disks are
ing, network routes are flapping, or data centers are being
troyed by tornados. Therefore, the service responsible for
naging shopping carts requires that it can always write to and
d from its data store, and that its data needs to be available
oss multiple data centers.

ling with failures in an infrastructure comprised of millions of
ponents is our standard mode of operation; there are always a
ll but significant number of server and network components
are failing at any given time. As such Amazon's software
ems need to be constructed in a manner that treats failure
dling as the normal case without impacting availability or
formance.

meet the reliability and scaling needs, Amazon has developed
s, of which the Amazon Simple
tside of Amazon and known as
known. This paper presents the
ynamo, another highly available
ore built for Amazon's platform.
state of services that have very
d need tight control over the
sistency, cost-effectiveness and
has a very diverse set of
requirements. A select set of
chnology that is flexible enough
ure their data store appropriately
chieve high availability and
t cost effective manner.

zon's platform that only need
. For many services, such as
sts, shopping carts, customer
sales rank, and product catalog,
ational database would lead to
vailability. Dynamo provides a
to meet the requirements of

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs



n is aware of the data schema it
n method that is best suited for
, the application that maintains
se to "merge" the conflicting
ed shopping cart.

[DeCandia *et al.*, SOSP '07]

Two "styles" of Conflict-Free Replicated Data Types:

"Convergent"

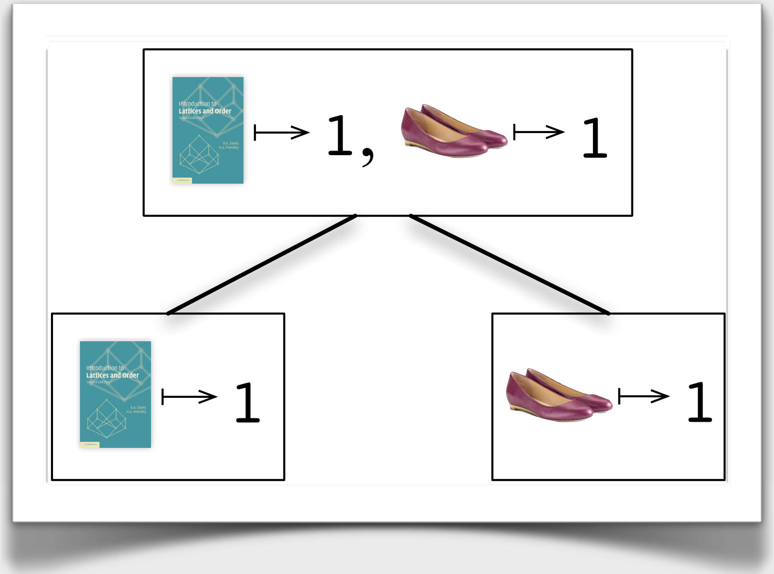
CvRDTs

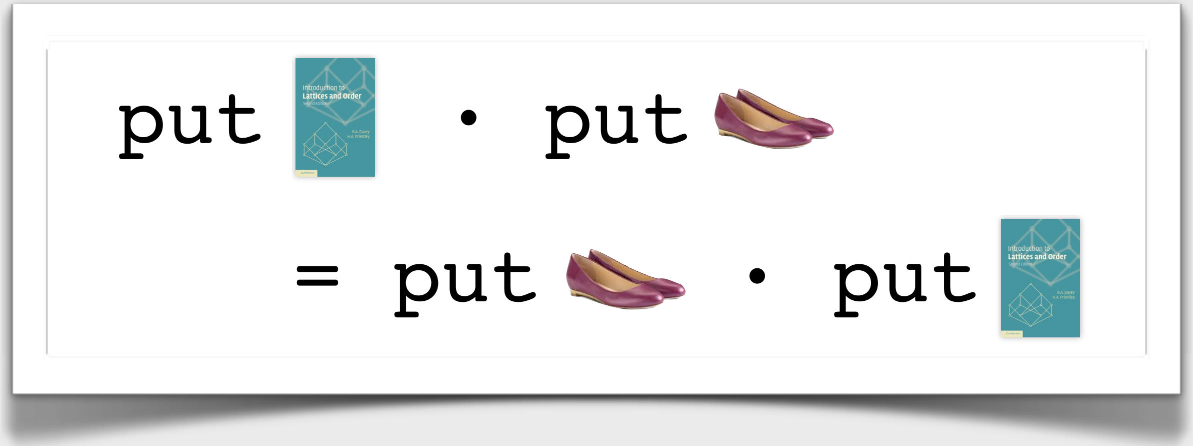
"state-based"

"Commutative"

CmRDTs

"op-based"



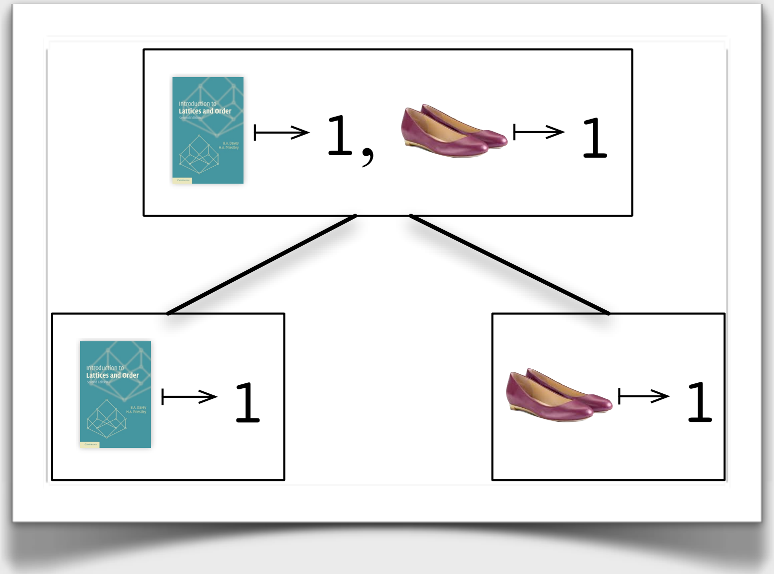


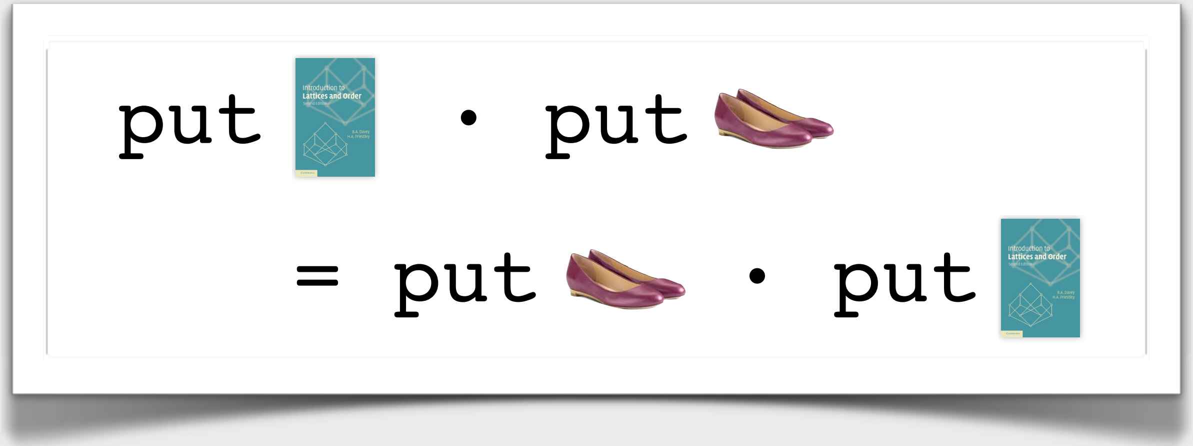
Two "styles" of Conflict-Free Replicated Data Types:

"Convergent" ⟺ "Commutative"

CvRDTs ⟺ CmRDTs

"state-based" [Shapiro *et al.*, SSS '11] "op-based"





Two "styles" of Conflict-Free Replicated Data Types:

"Convergent"

CvRDTs

"state-based"

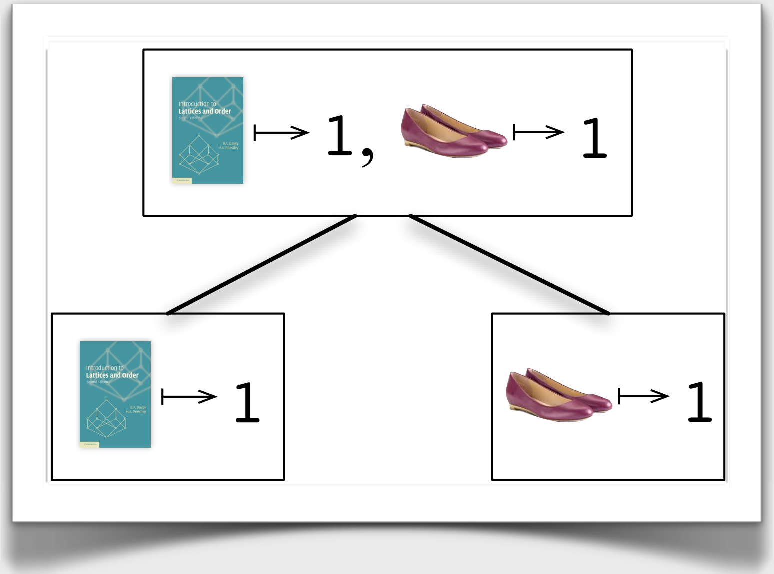
⟺

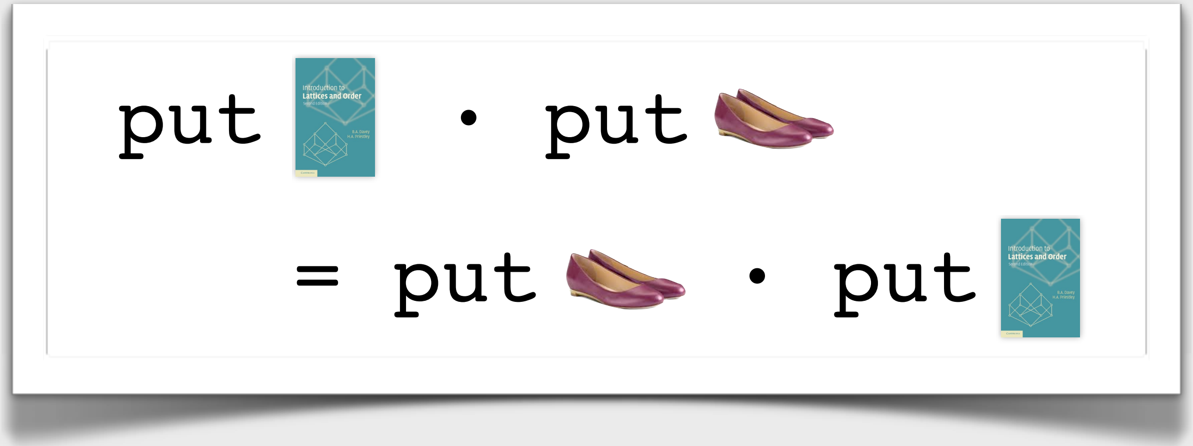
[Shapiro *et al.*, SSS '11]

"Commutative"

CmRDTs

"op-based"





put · put

= put · put

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

so
we're
proposing
[WoDet '14]

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

so
we're
proposing
[WoDet '14]

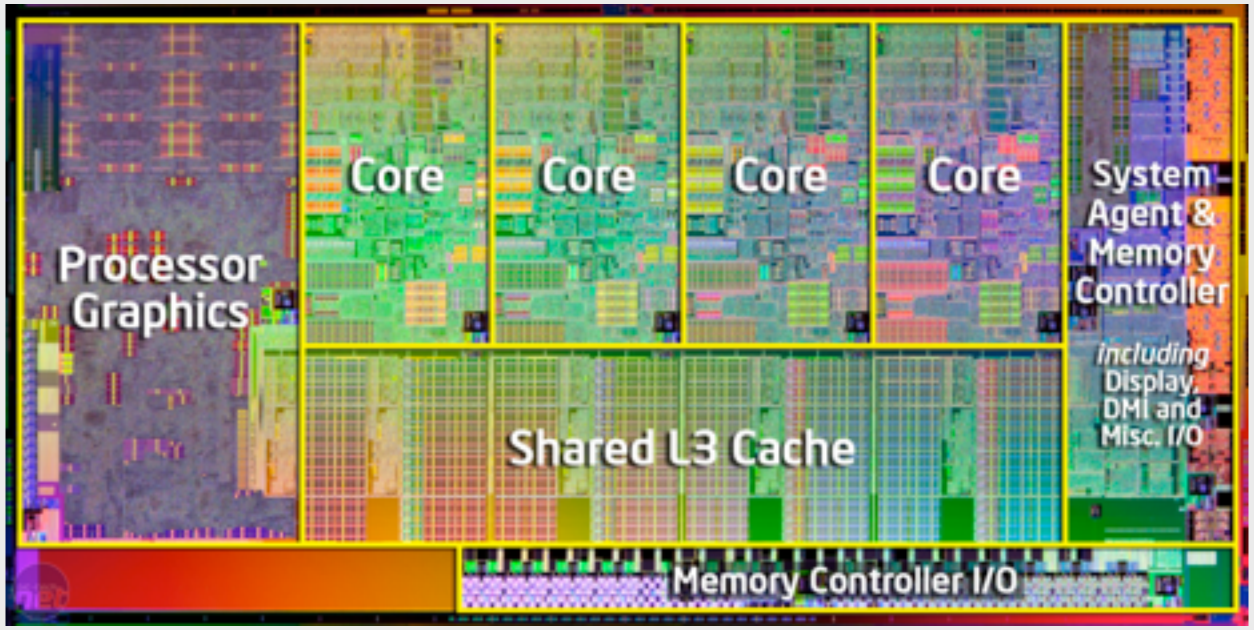
- Adding threshold reads to CvRDTs
  One framework for reasoning about both eventual and strong consistency

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (nondeterministic)
Least-upper-bound writes (every write computes a join)	General inflationary writes (only *merges* must be joins)
Shared memory	Replicated!

so we're proposing [WoDet '14]

- Adding threshold reads to CvRDTs
  One framework for reasoning about both eventual and strong consistency
- Adding general inflationary writes to LVars
  Non-idempotent, incrementable counters

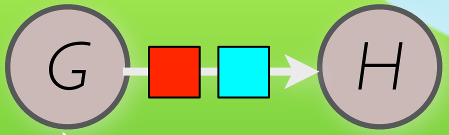


Parallel systems

Canada
United States
North Atlantic Ocean
South Atlantic Ocean
South Pacific Ocean
Indian Ocean
Russia
China
Australia
☑ Active Cables
□ Future Cables

Distributed systems

LVars and LVish across the landscape:



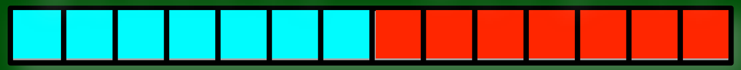
✓ Kahn process networks

✓✓ single-assignment

✓ quasi-det.

✓ imperative disjoint

```
g(left)   h(rght)
```



✓ λ-calculus

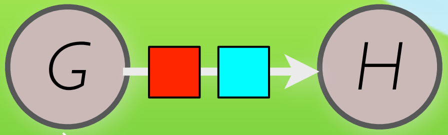
```
f  (g  x)
   (h  y)
```

CvRDTs ✓

distributed LVish ✓

LVars and LVish across the landscape:

quasi-det. ✓



Kahn process networks ✓

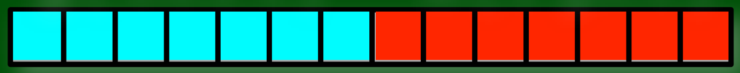
single-assignment ✓✓

imperative disjoint ✓

λ-calculus ✓

```
f (g x)
  (h y)
```

```
g(left)  h(rght)
```



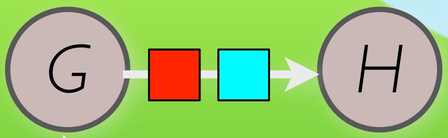
LVars and LVish across the landscape:

✓ CvRDTs

✓ distributed LVish

✓ quasi-det.

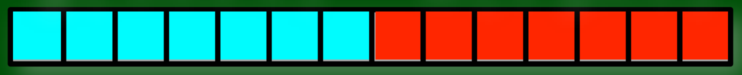
✓✓ single-assignment



✓ Kahn process networks

✓ imperative disjoint

```
g(left)   h(rght)
```



✓ λ-calculus

```
f  (g  x)
   (h  y)
```

Thank you!

Email: lkuper@cs.indiana.edu
LVars project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples
Papers: cs.indiana.edu/~lkuper
Research blog: composition.al