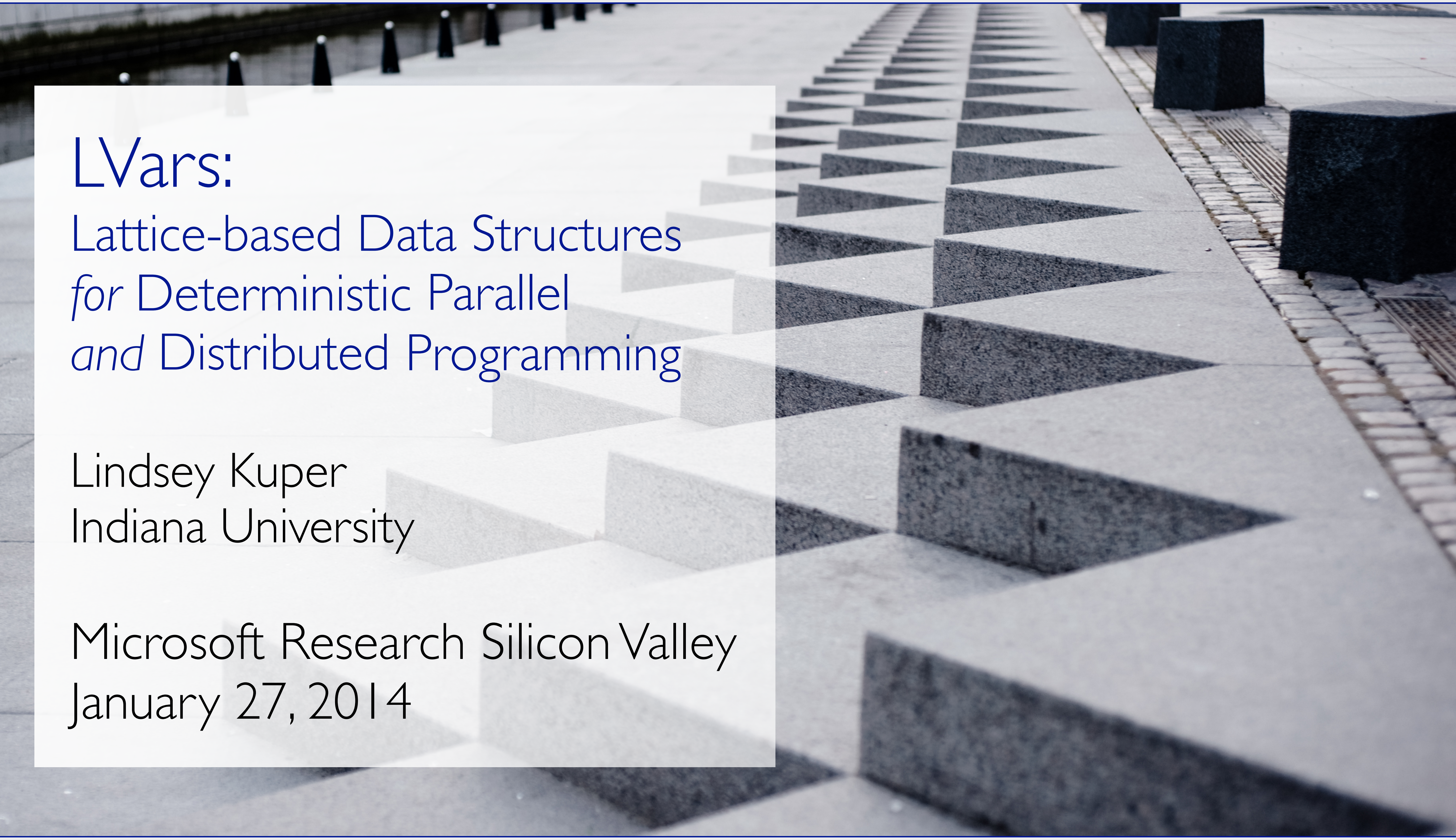
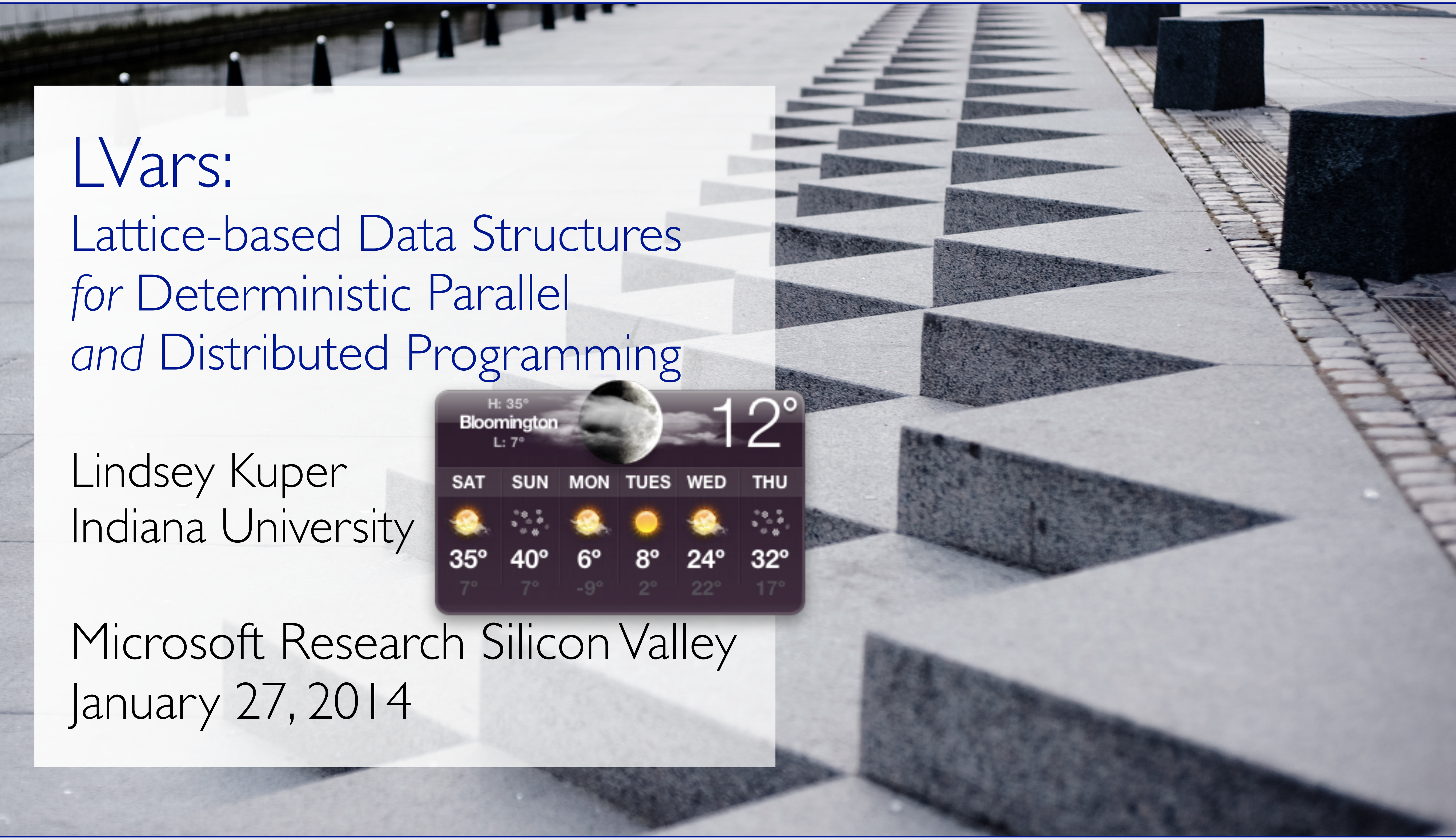


LVars: Lattice-based Data Structures *for* Deterministic Parallel *and* Distributed Programming

Lindsey Kuper
Indiana University

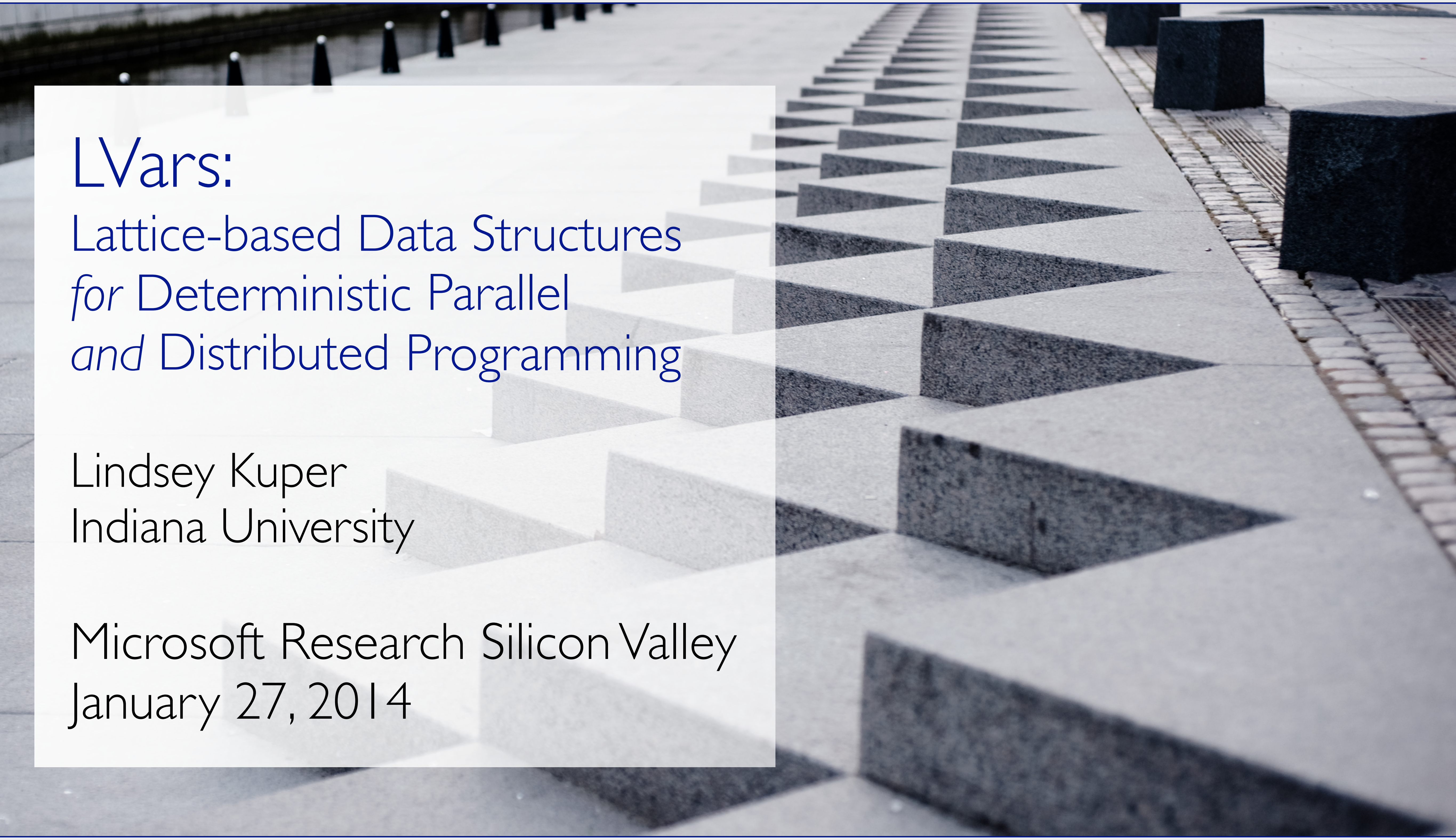
Microsoft Research Silicon Valley
January 27, 2014



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

Lindsey Kuper
Indiana University

Microsoft Research Silicon Valley
January 27, 2014

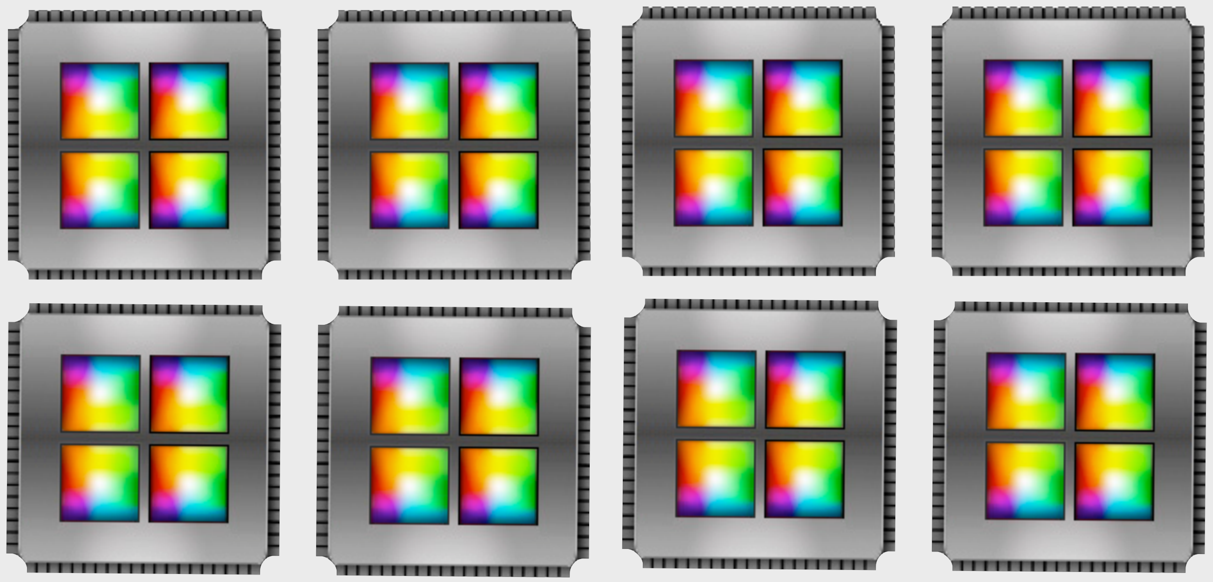


LVars:

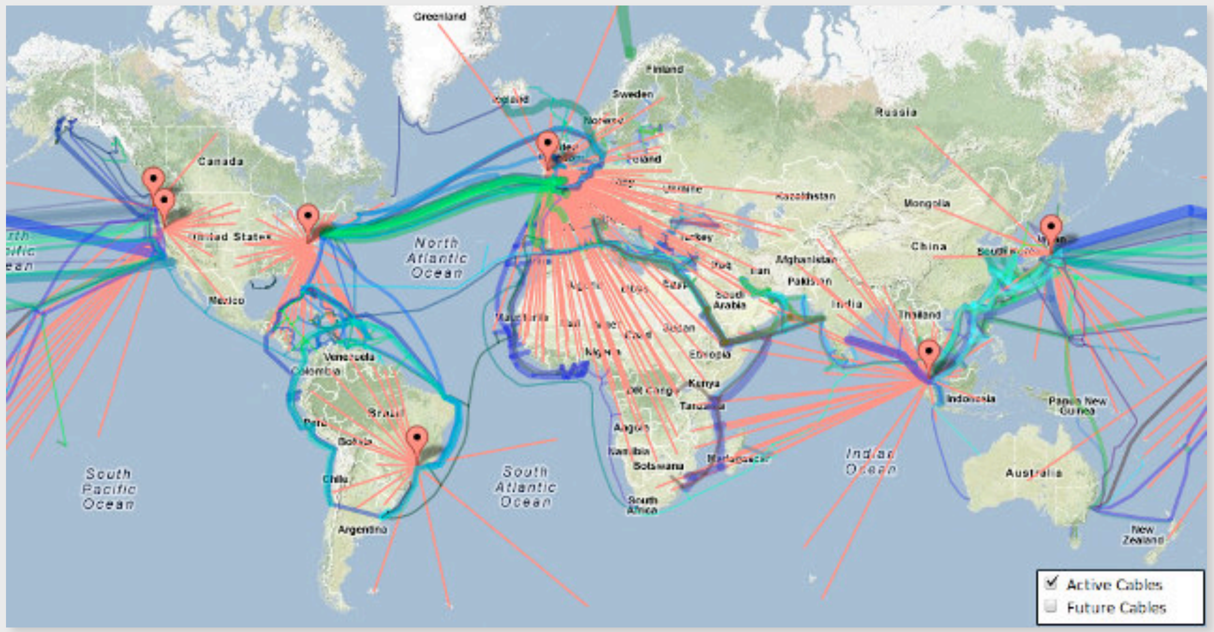
Lattice-based Data Structures *for* Deterministic Parallel *and* Distributed Programming

Lindsey Kuper
Indiana University

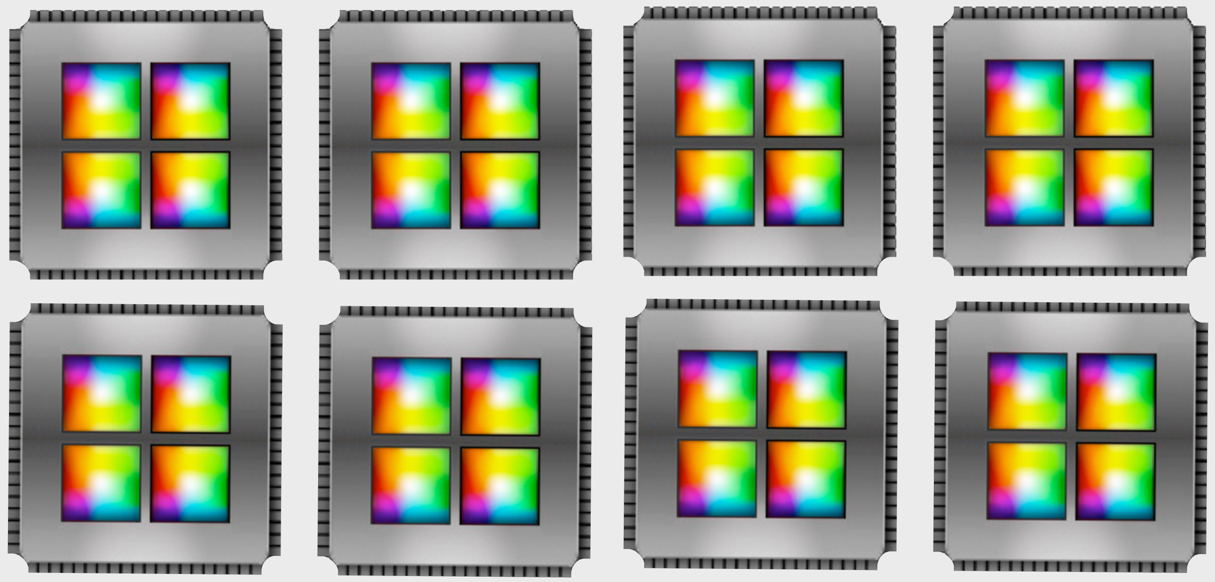
Microsoft Research Silicon Valley
January 27, 2014



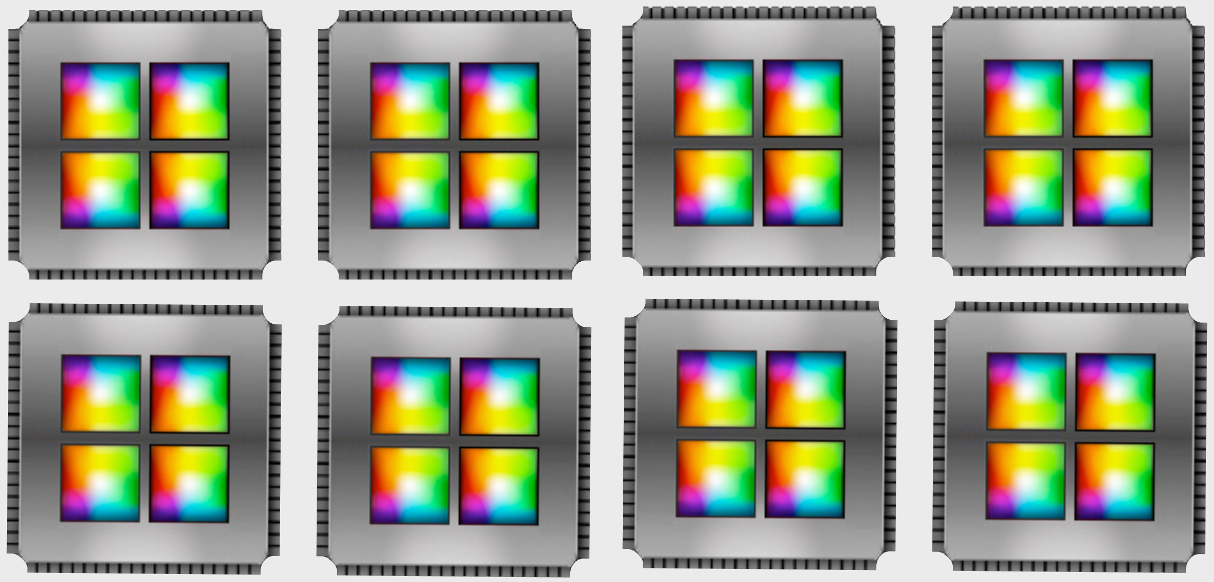
Parallel systems



Distributed systems



Deterministic Parallel Programming



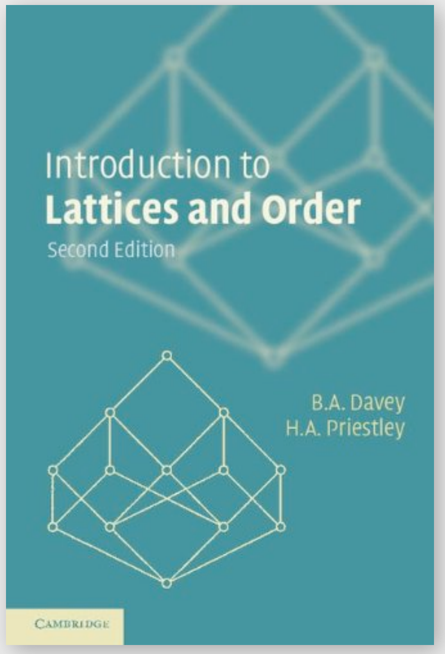
(observably)
Deterministic Parallel Programming

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

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

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

```haskell
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
```



```
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
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

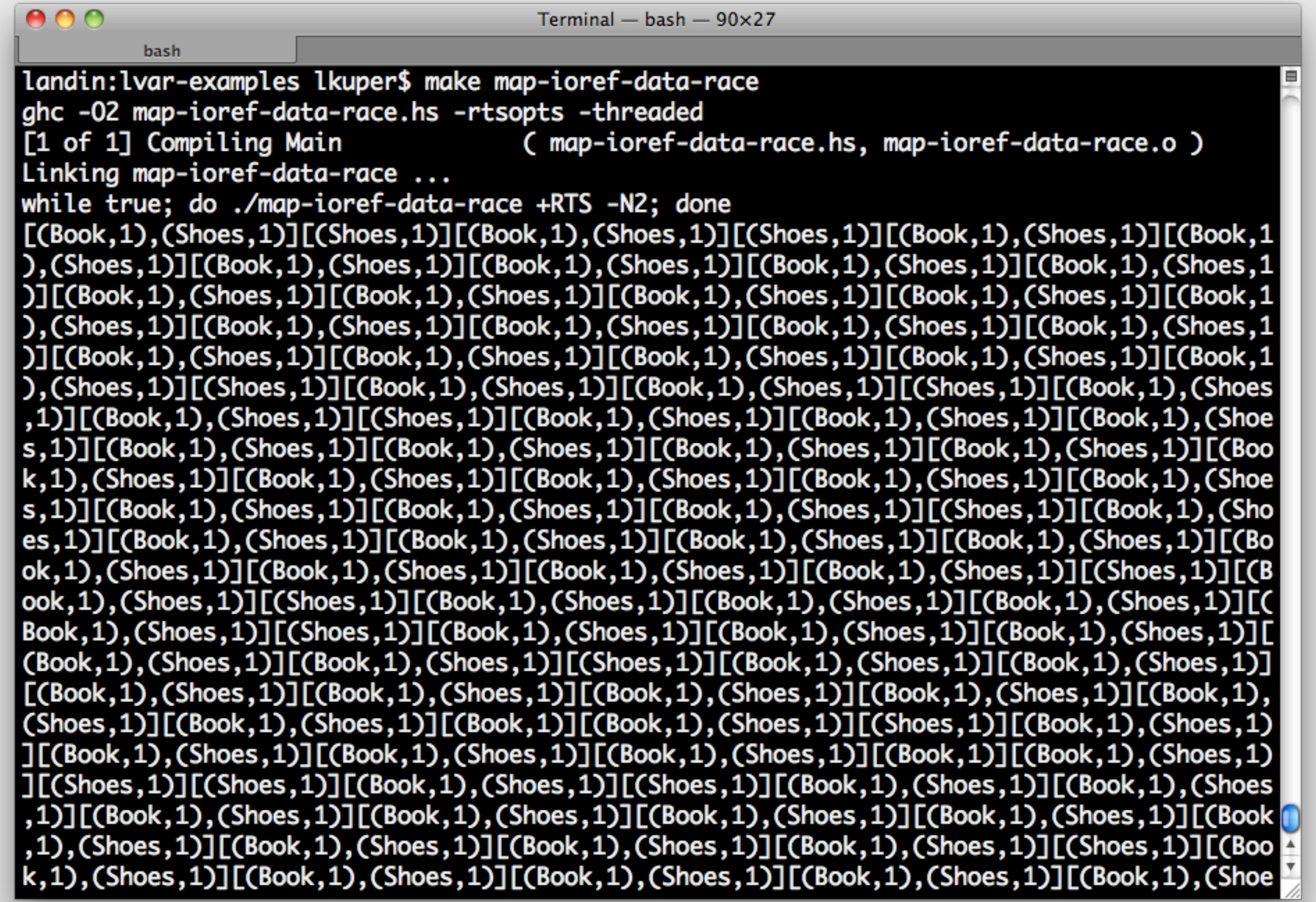
```
data Item = Book | Shoes | ...

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, ())))
```

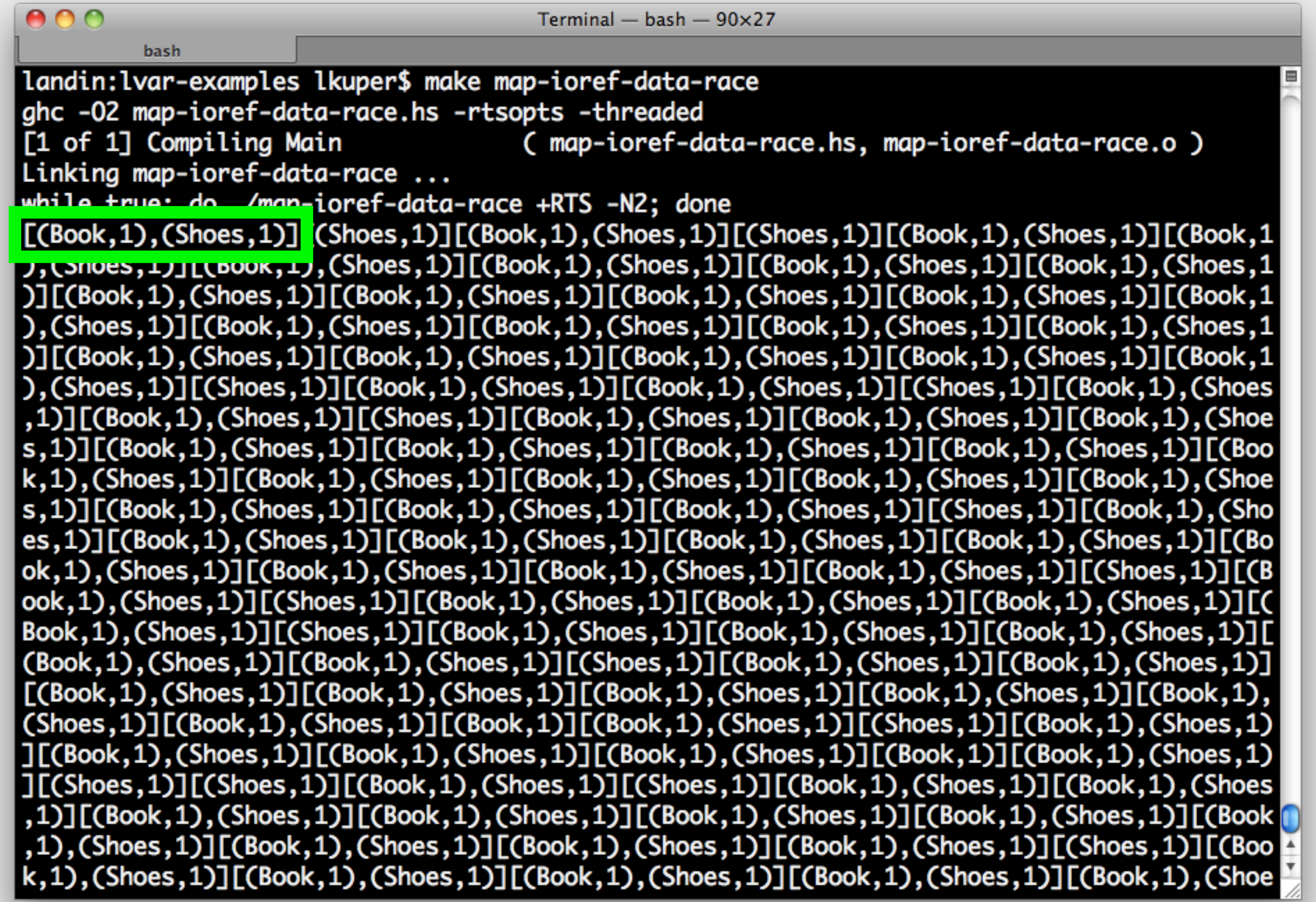
```haskell
data Item = Book | Shoes | ...

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, ())))
```

```
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
```


Terminal — bash — 90×27

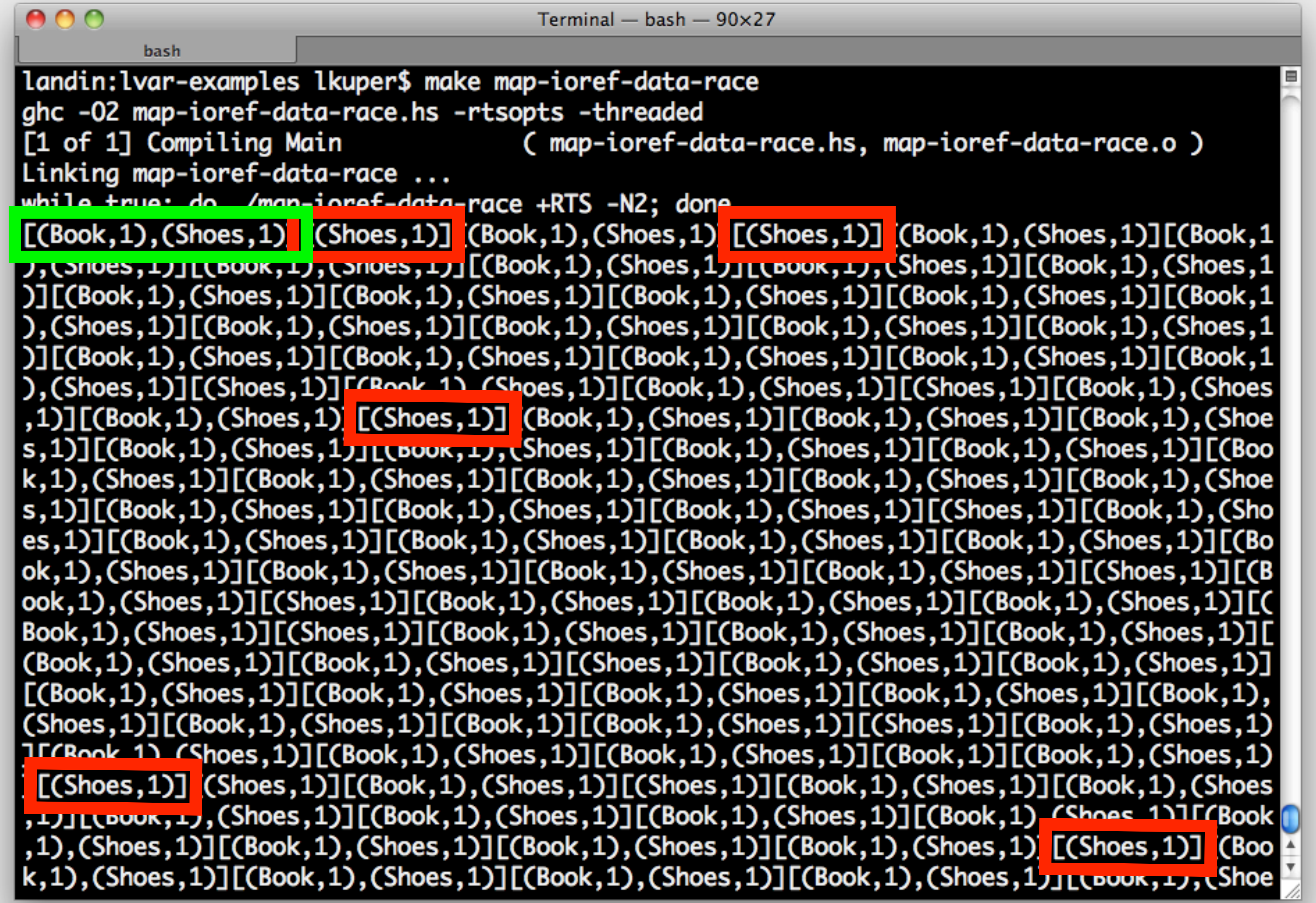
```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Bo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



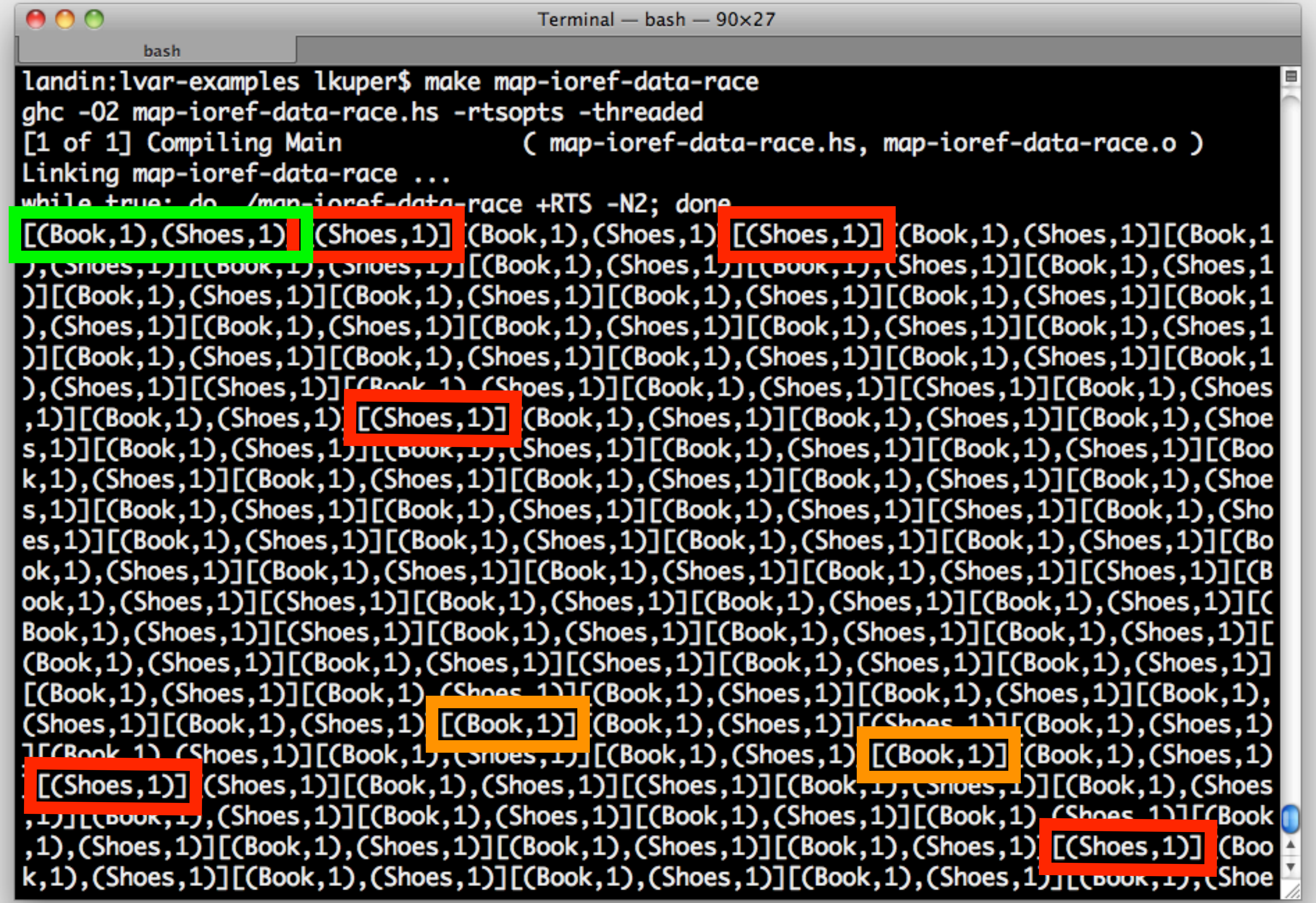
Terminal — bash — 90×27

bash

```
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
```


Terminal — bash — 90×27

```
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
```

```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
       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
```

```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 (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
          putStr (show (toList v))
```

Deterministic

```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
          putStr (show (toList 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
          putStr (show (toList 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
          putStr (show (toList 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 = do
  putStr (show (toList (fromIMap
    (runParThenFreeze p))))
```

Deterministic by construction

[Kuper and Newton, FHPC '13]

[Kuper *et al.*, POPL '14]

```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
```

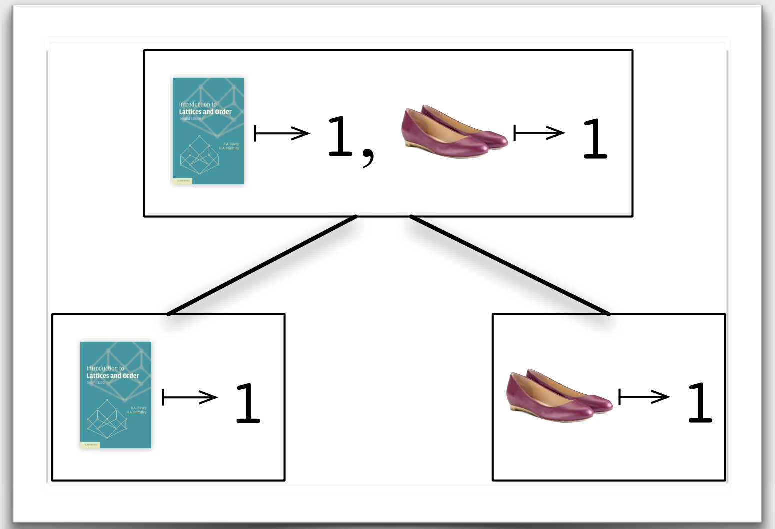
```
data Item = Book | Shoes | ...

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
```

```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

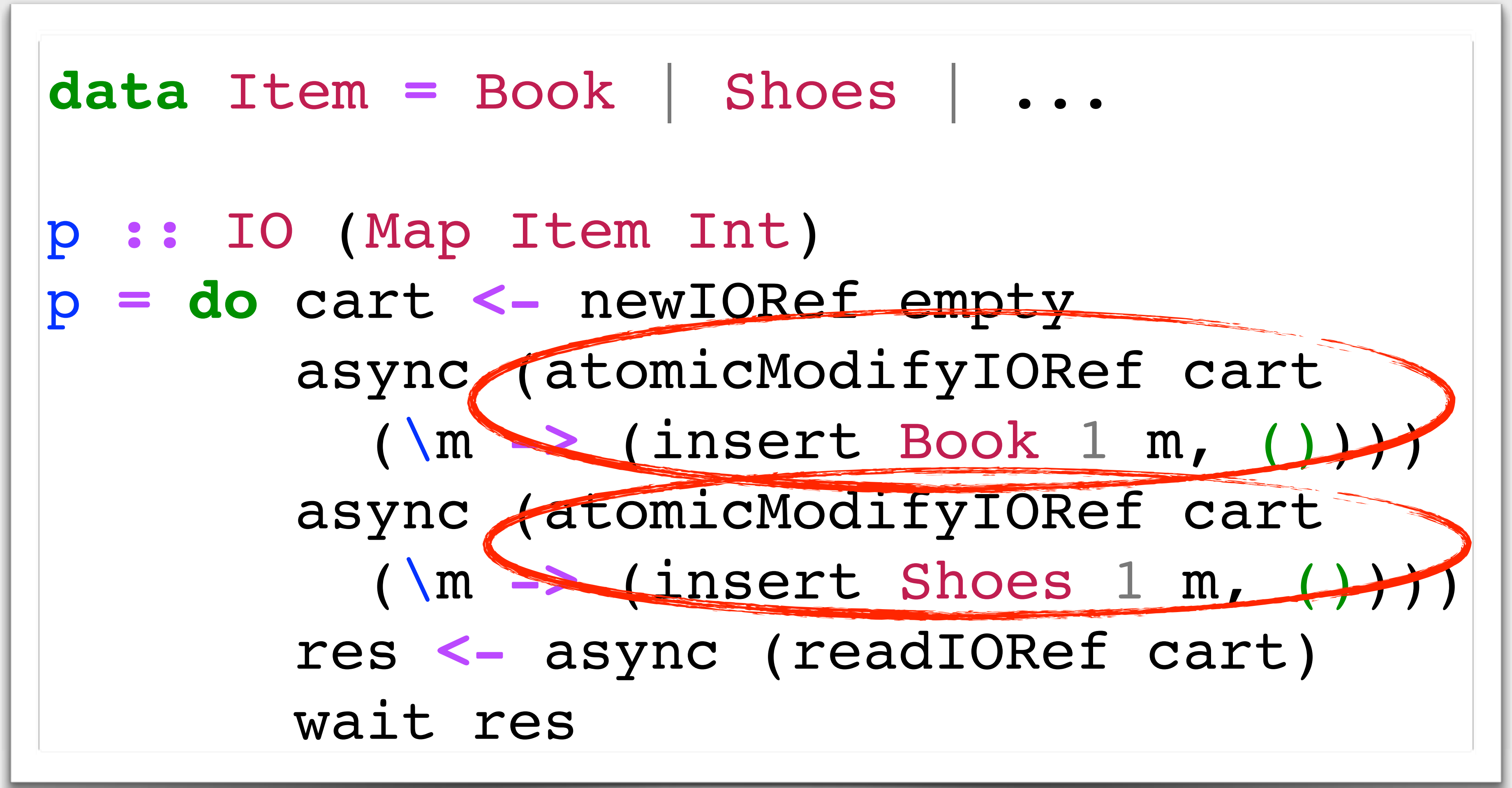
[Arvind *et al.*, 1989]

```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
```

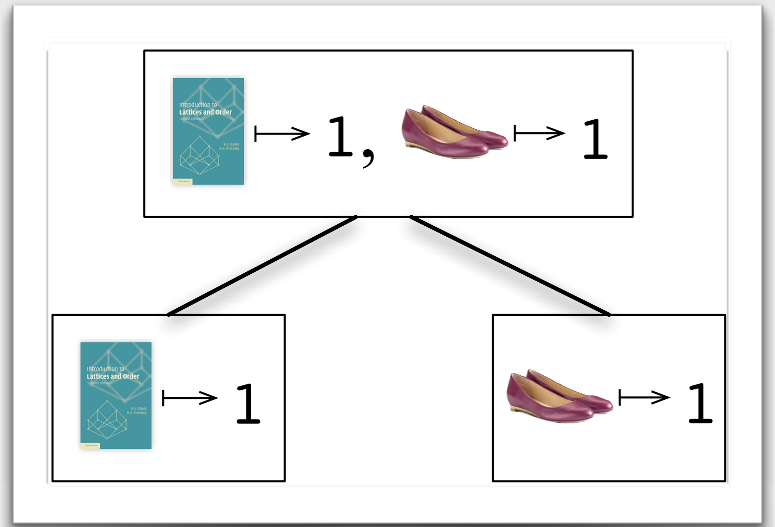


IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

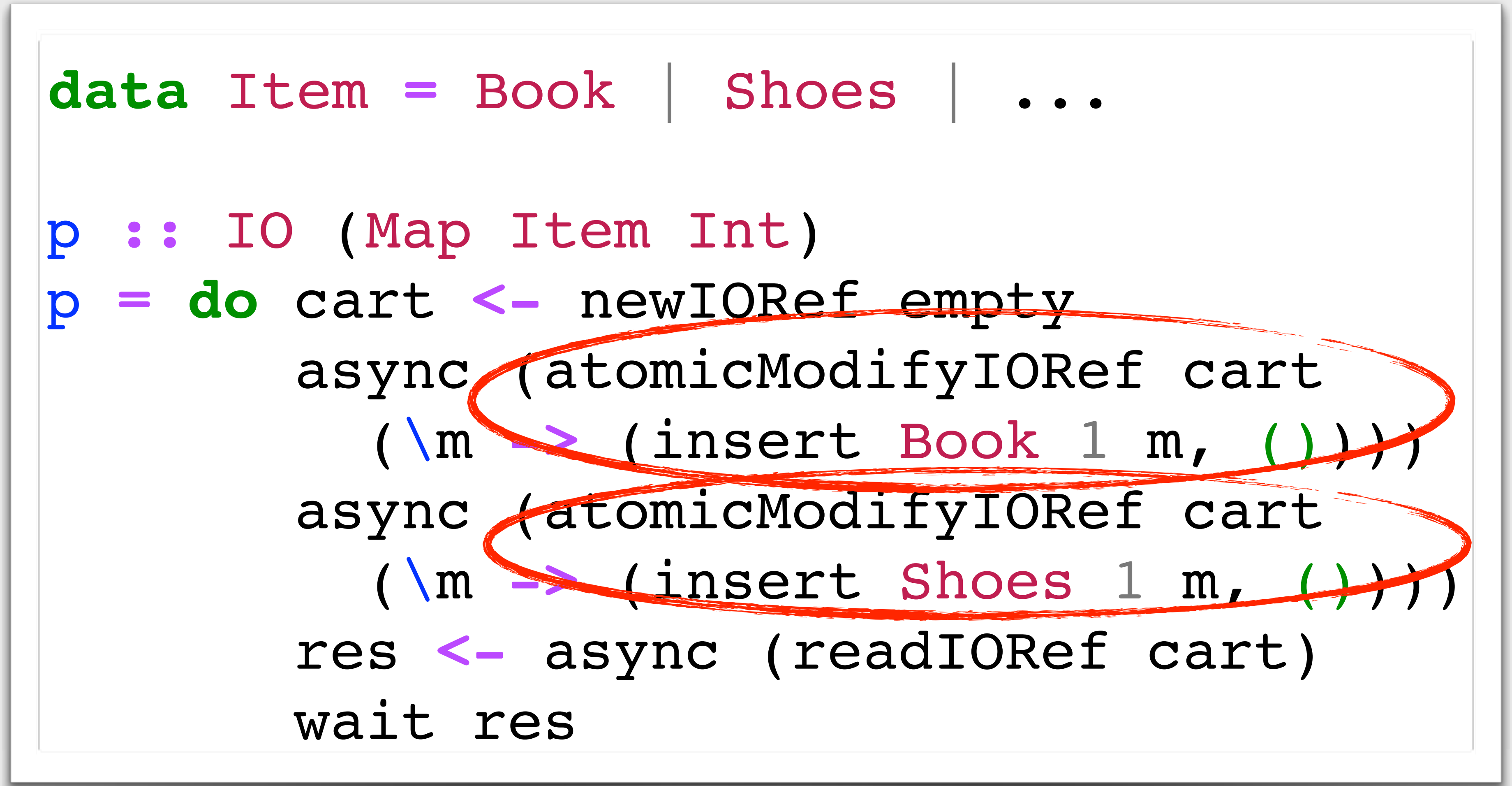


```
data Item = Book | Shoes | ...

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
```

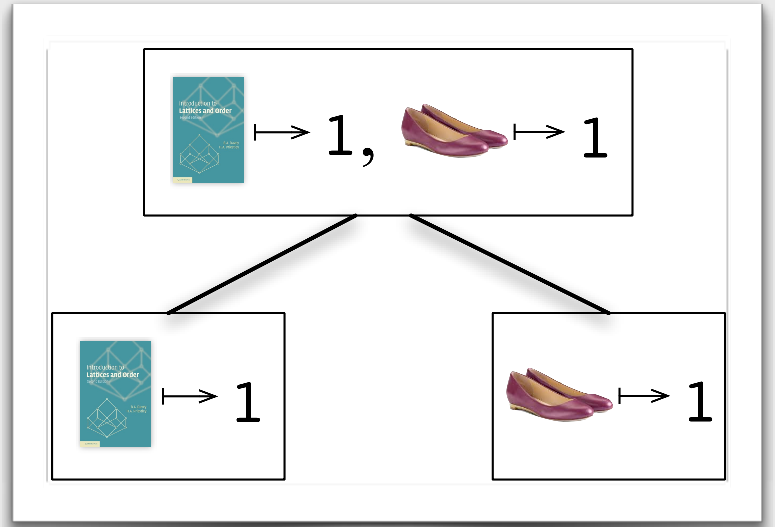


IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *least-upper-bound* writes, blocking *threshold* reads
[Kuper and Newton, 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

[Kuper and Newton, FHPC '13]

* actually a bounded join-semilattice

`num`

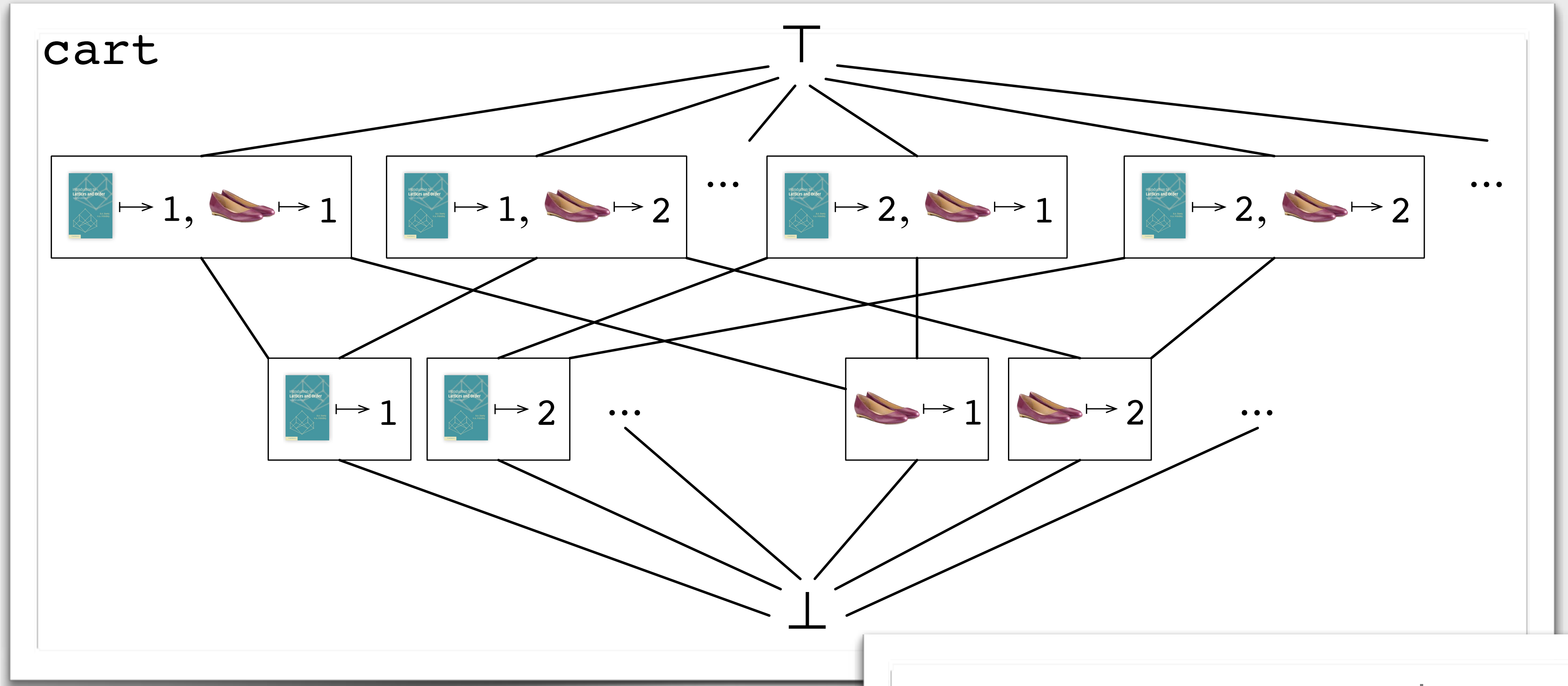
$$\begin{tikzcd}[column sep=small] & & & \top \arrow[dlll, dash] \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\ 0 & 1 & 2 & 3 & 4 & \ldots \\ & & & \bot \arrow[ulll, dash] \arrow[ull, dash] \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] \arrow[urr, dash] & & \end{tikzcd}$$

Raises an error, since $3 \sqcup 4 = \top$

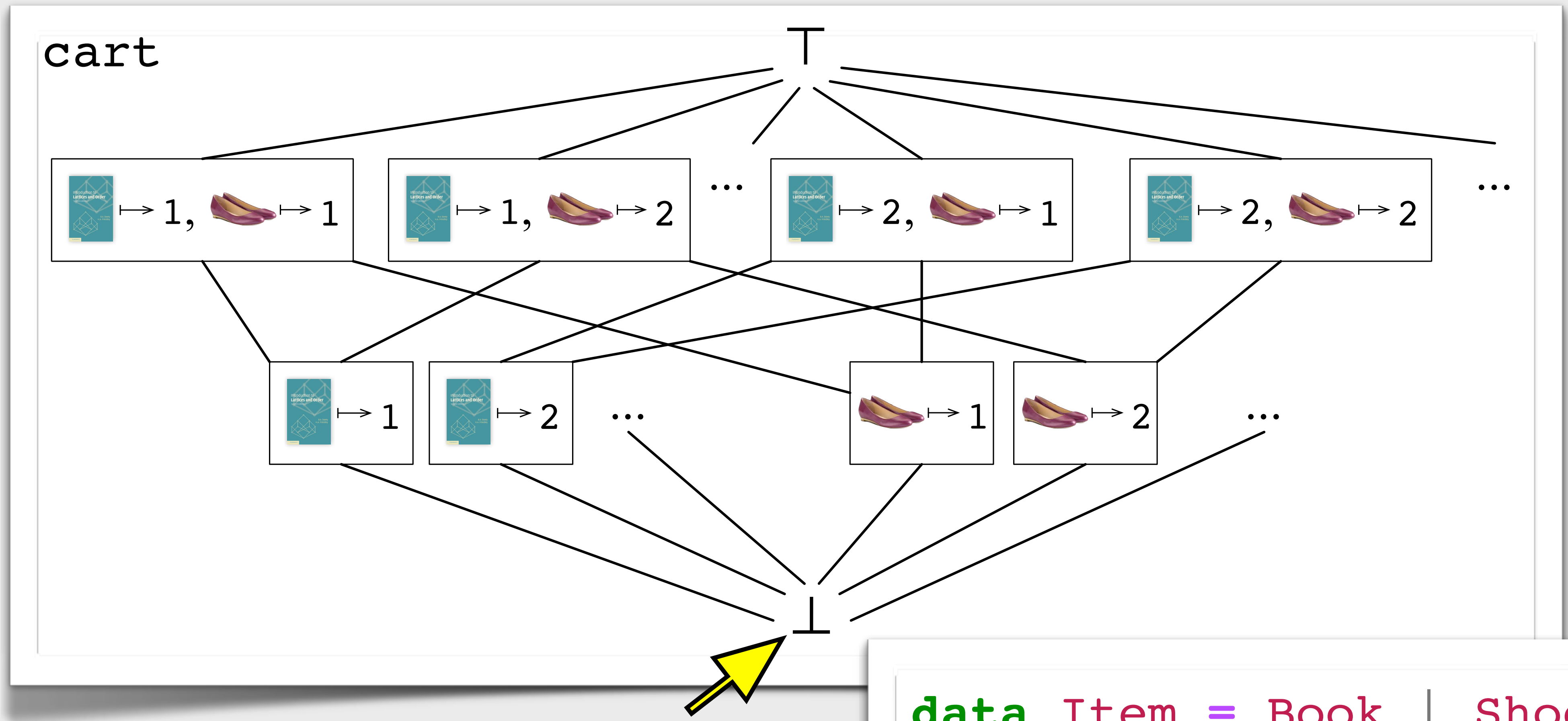
```
do
  fork (put num 3)
  fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

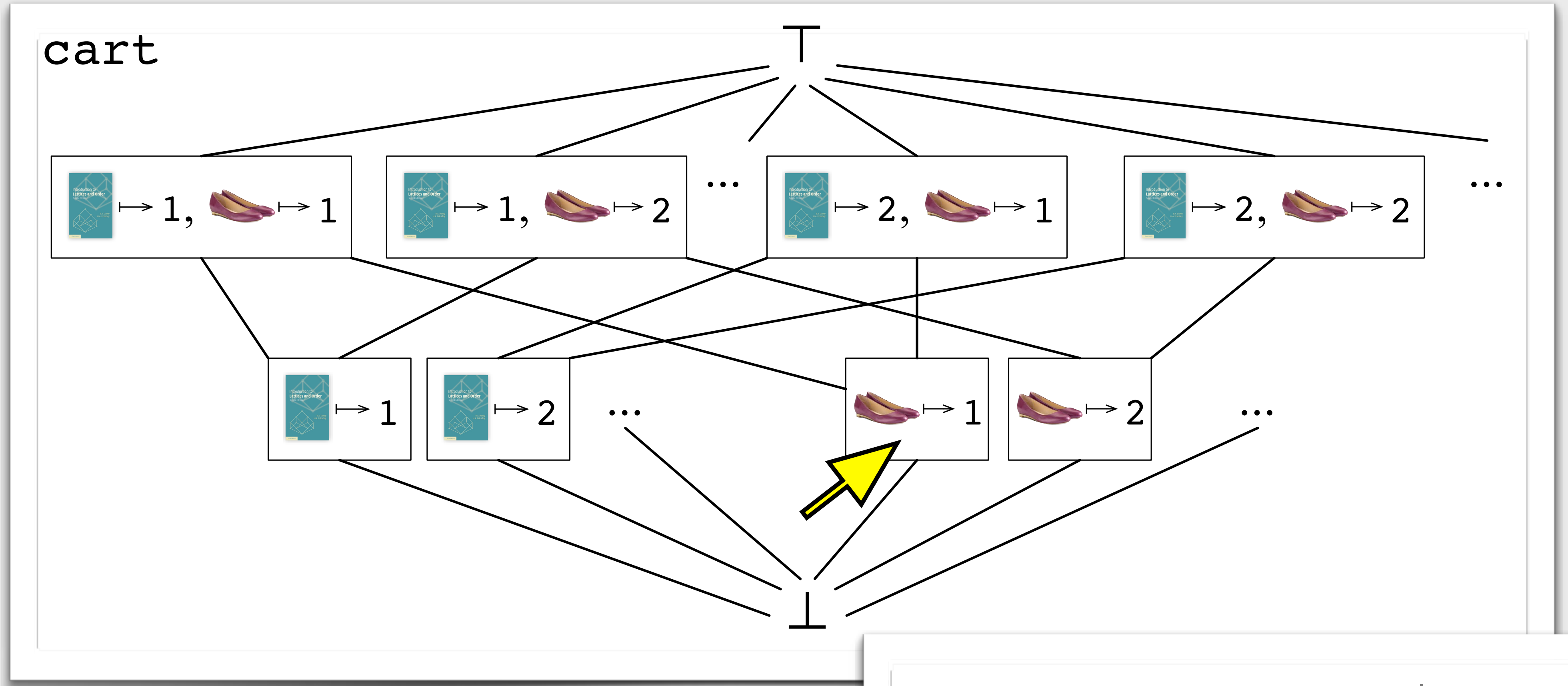
```
do
  fork (put num 4)
  fork (put num 4)
```



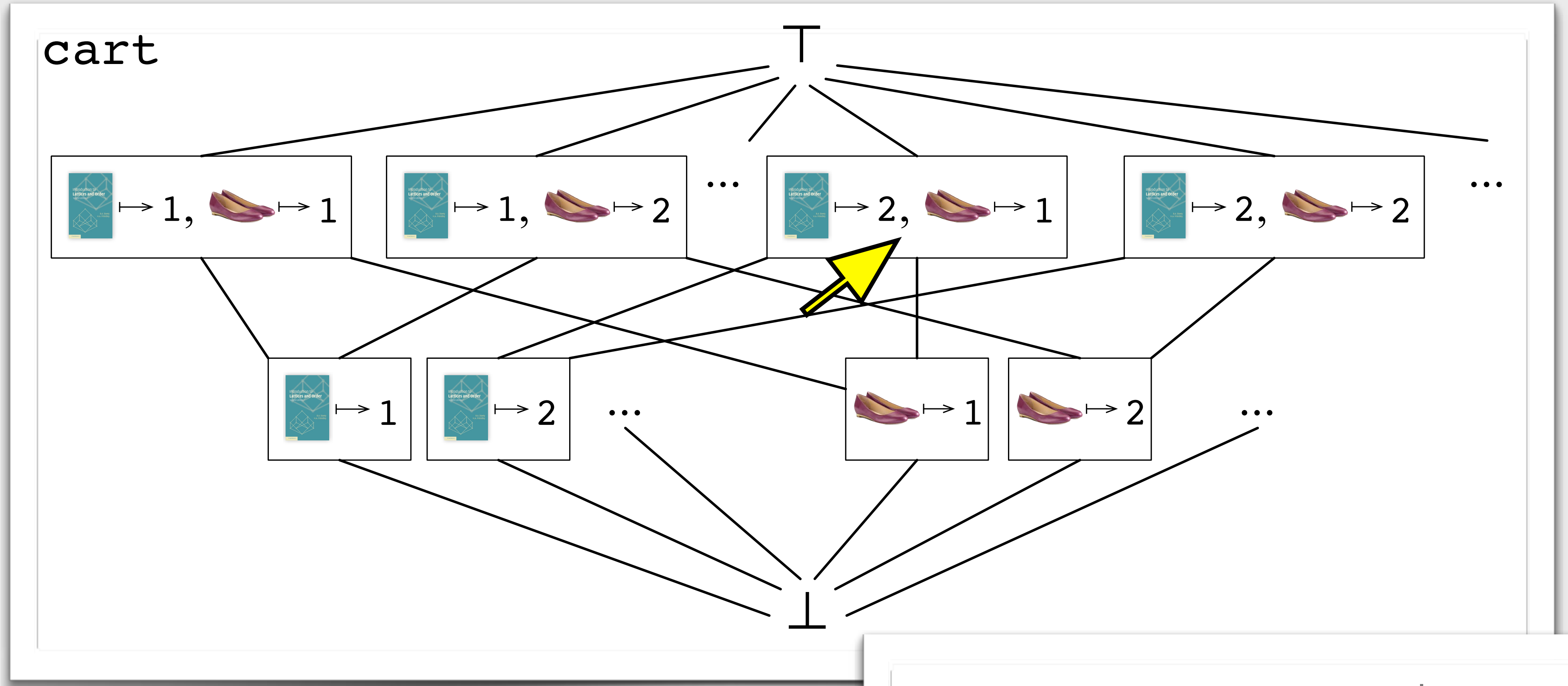
```
data Item = Book | Shoes | ...

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



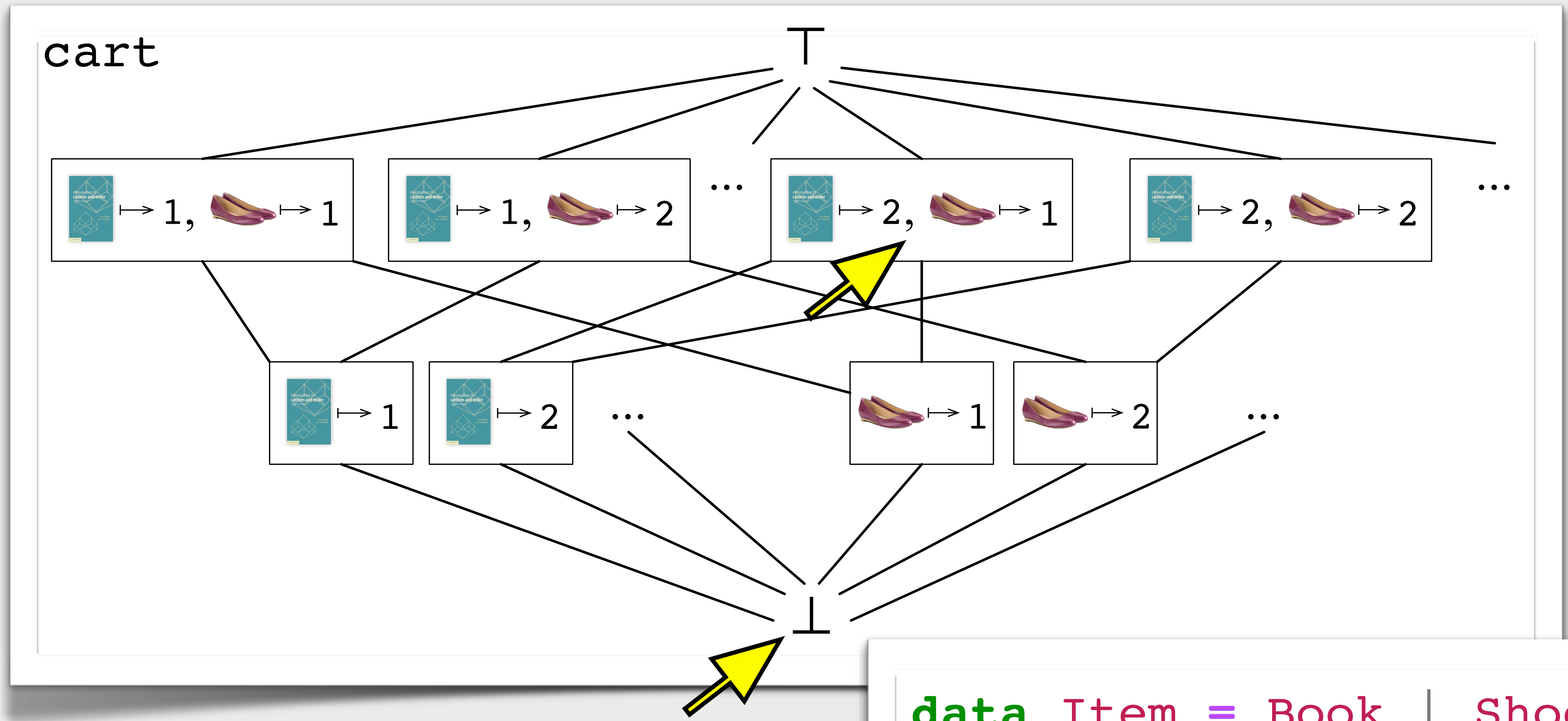
```
data Item = Book | Shoes | ...

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



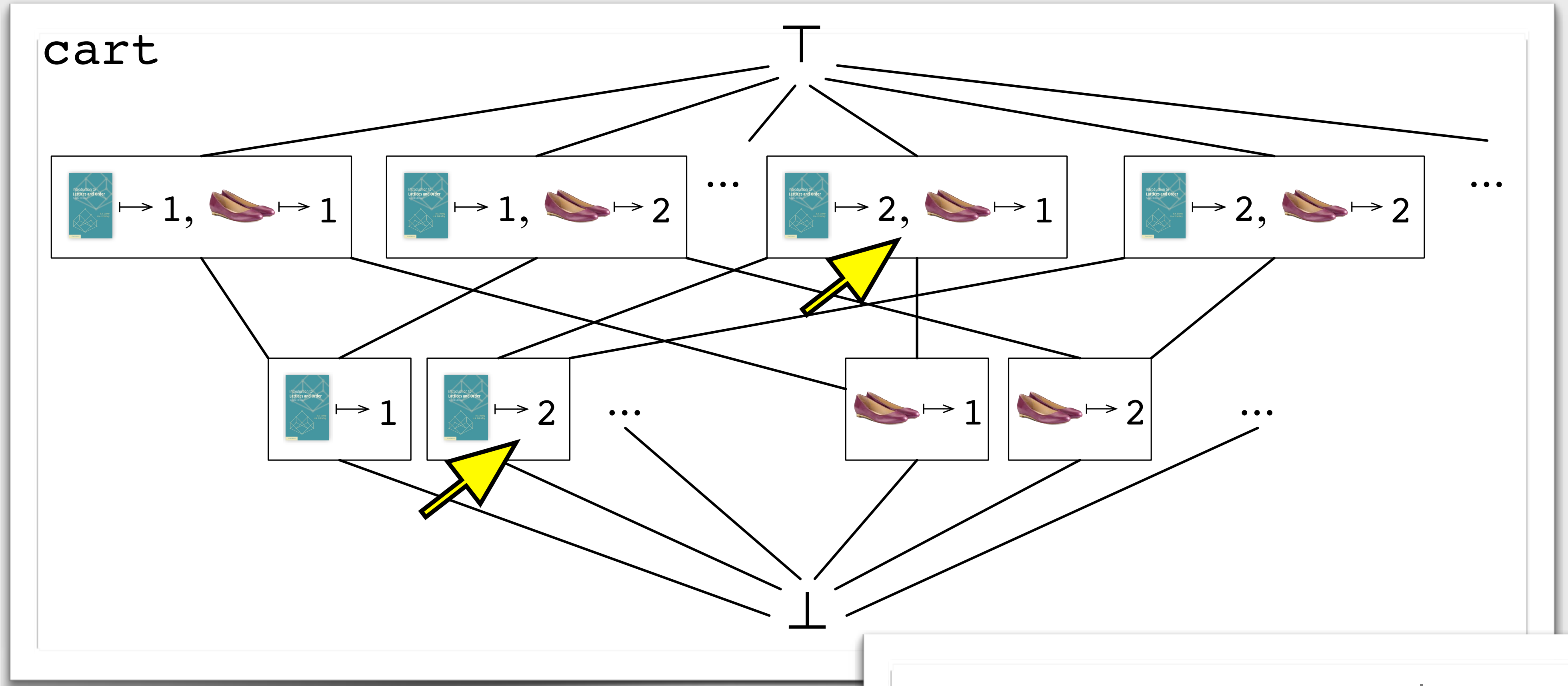
```
data Item = Book | Shoes | ...

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



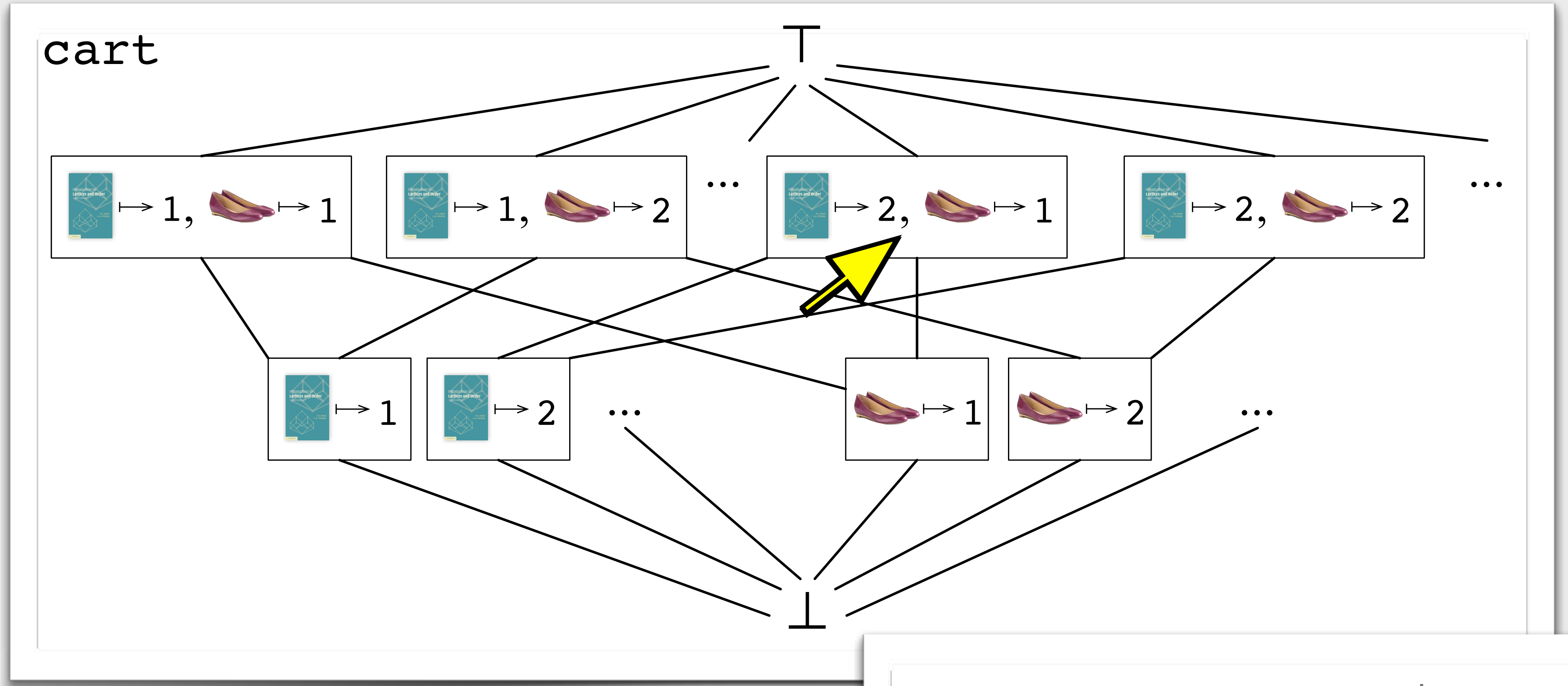
```
data Item = Book | Shoes | ...

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



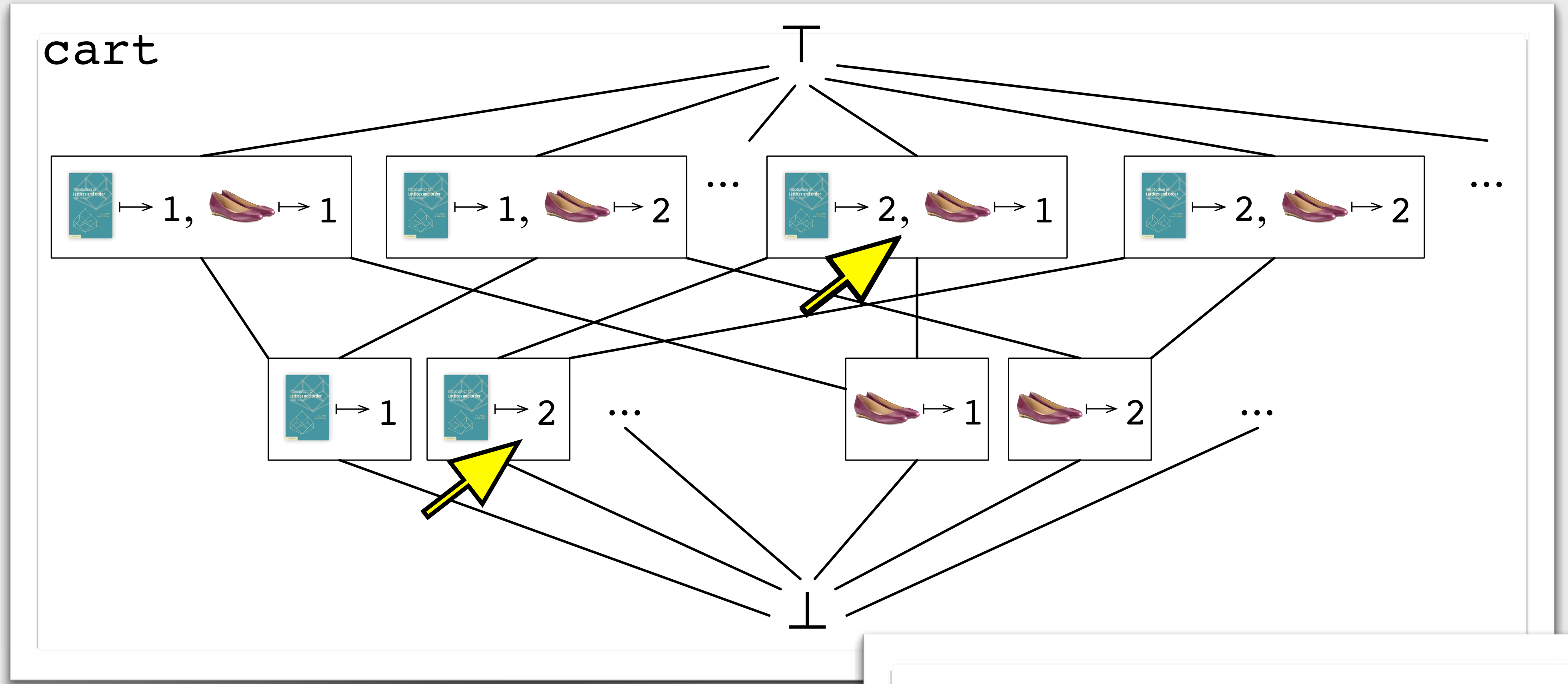
```
data Item = Book | Shoes | ...

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



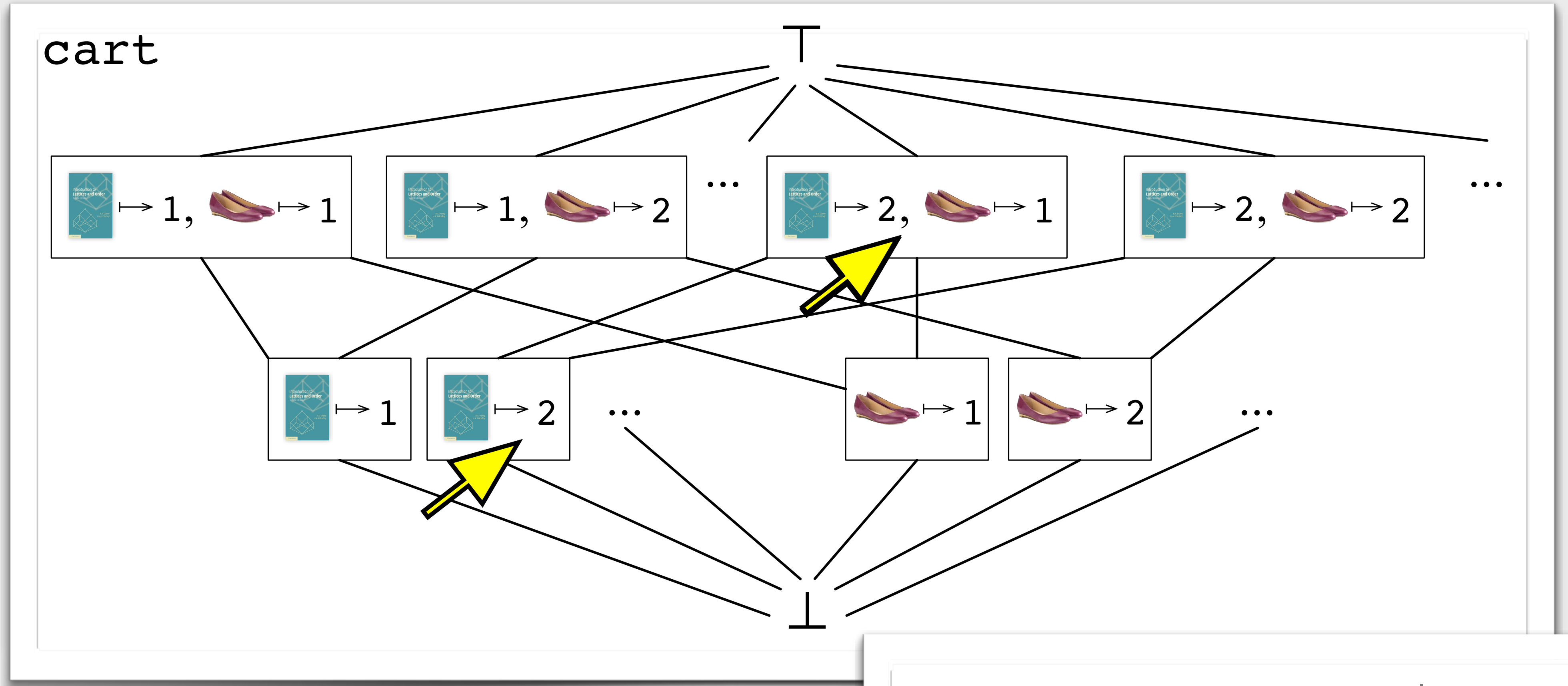
```
data Item = Book | Shoes | ...

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
```

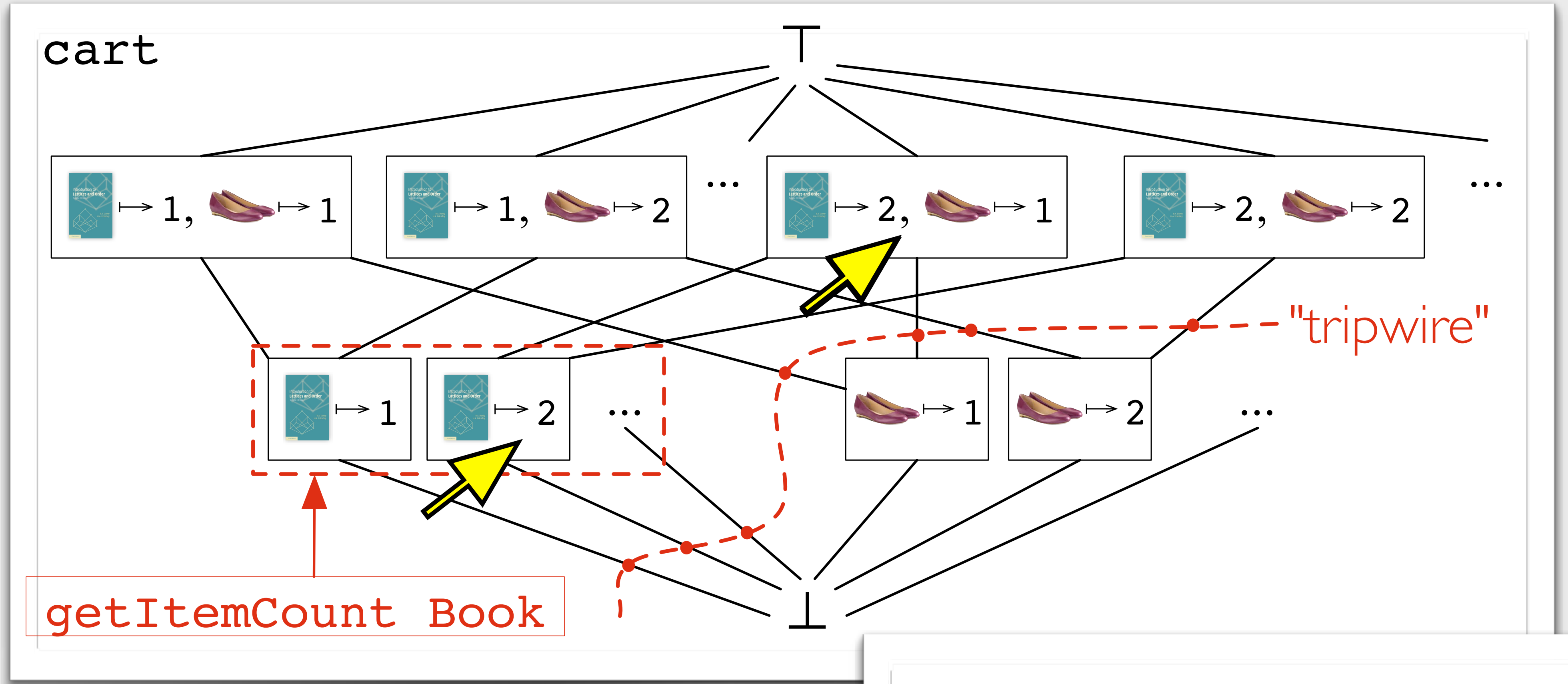


```
data Item = Book | Shoes | ...

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



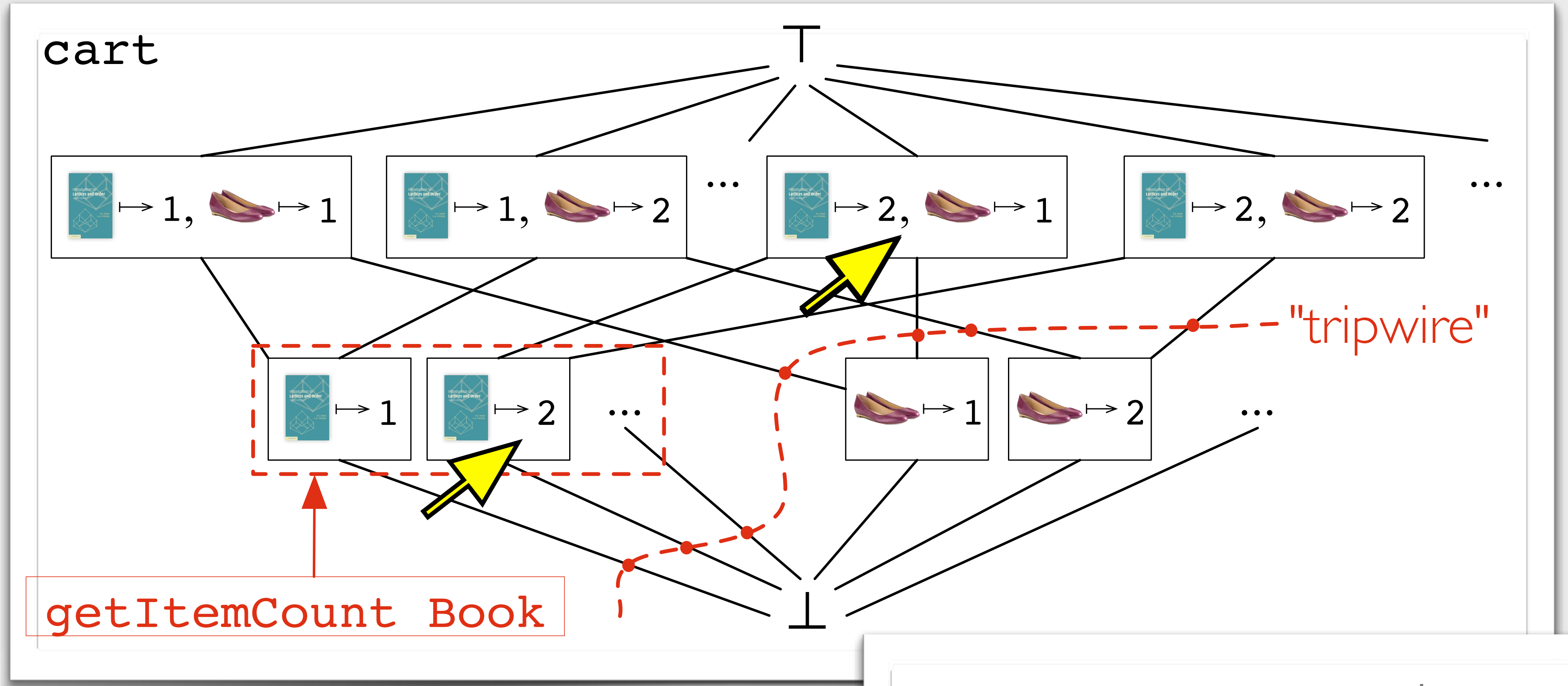
$\{$(Book,1), (Book,2), …$\}$

```
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), ...}

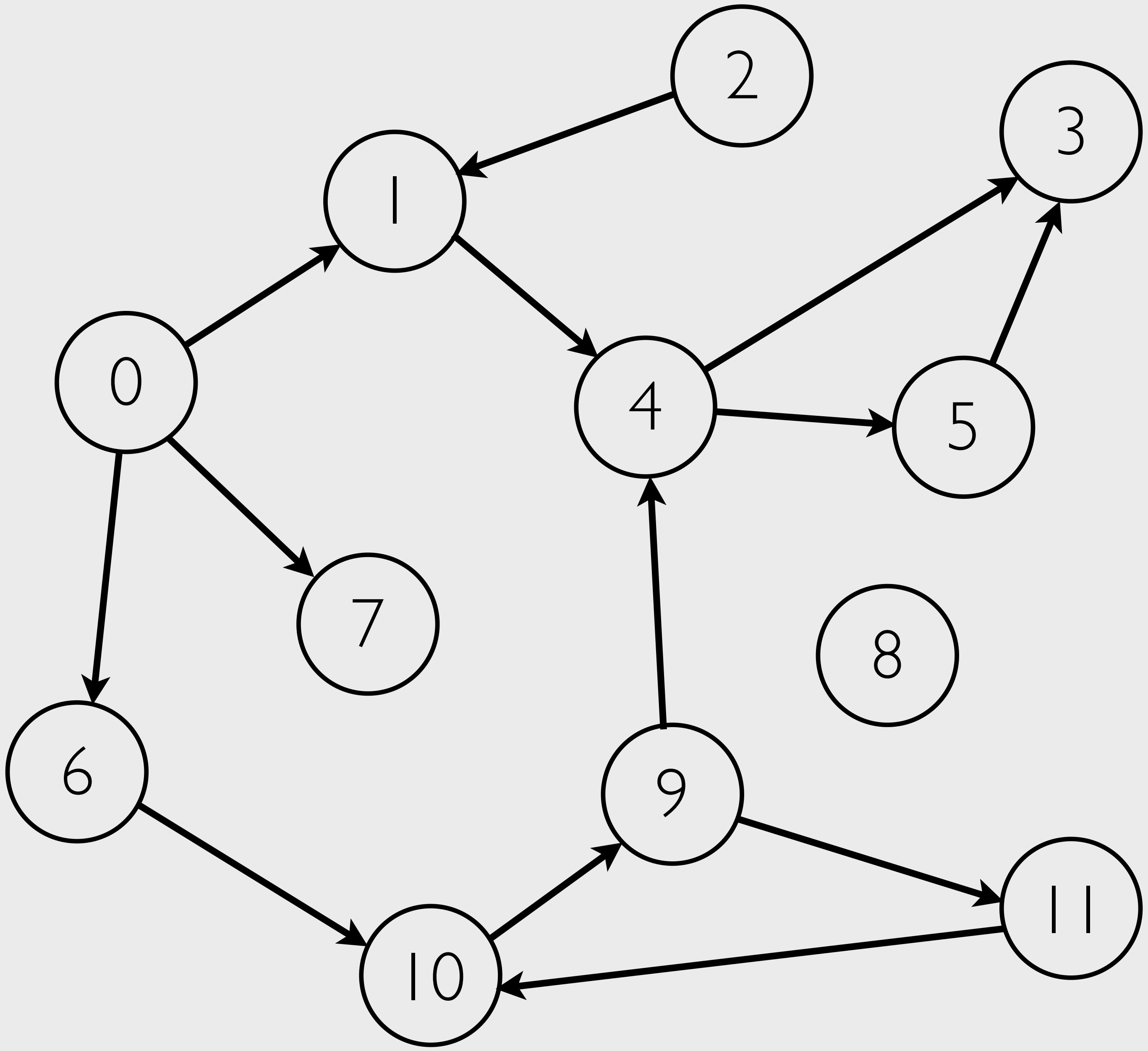
```
data Item = Book | Shoes | ...

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



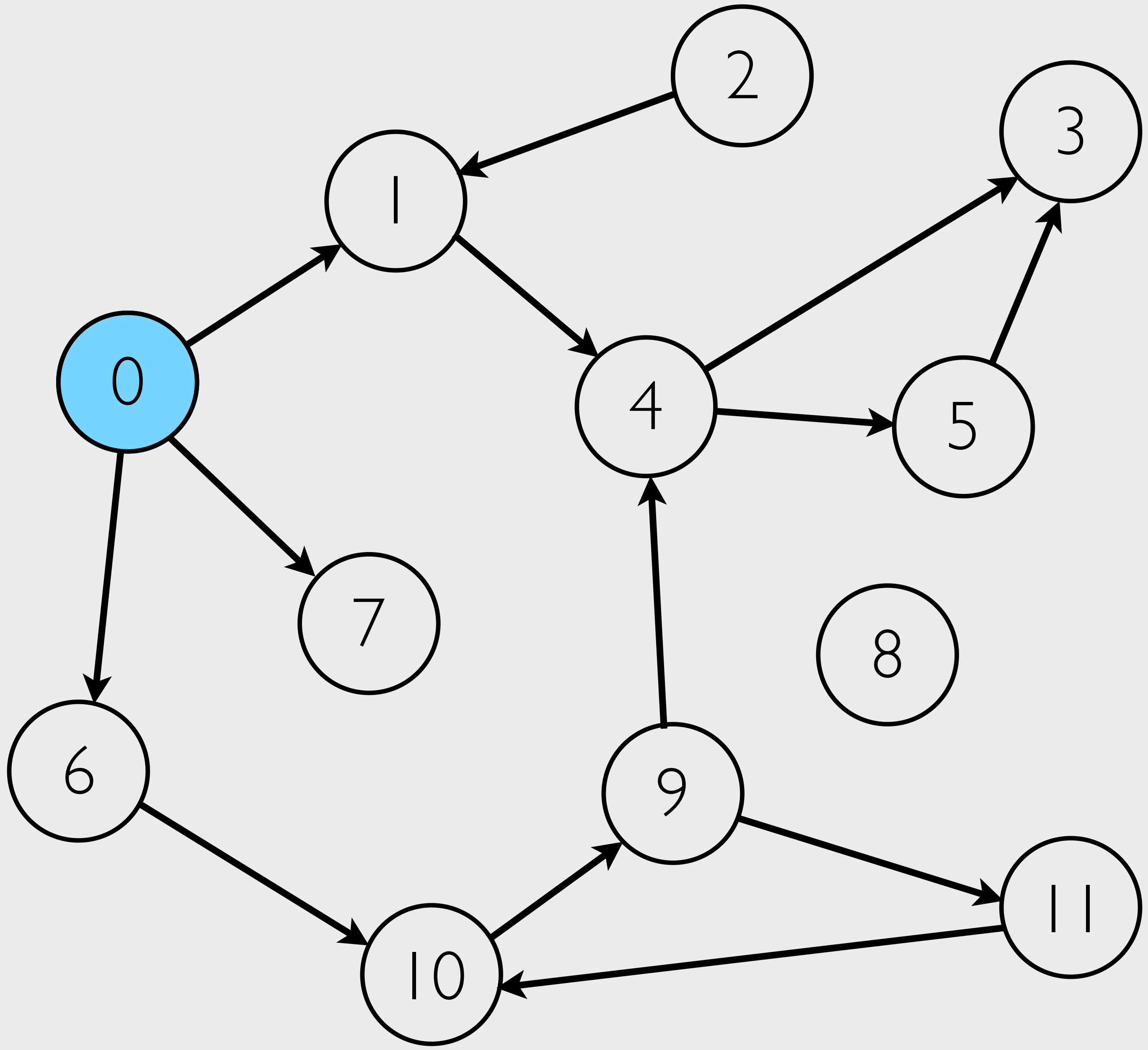
{(Book,1), (Book,2), …}

The threshold set must be *pairwise incompatible*

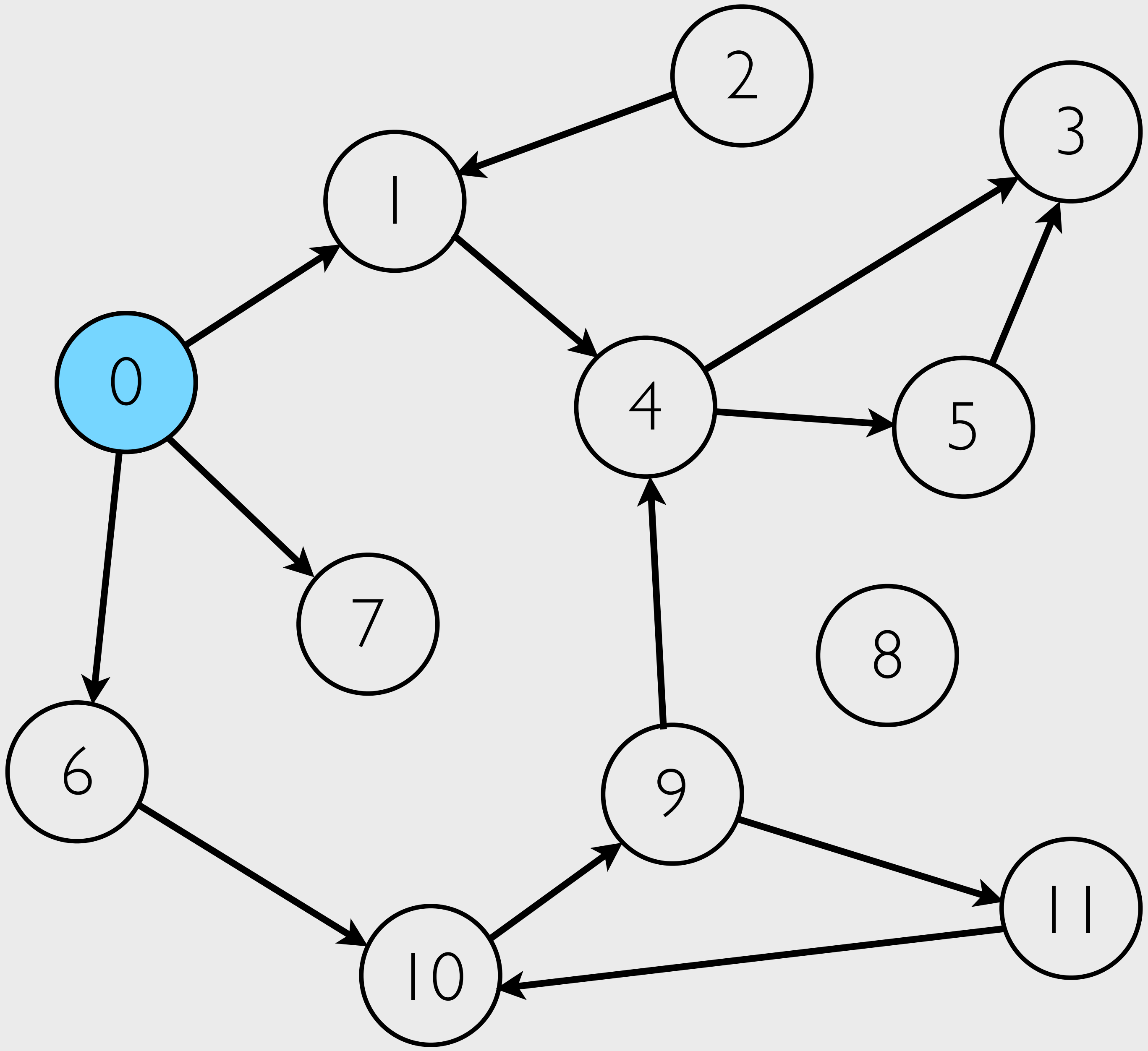
```
data Item = Book | Shoes | ...

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

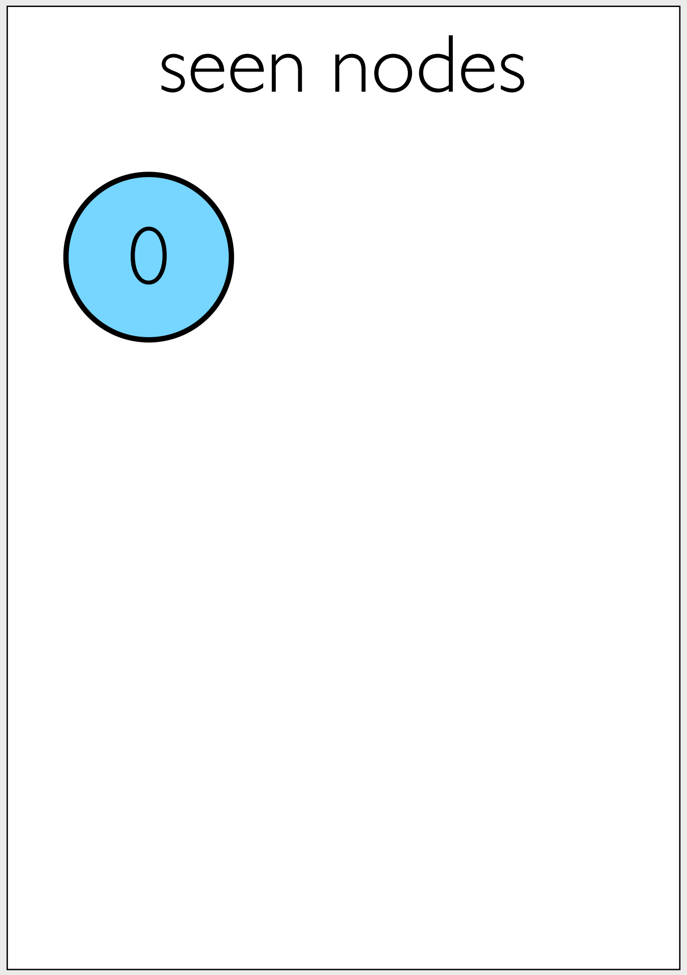


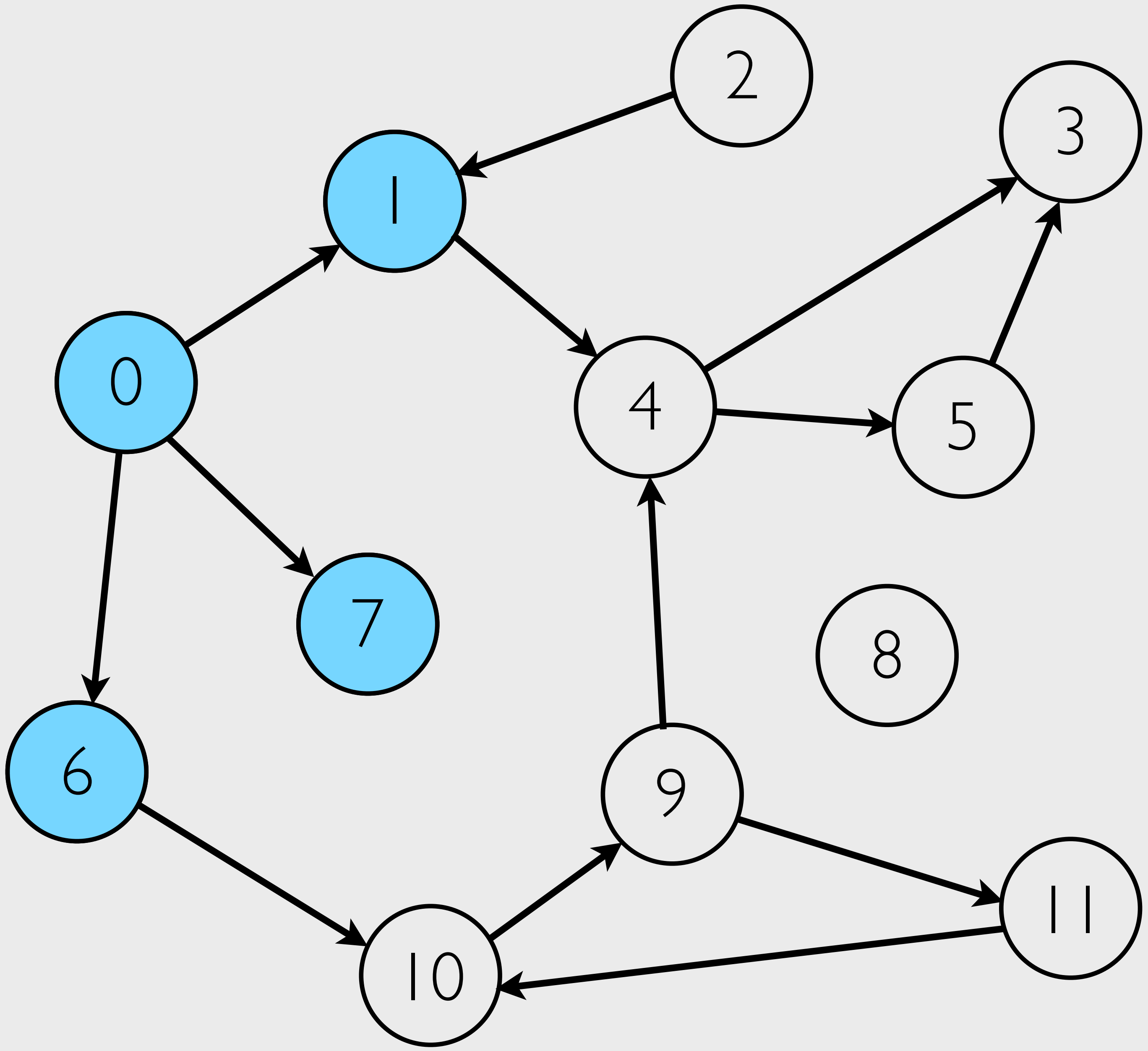
seen nodes

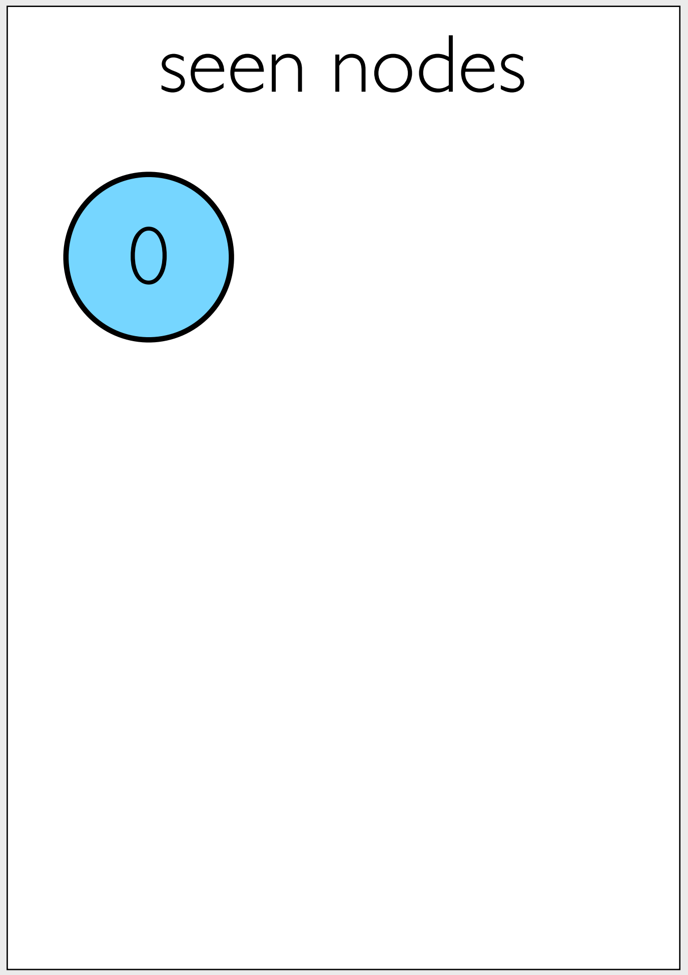


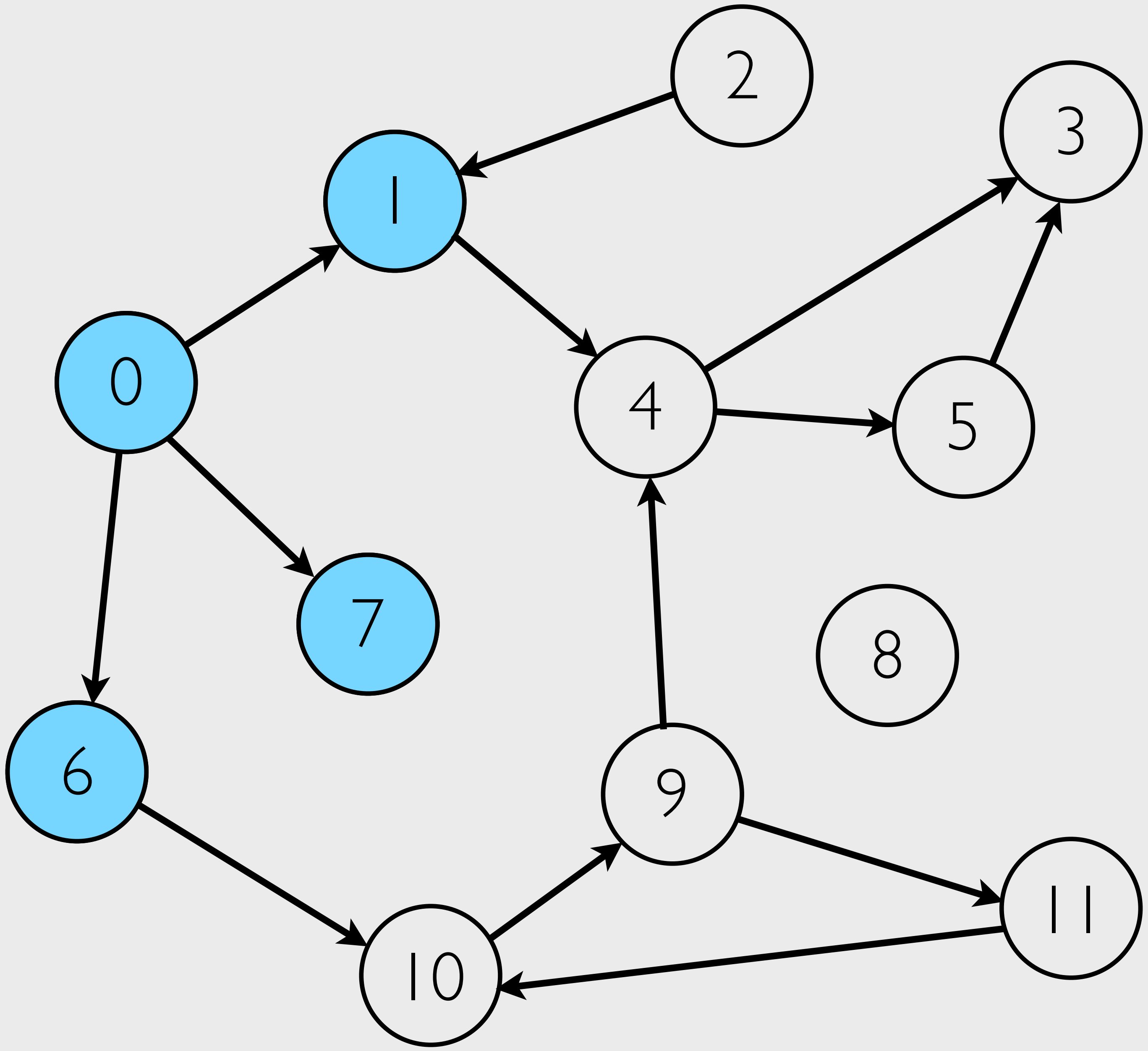
seen nodes

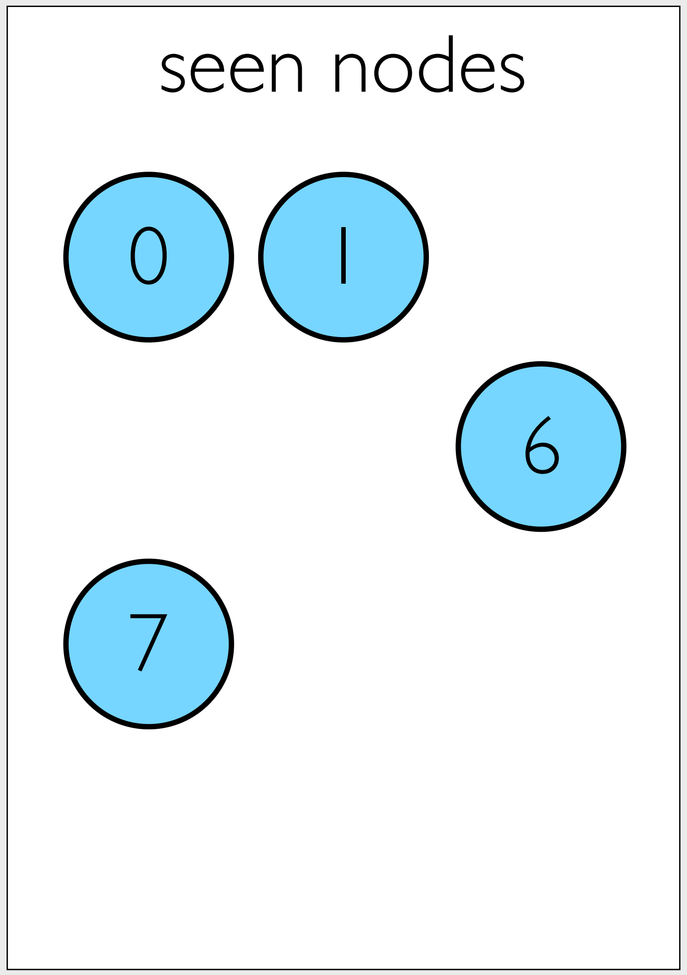

0
1
2
3
4
5
6
7
8
9
10
11

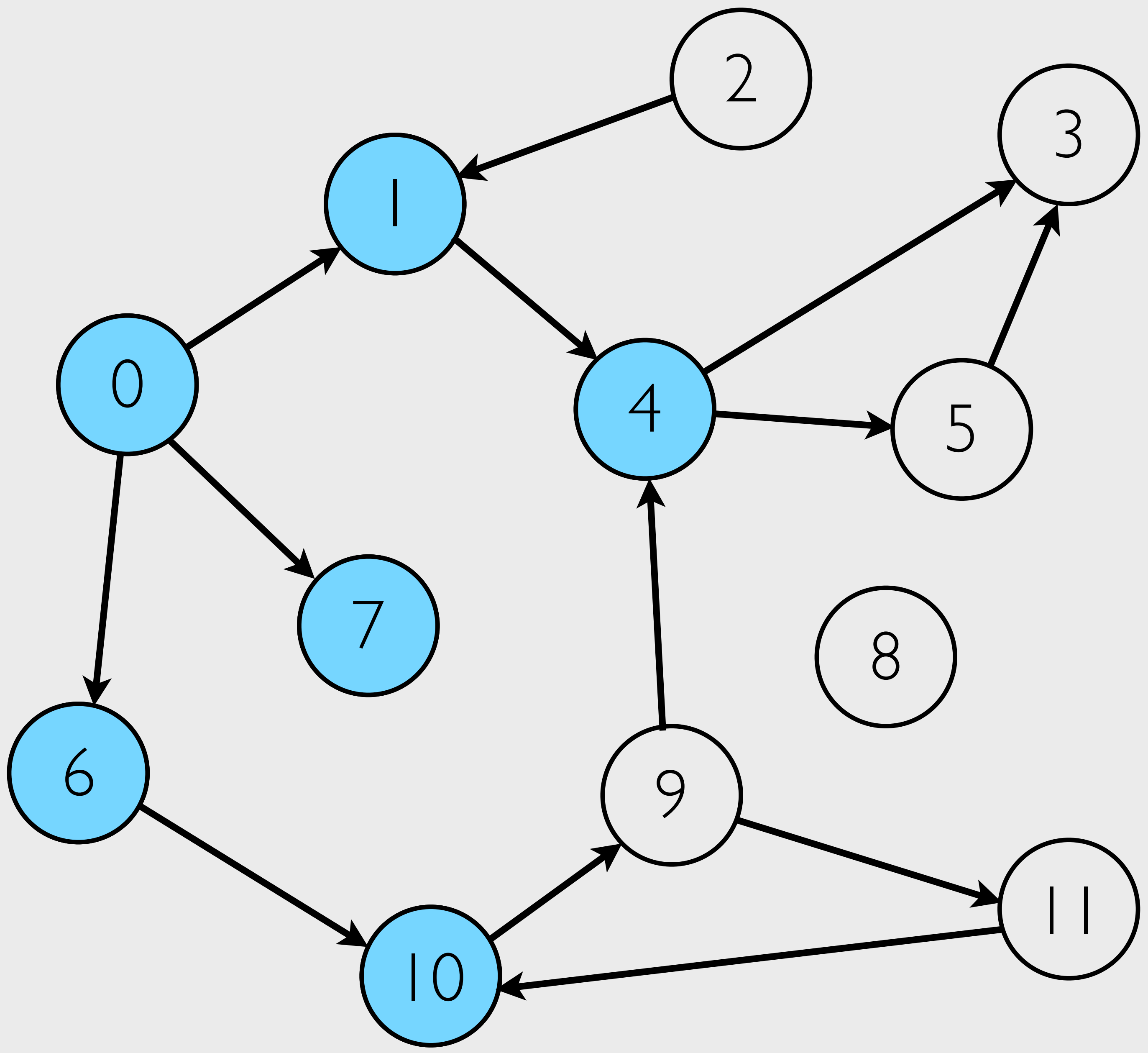

seen nodes
0

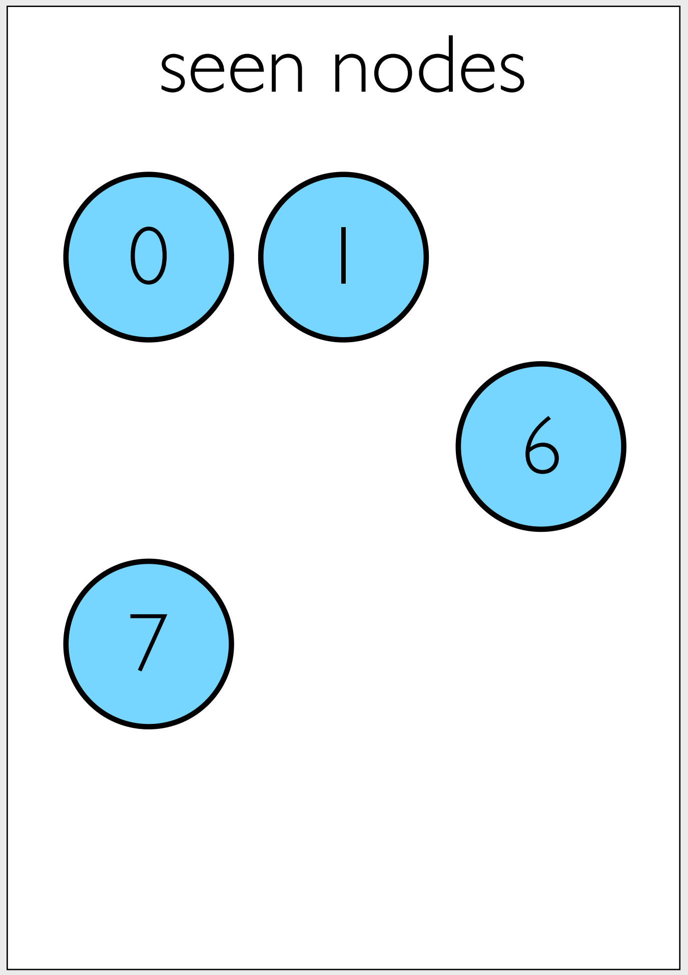

0
1
2
3
4
5
6
7
8
9
10
11

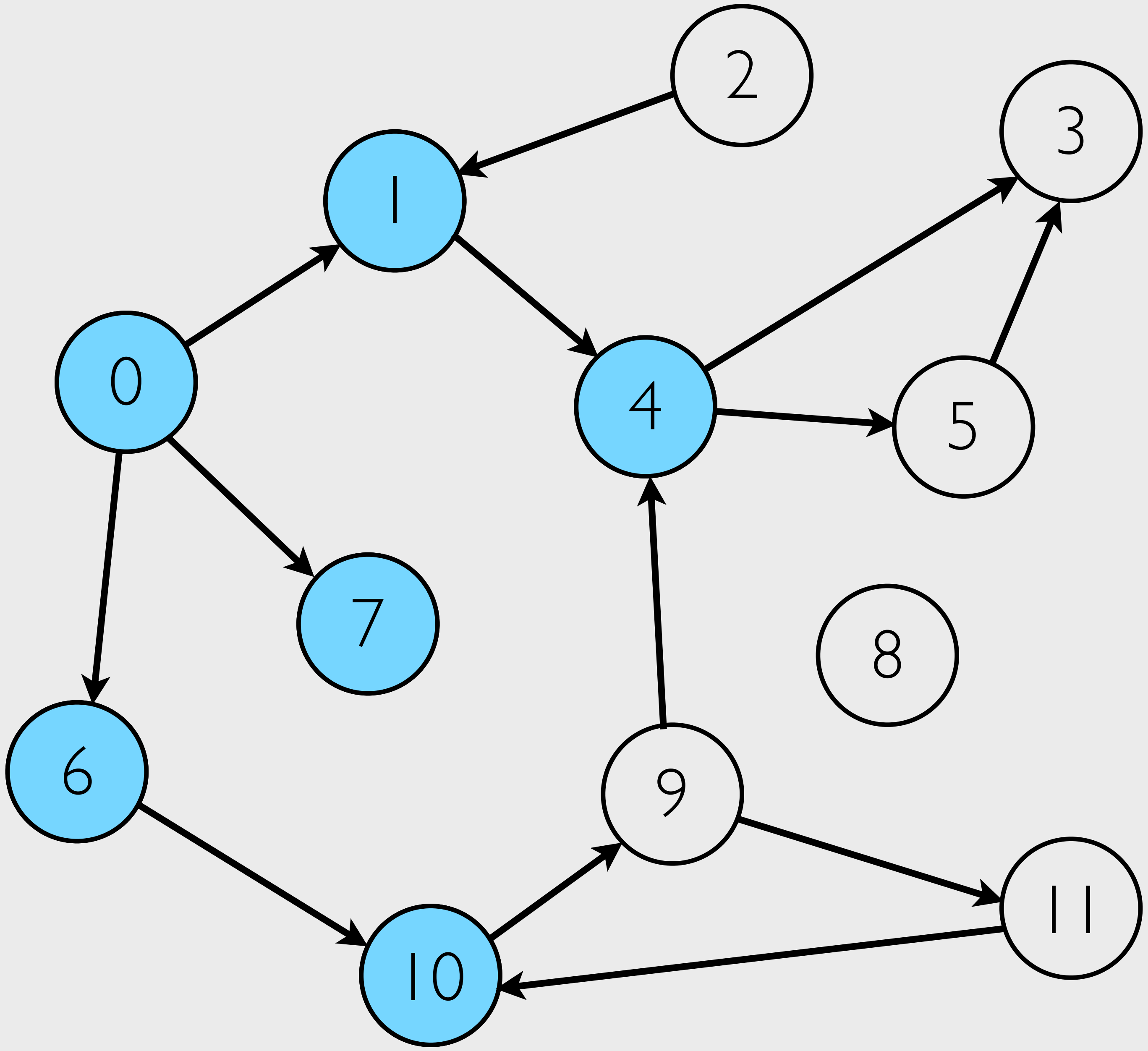

seen nodes
0

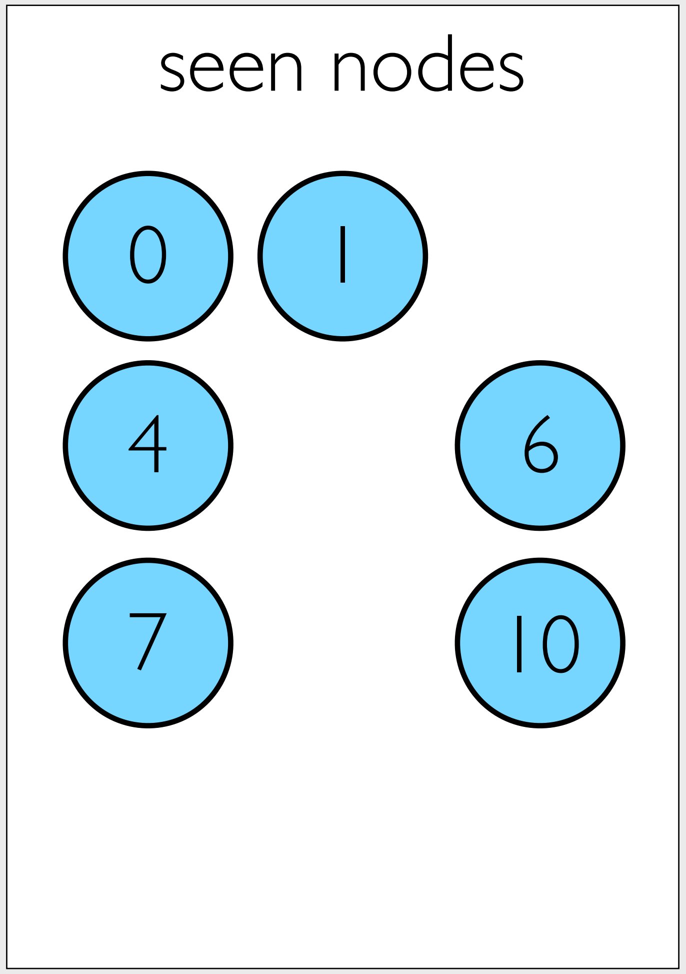

0
1
2
3
4
5
6
7
8
9
10
11

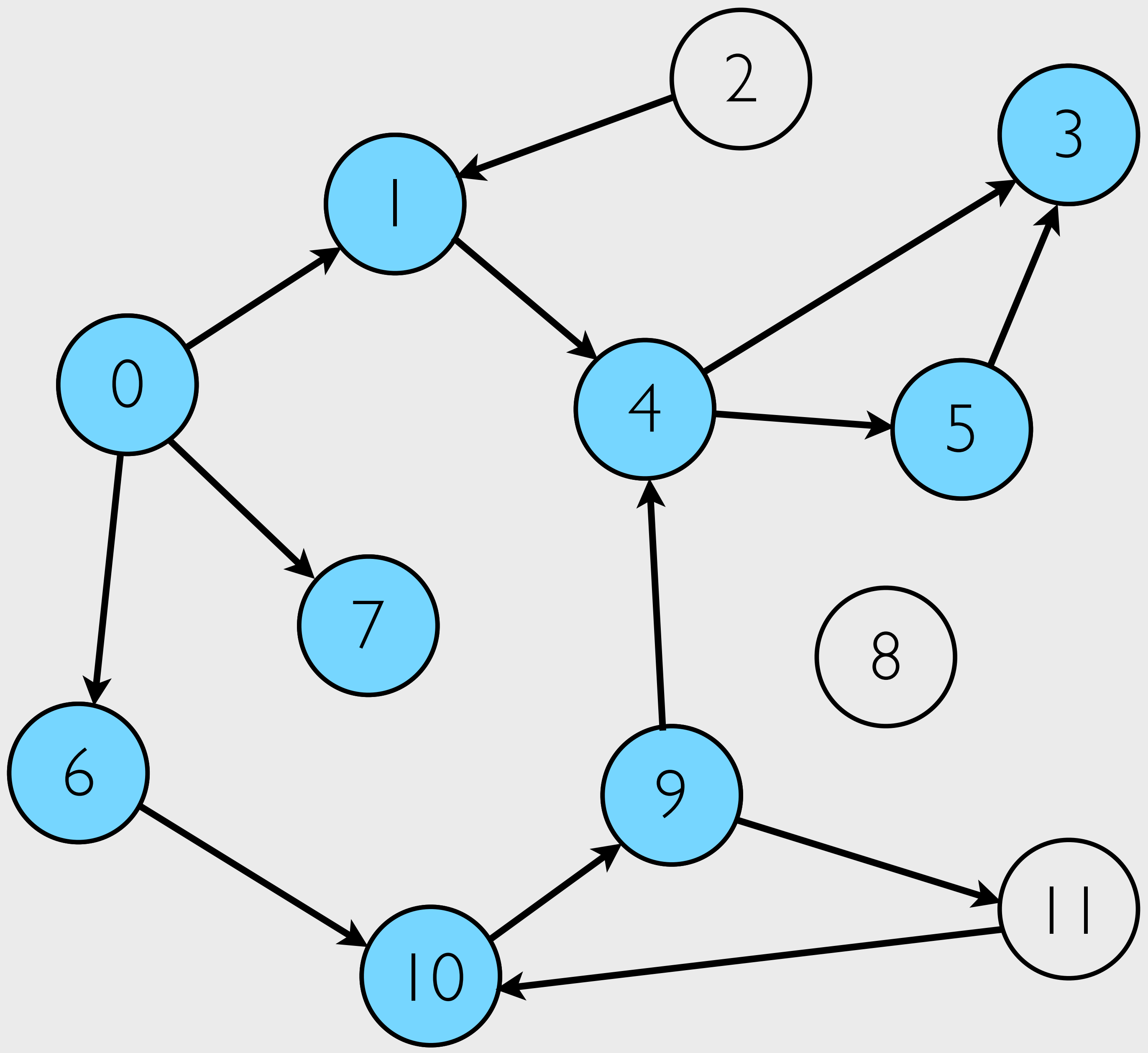

seen nodes
0
1
6
7

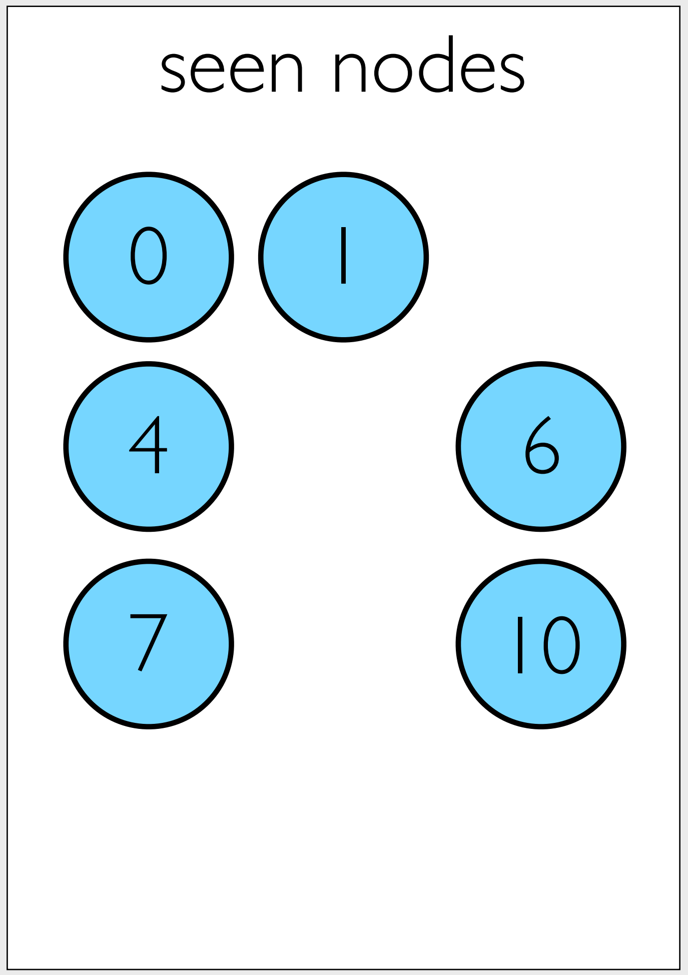

2
1
3
0
4
5
7
8
6
9
10
11

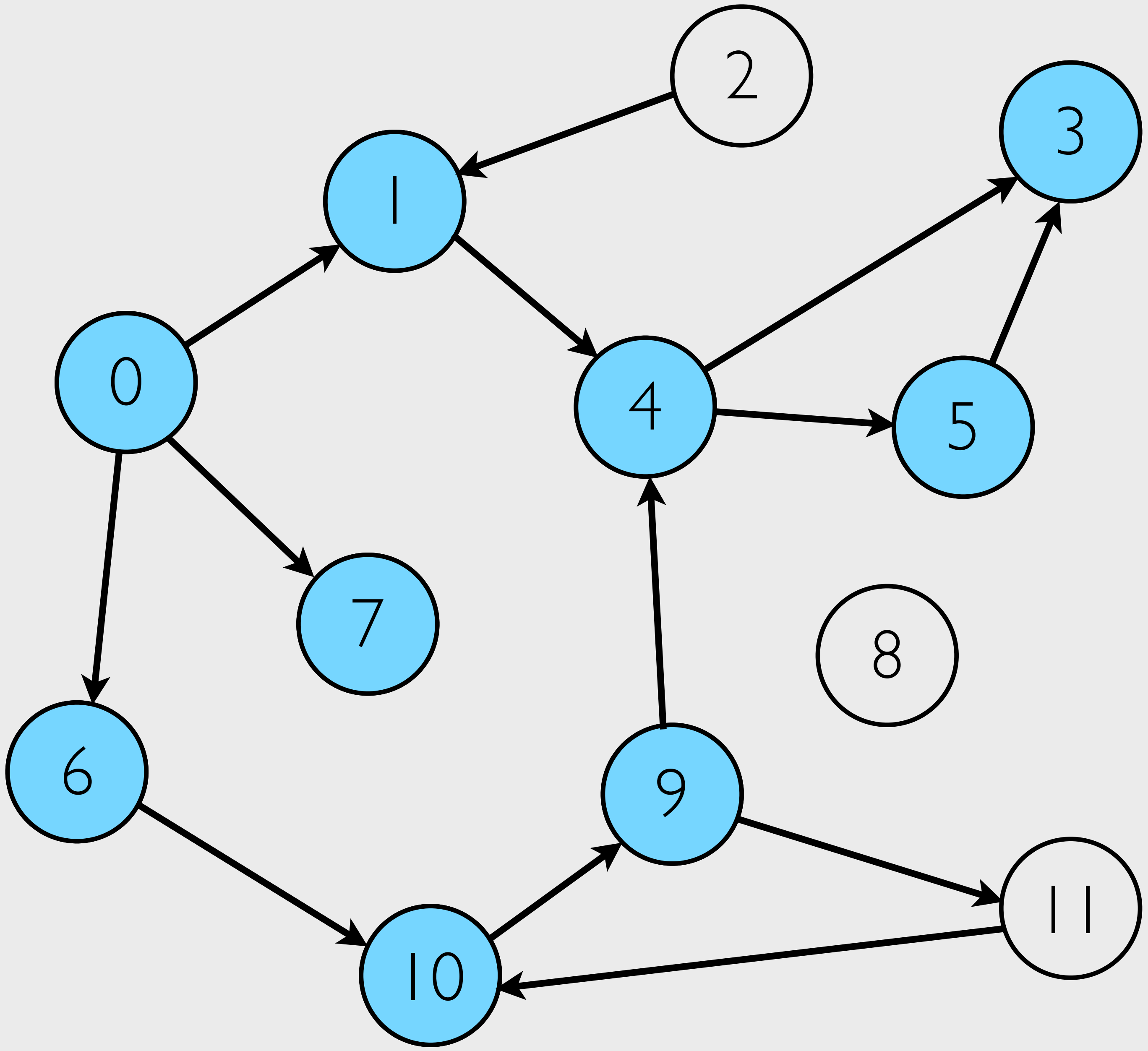

seen nodes
0
1
6
7

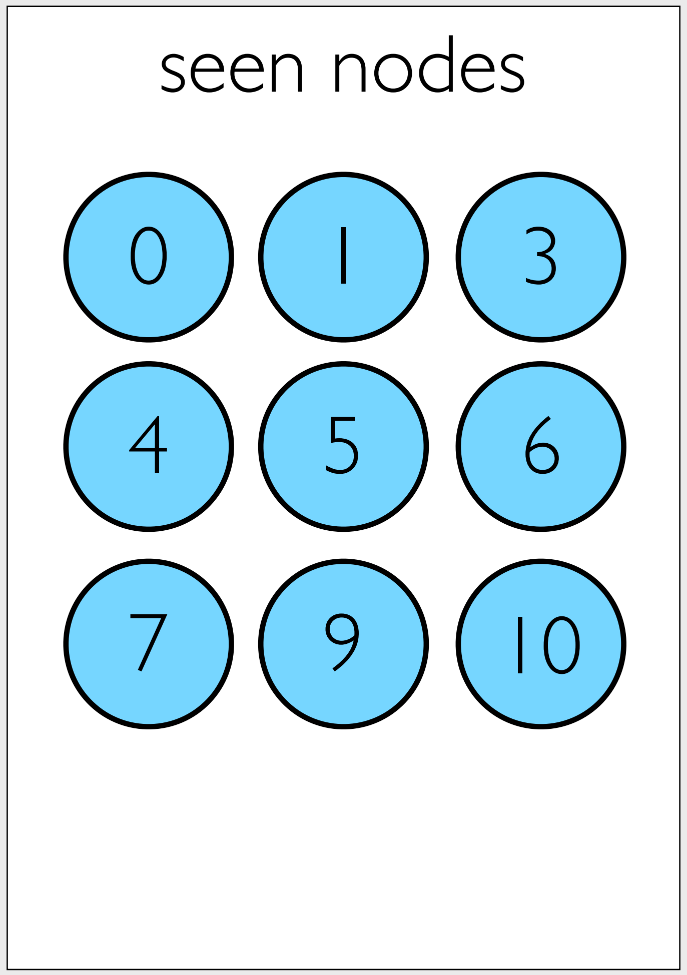

0
1
2
3
4
5
6
7
8
9
10
11

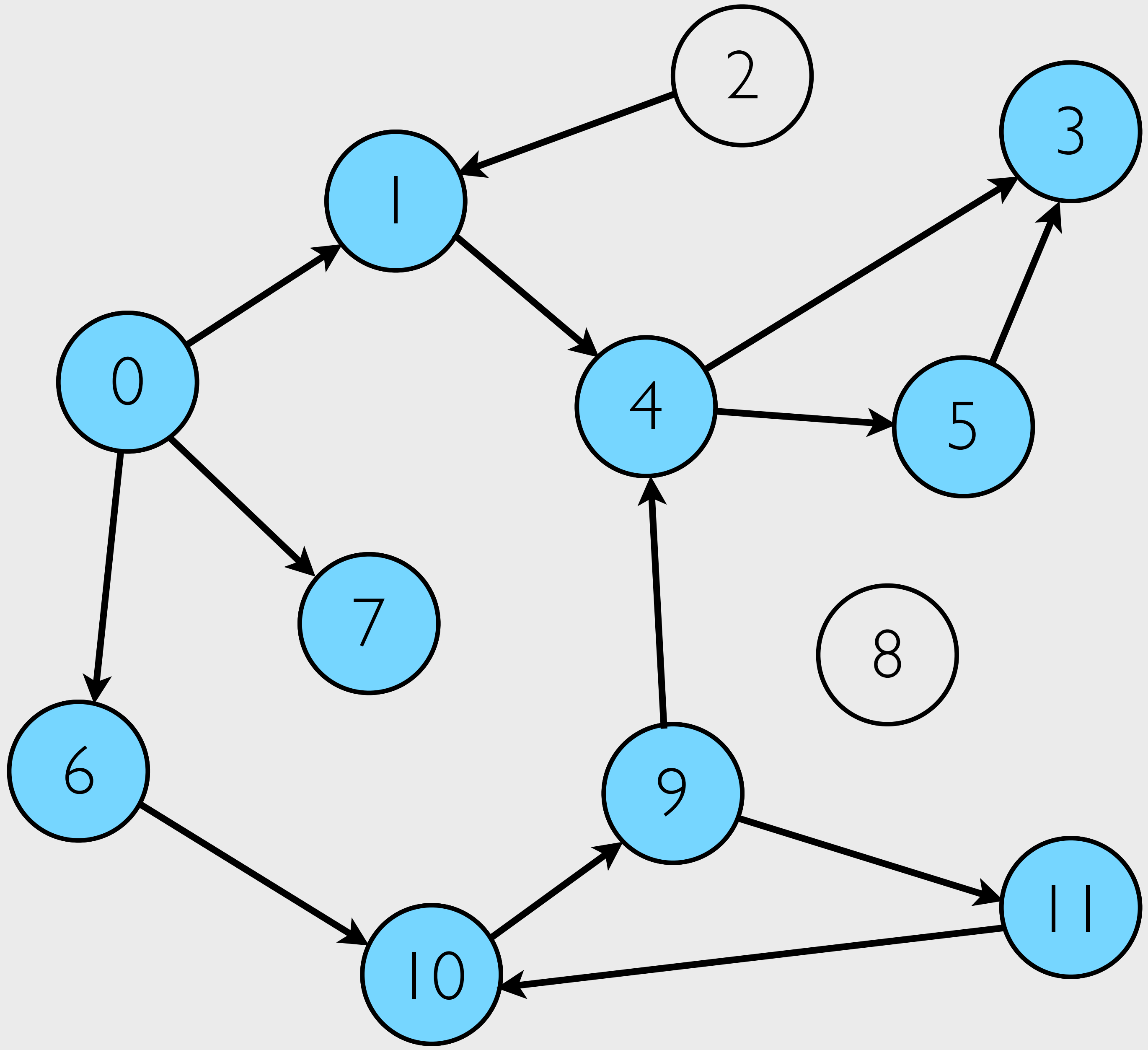

seen nodes
0
1
4
6
7
10

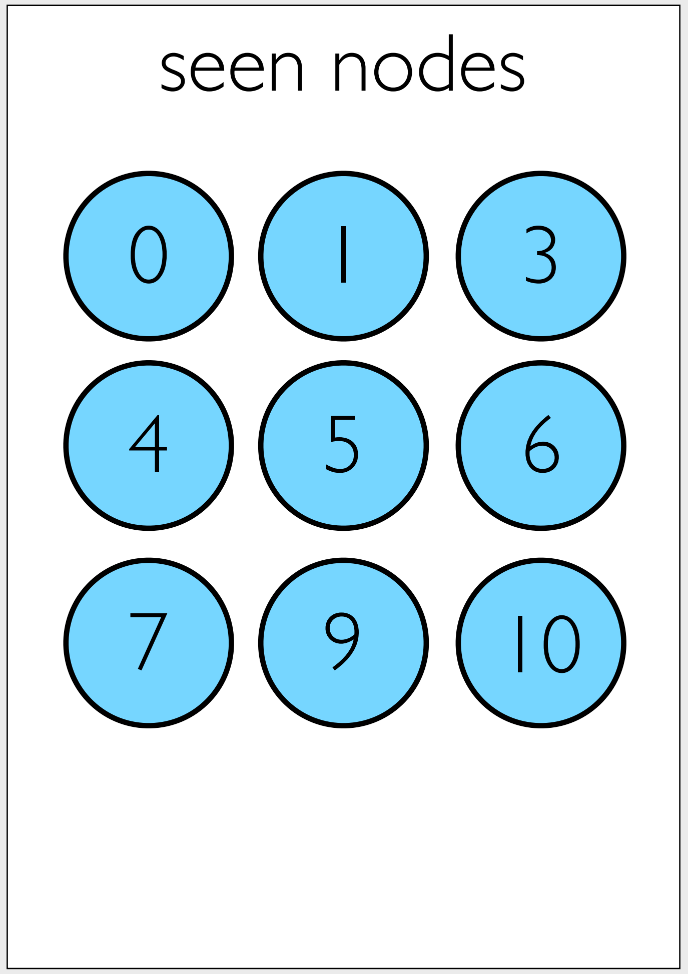

0
1
2
3
4
5
6
7
8
9
10
11

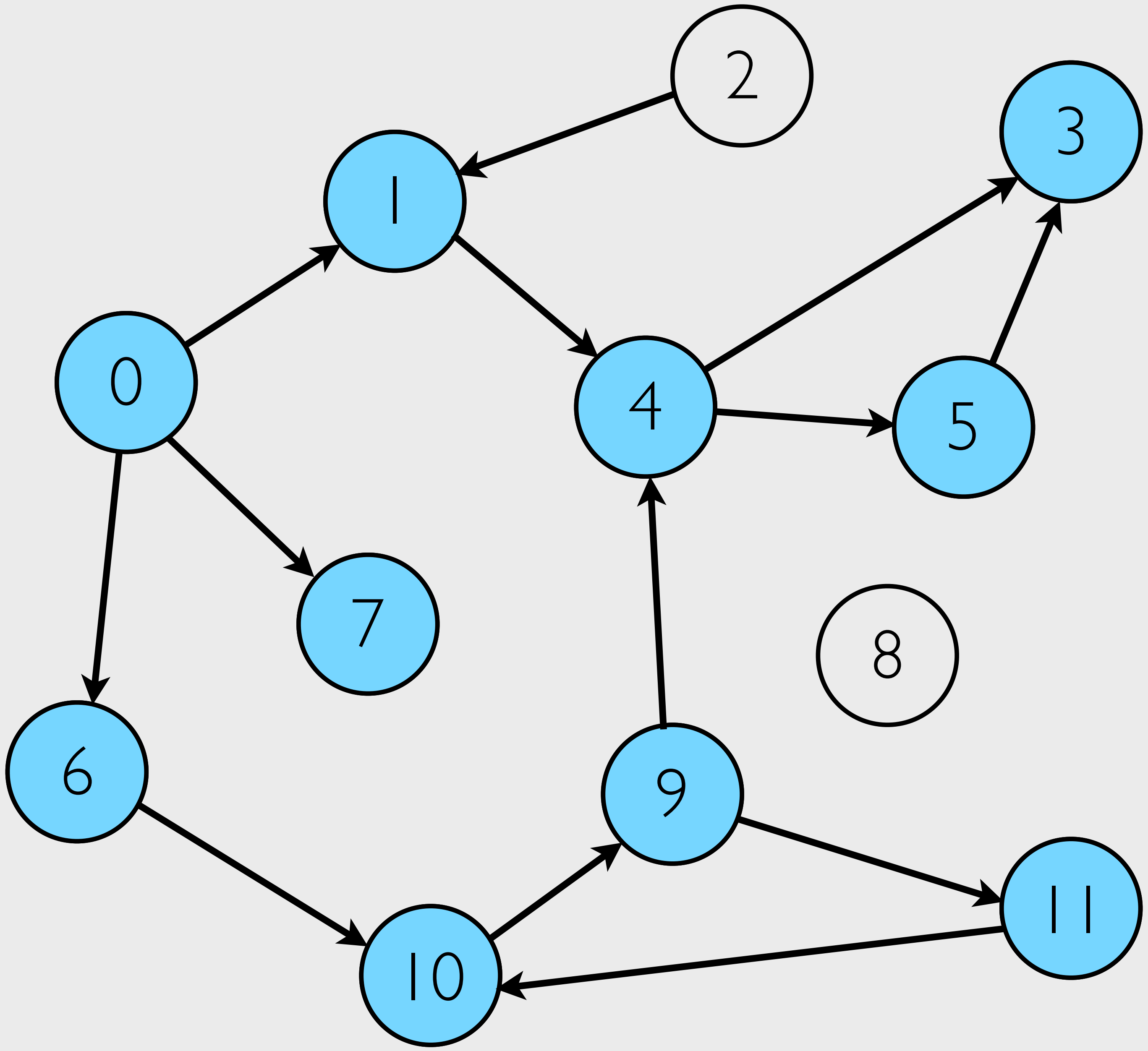

seen nodes
0
1
4
6
7
10

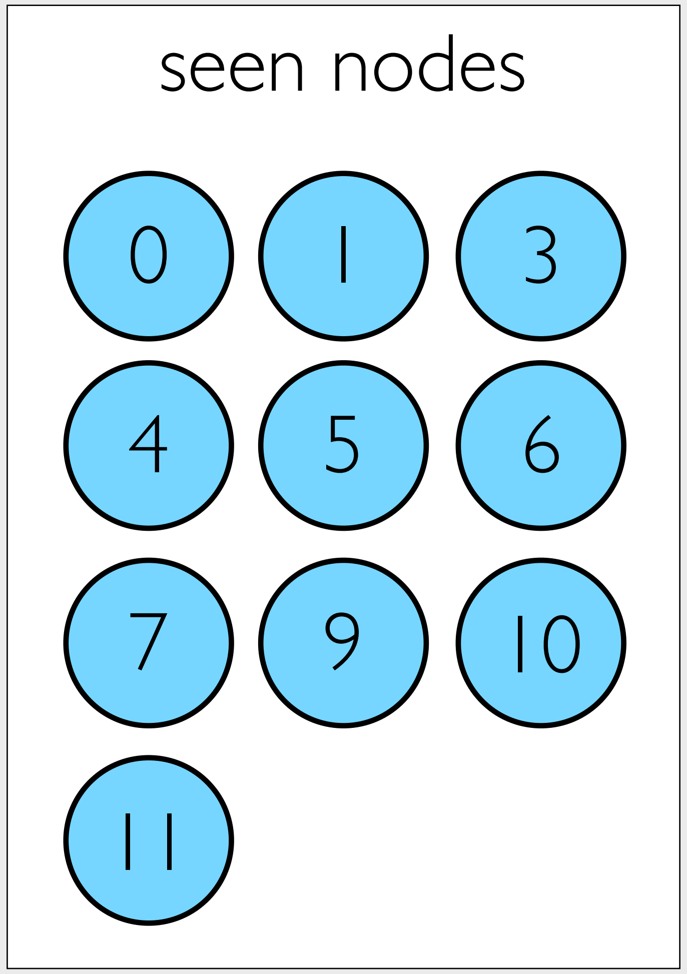

0
1
2
3
4
5
6
7
8
9
10
11

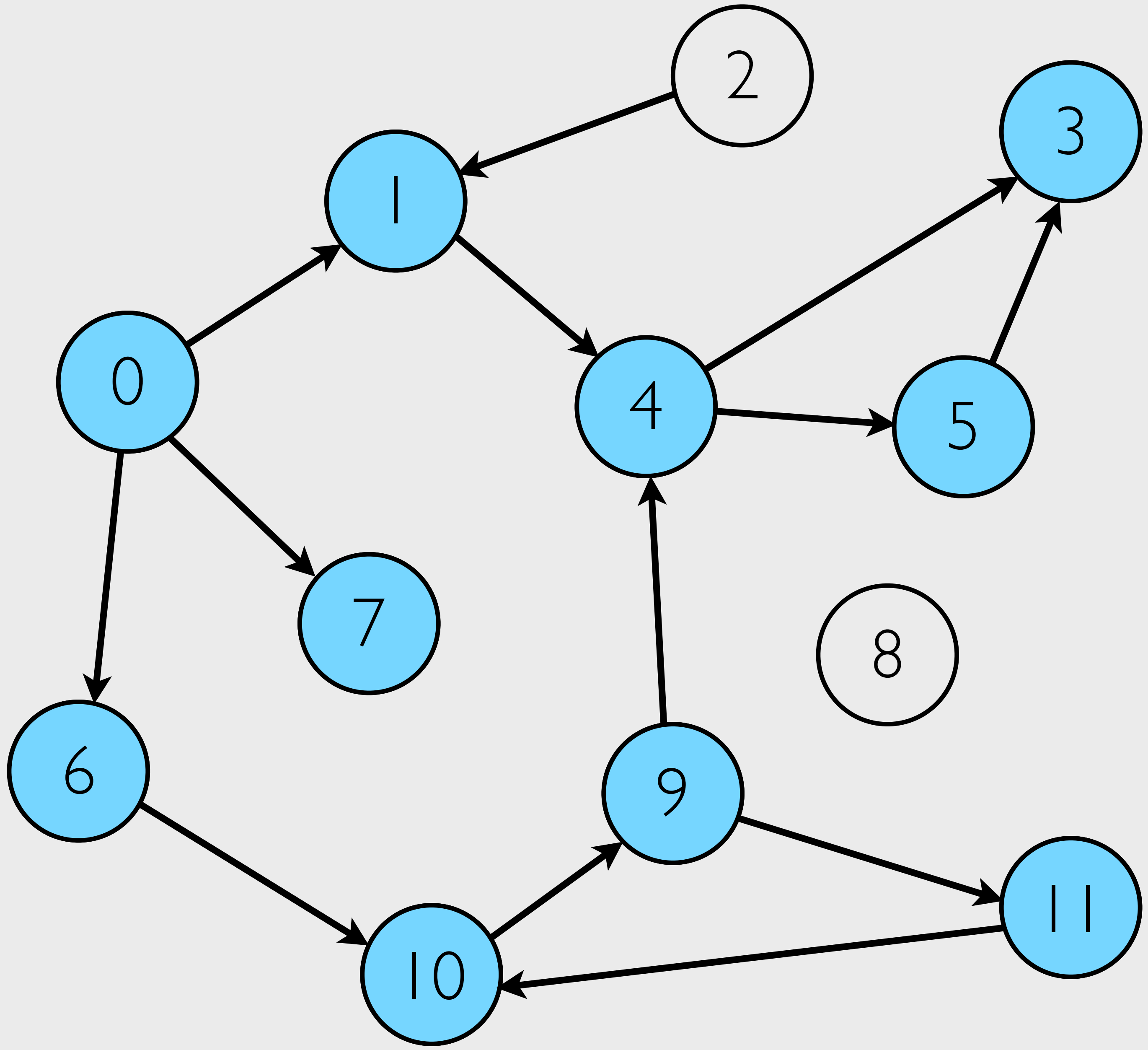

seen nodes
0
1
3
4
5
6
7
9
10

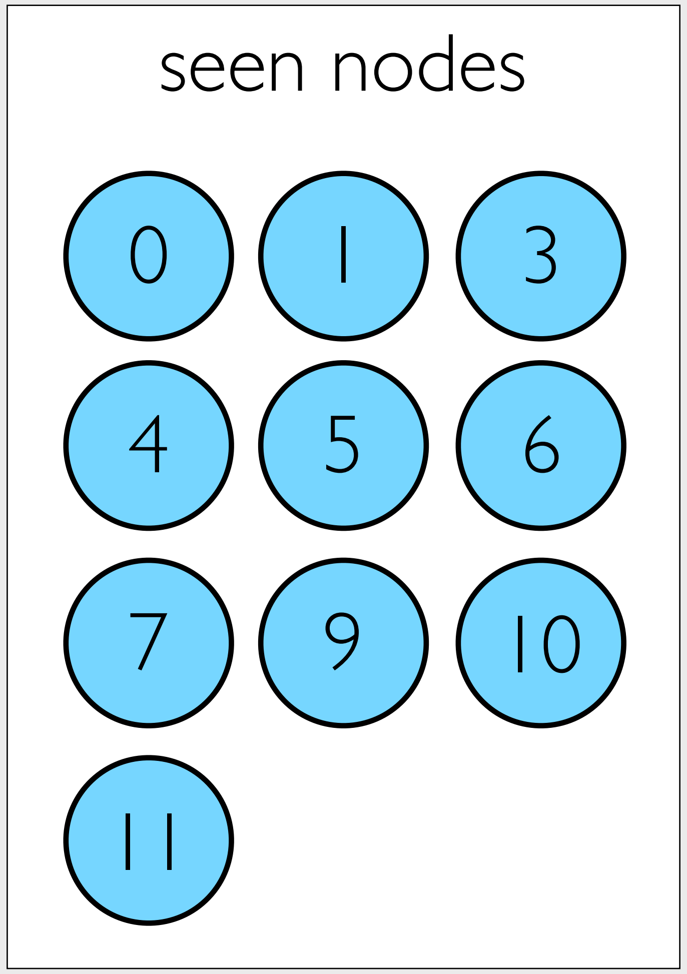

2
1
3
0
4
5
7
8
6
9
10
11

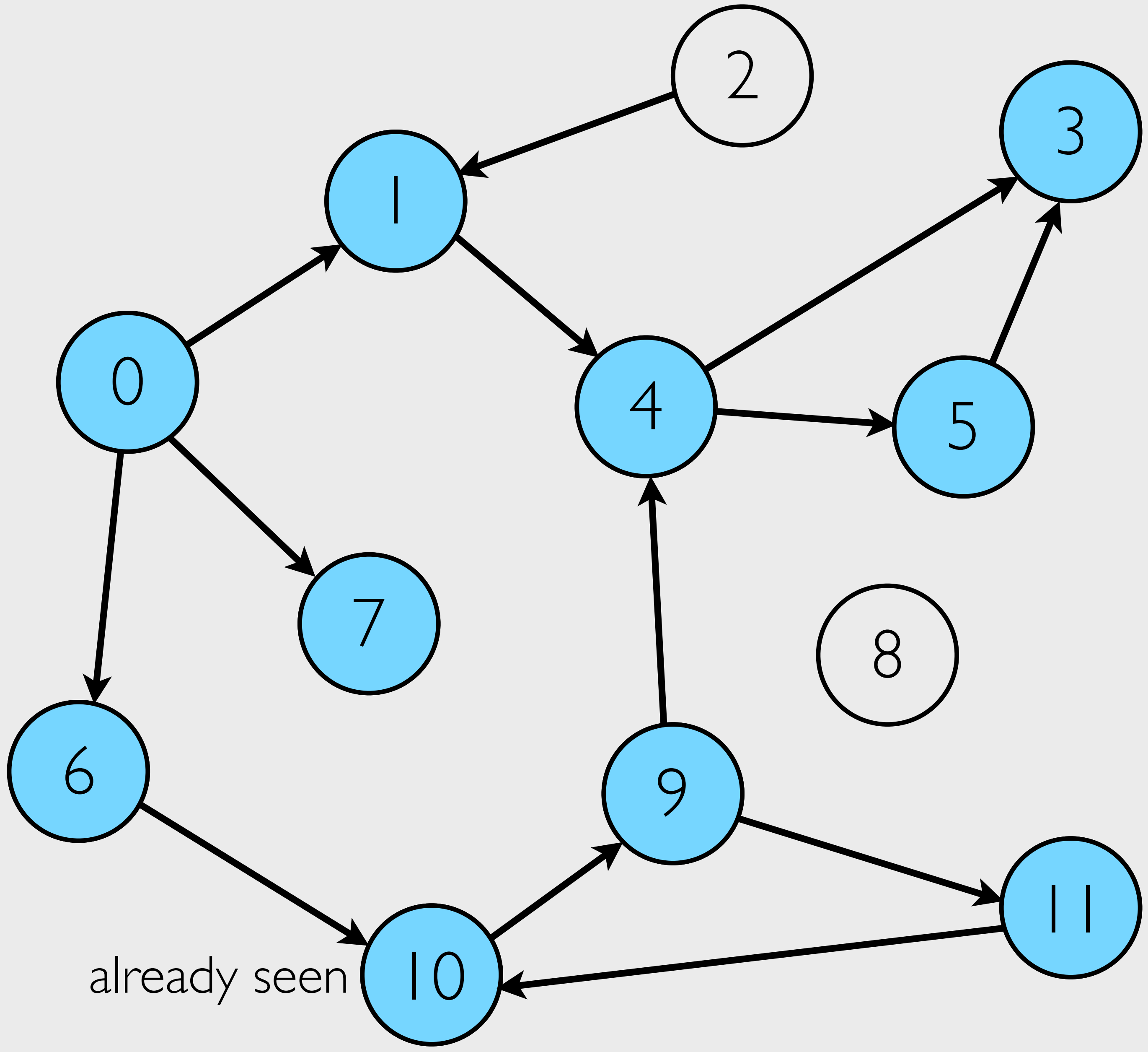

seen nodes
0
1
3
4
5
6
7
9
10


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

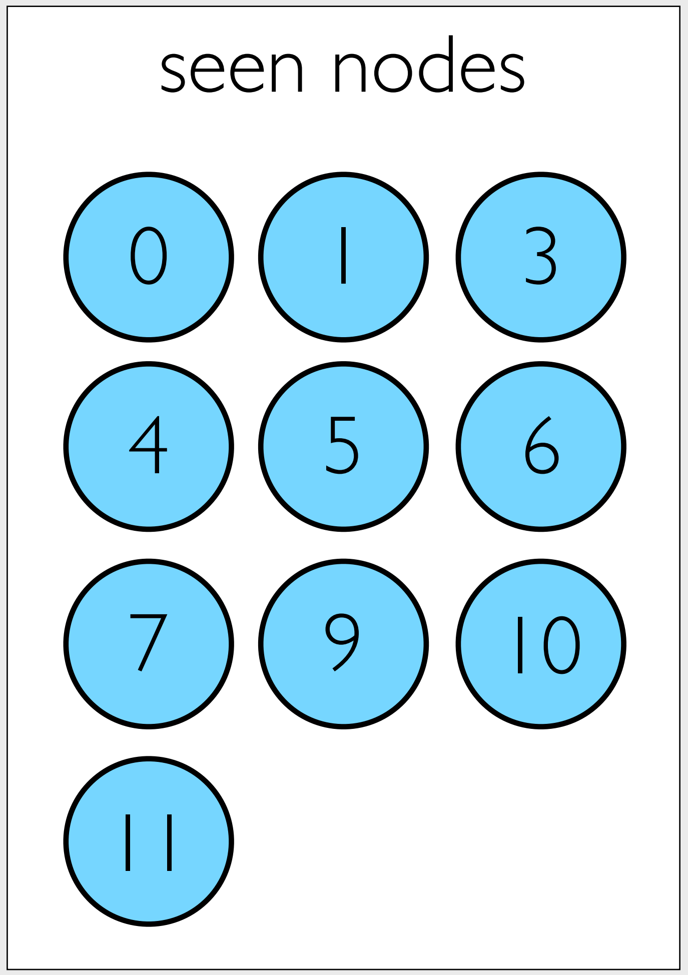

seen nodes
0
1
3
4
5
6
7
9
10
11


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

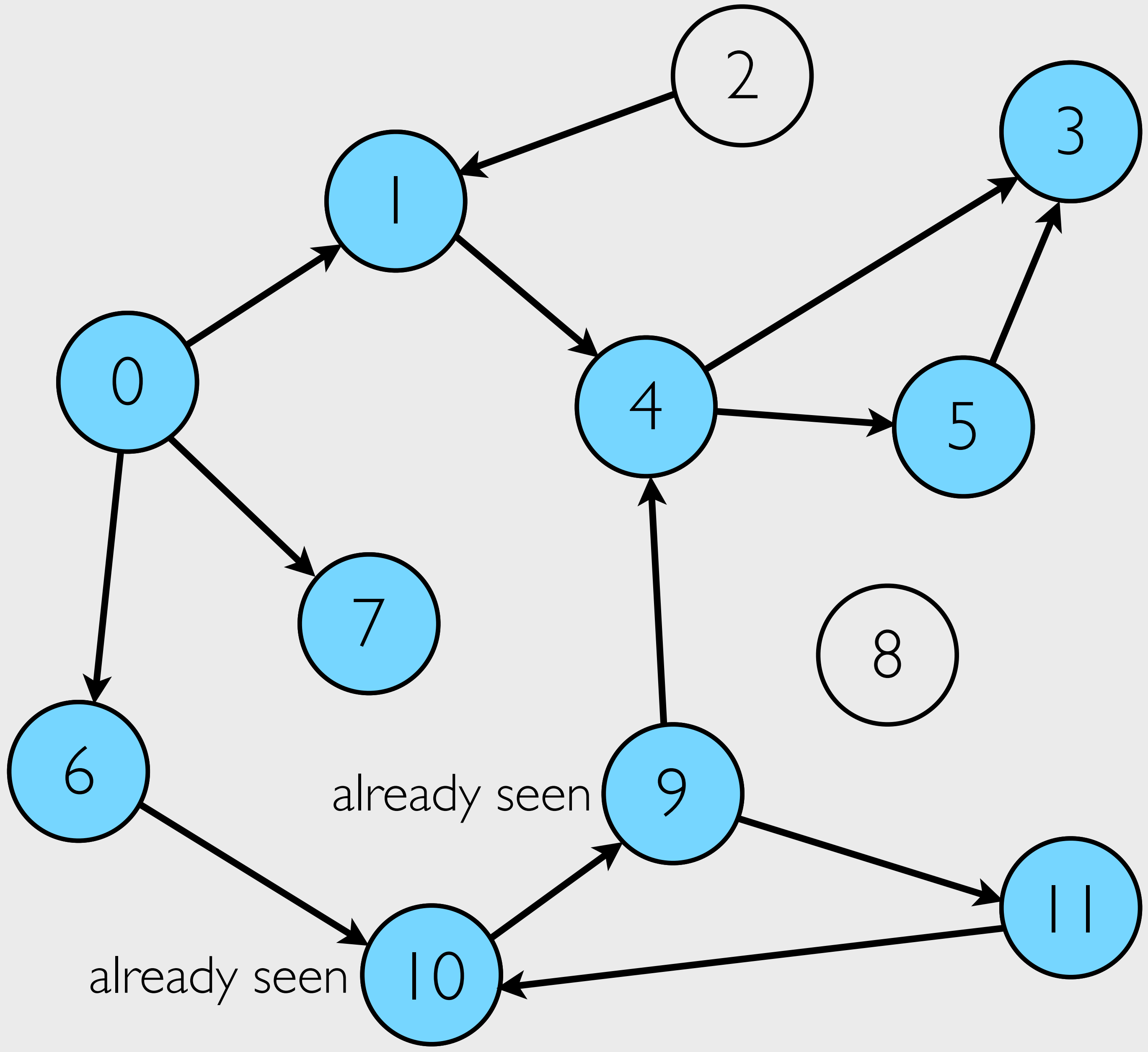

seen nodes
0
1
3
4
5
6
7
9
10
11

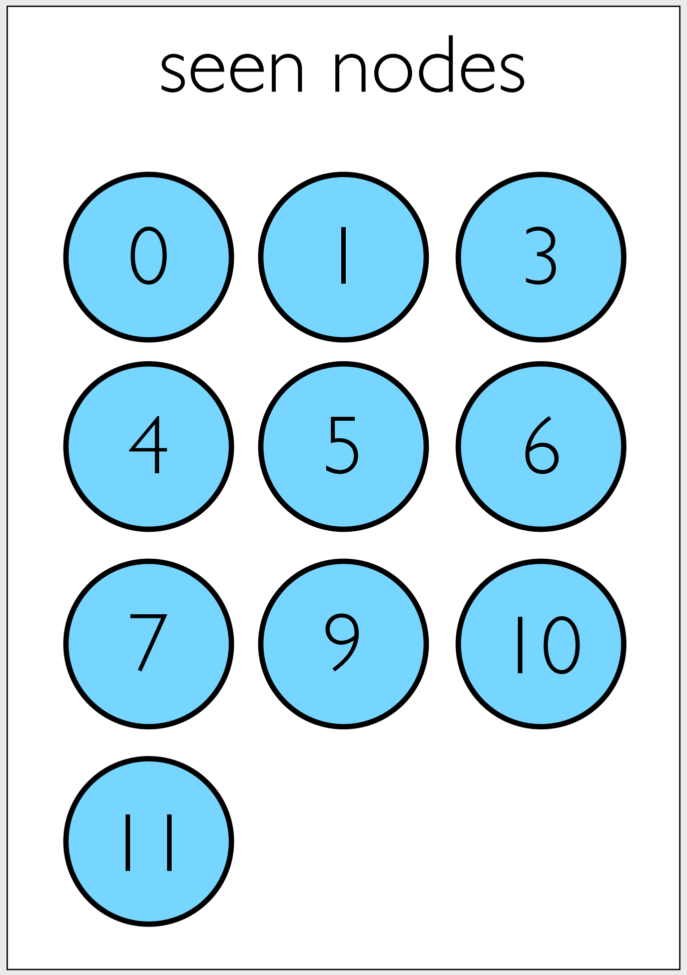


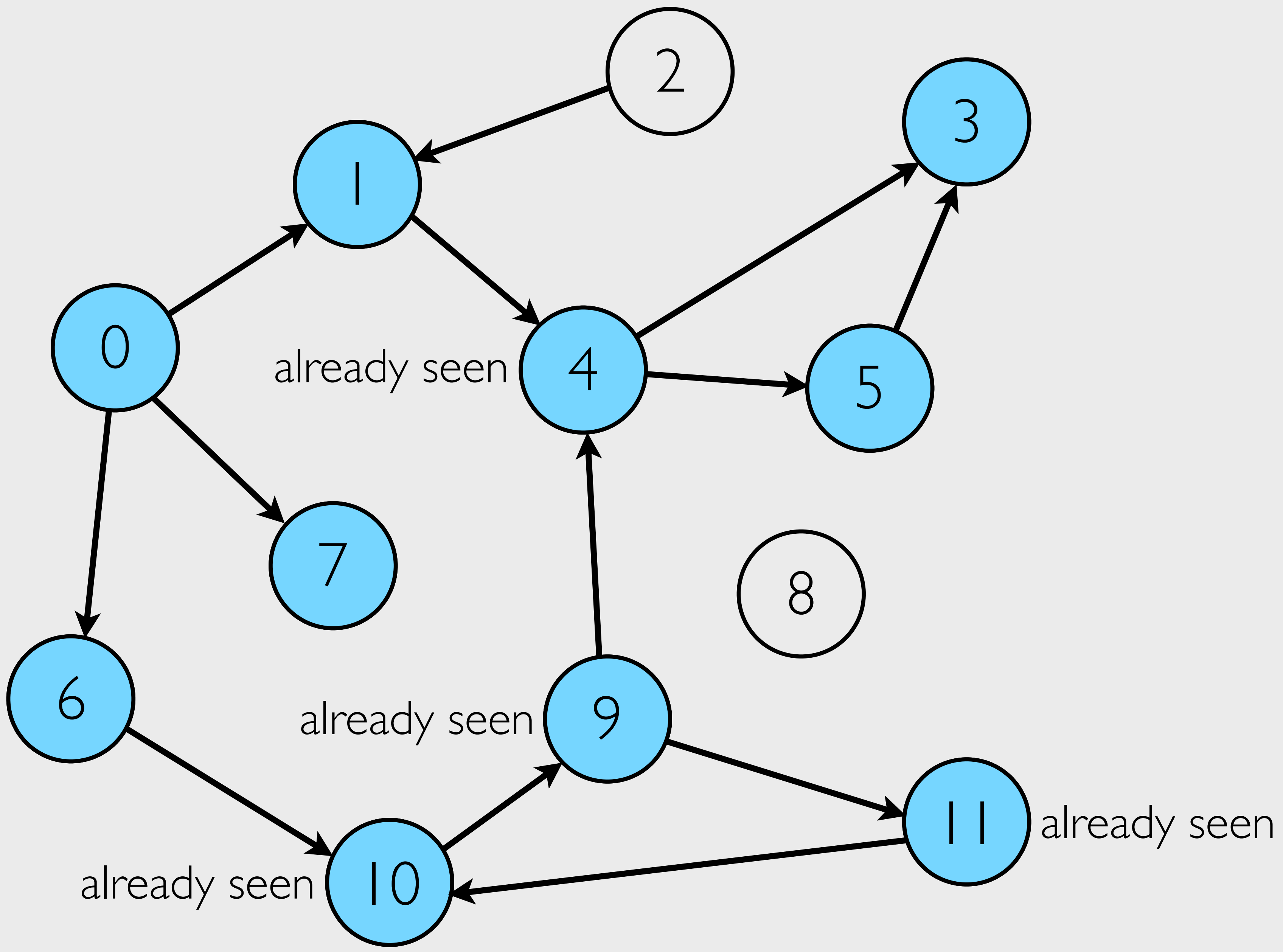
2
3
1
0
4
5
7
8
6
9
11
already seen
10



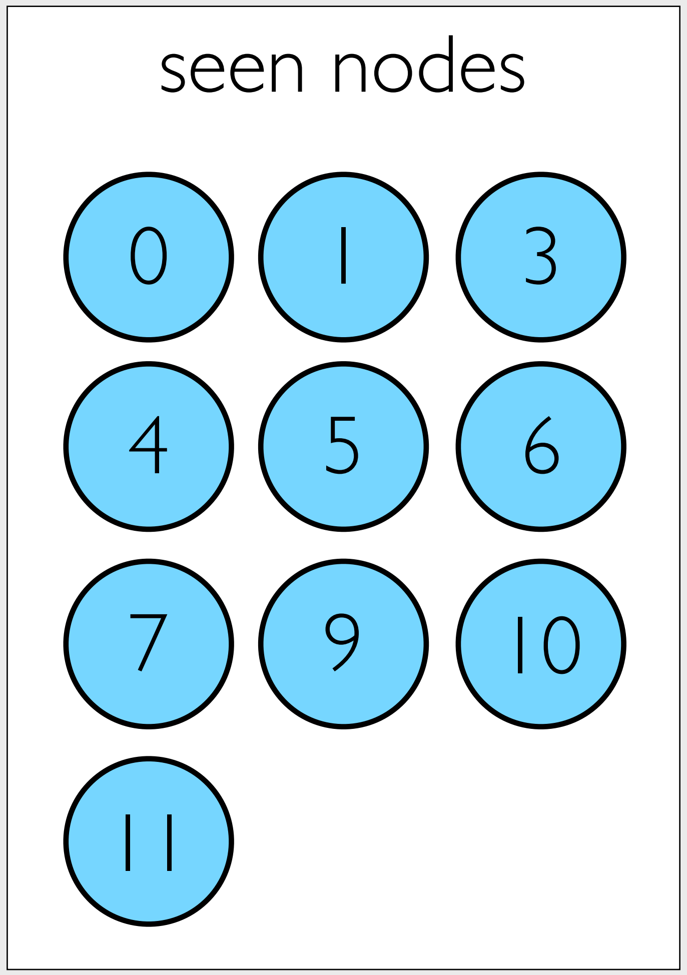
seen nodes
0
1
3
4
5
6
7
9
10
11


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

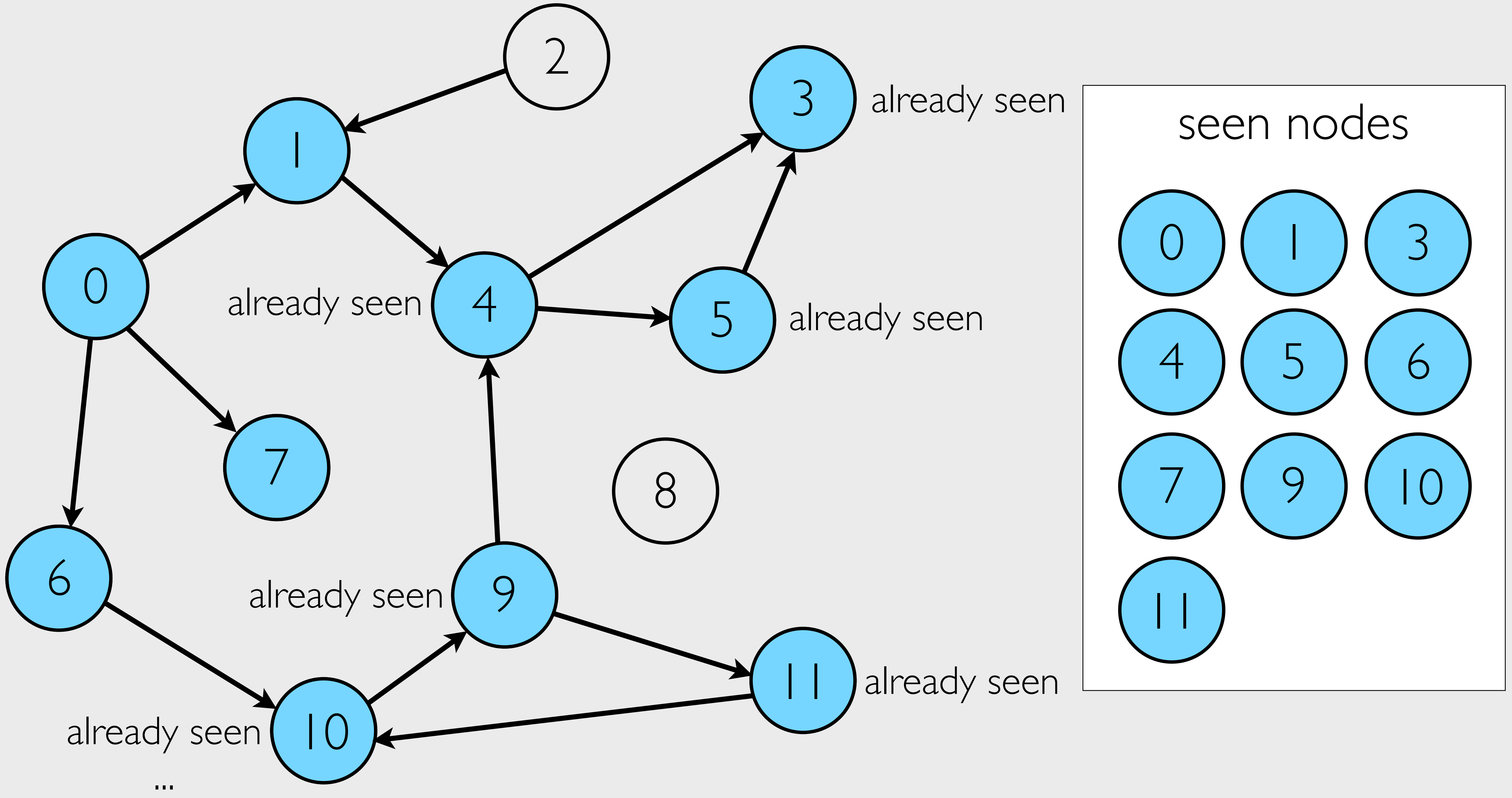

seen nodes
0
1
3
4
5
6
7
9
10
11

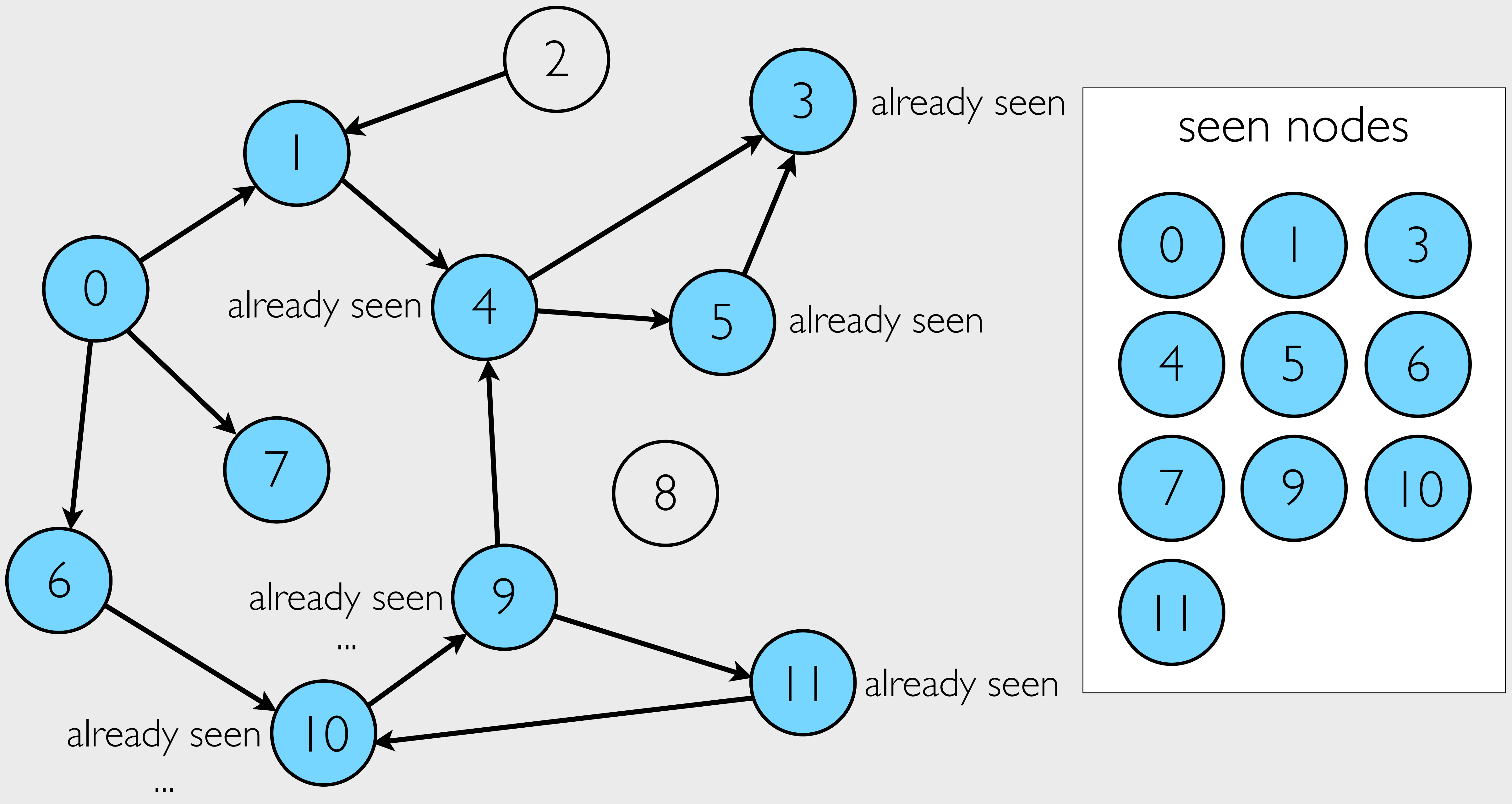


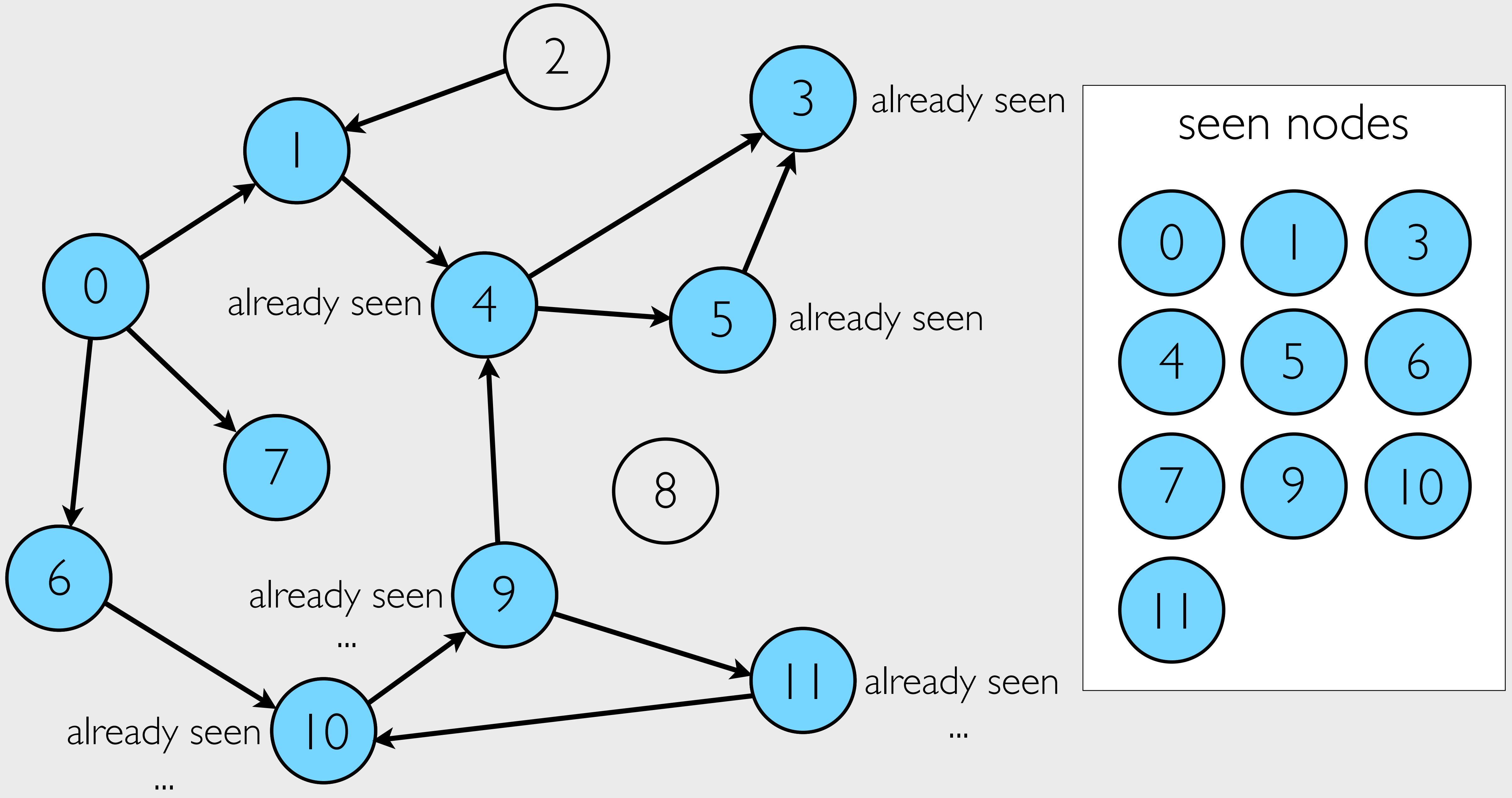
2
3
1
0
already seen
4
5
7
8
6
already seen
9
11
already seen
already seen
10



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

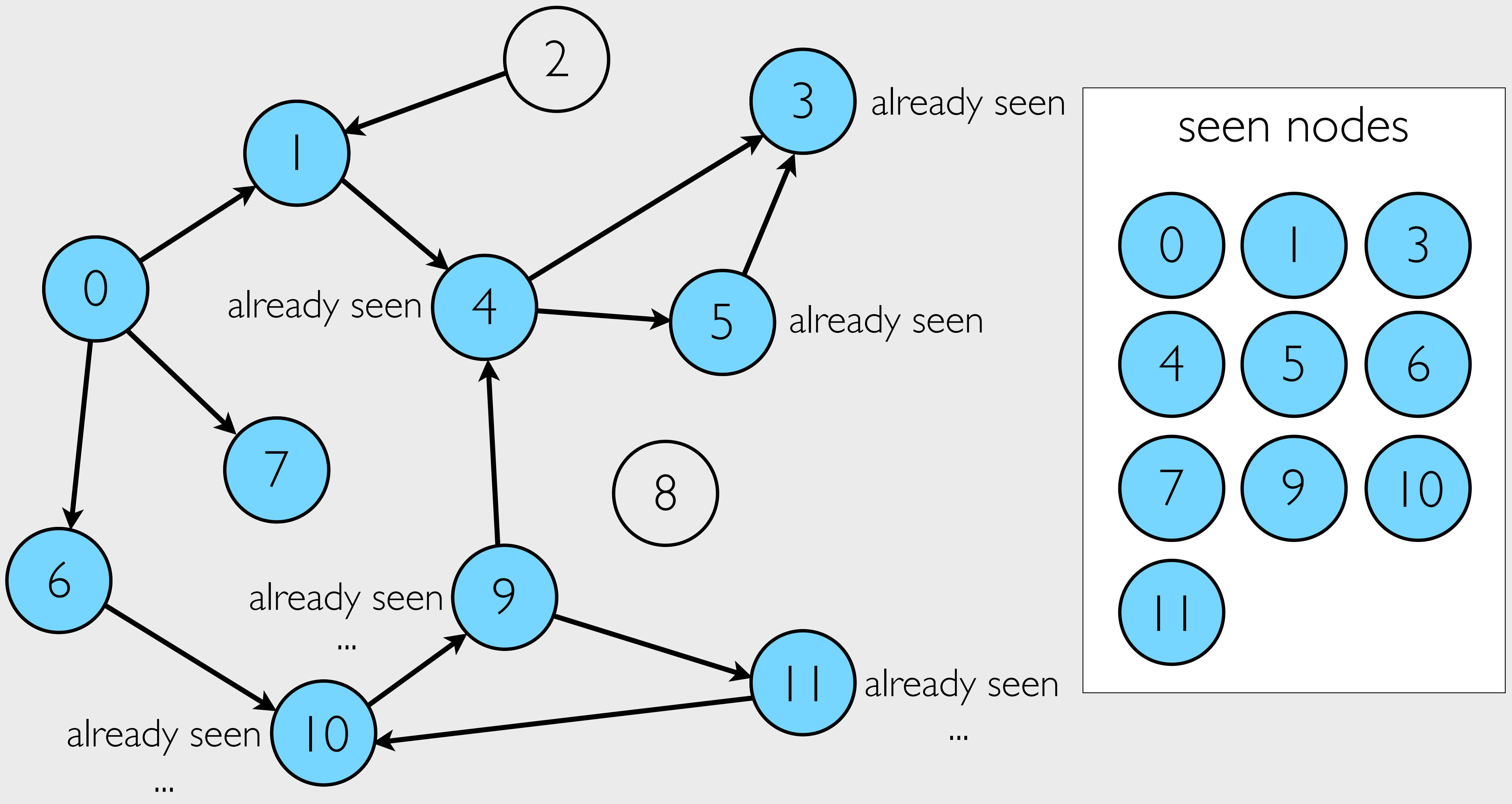

2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
9
11
already seen
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11


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



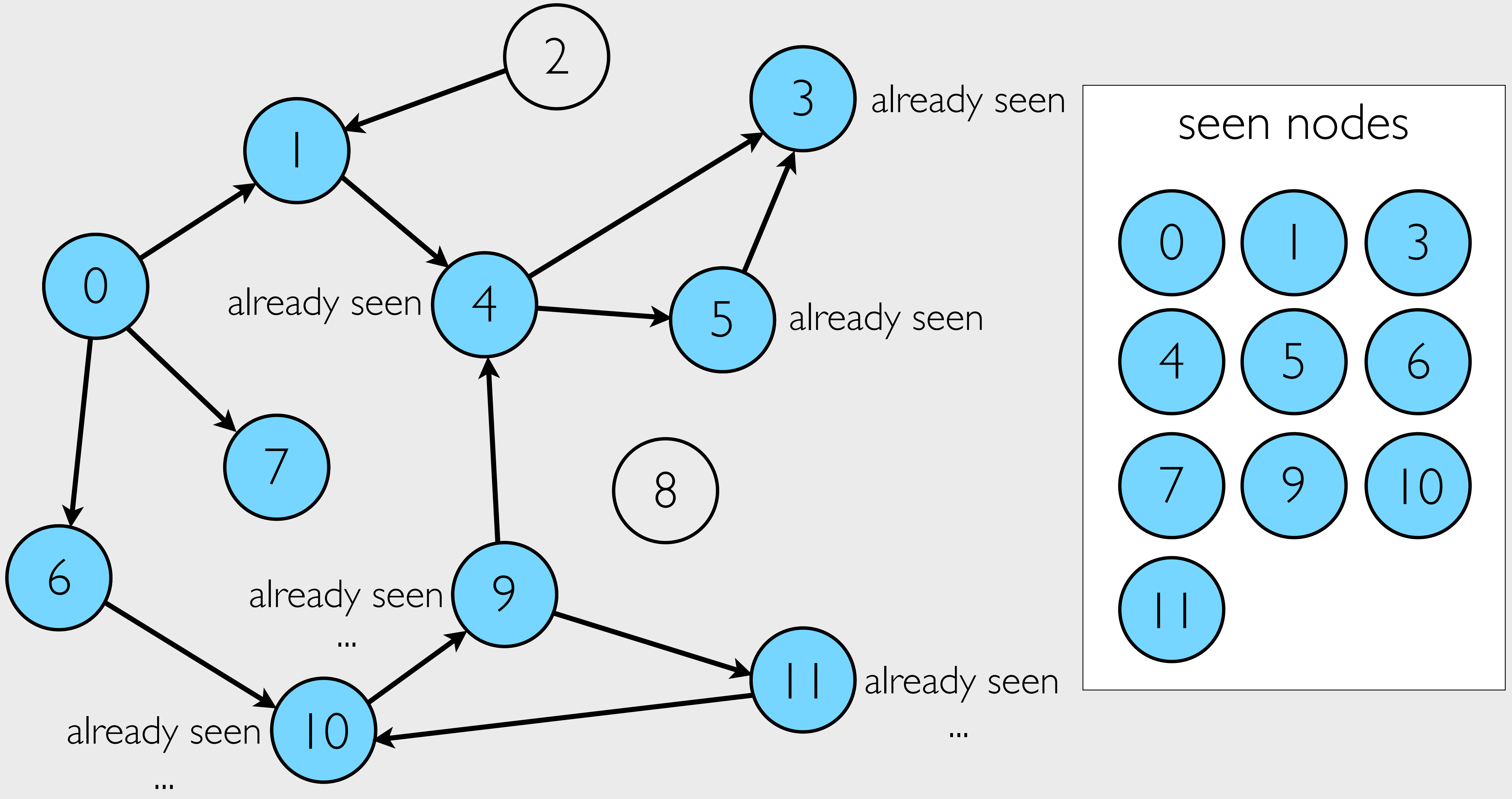
2
1
3
already seen
seen nodes
0
1
3
4
5
6
7
9
10
11
0
already seen
4
5
already seen
7
8
6
already seen
9
...
11
already seen
already seen
10
...
...

Events are updates that change an LVar's state



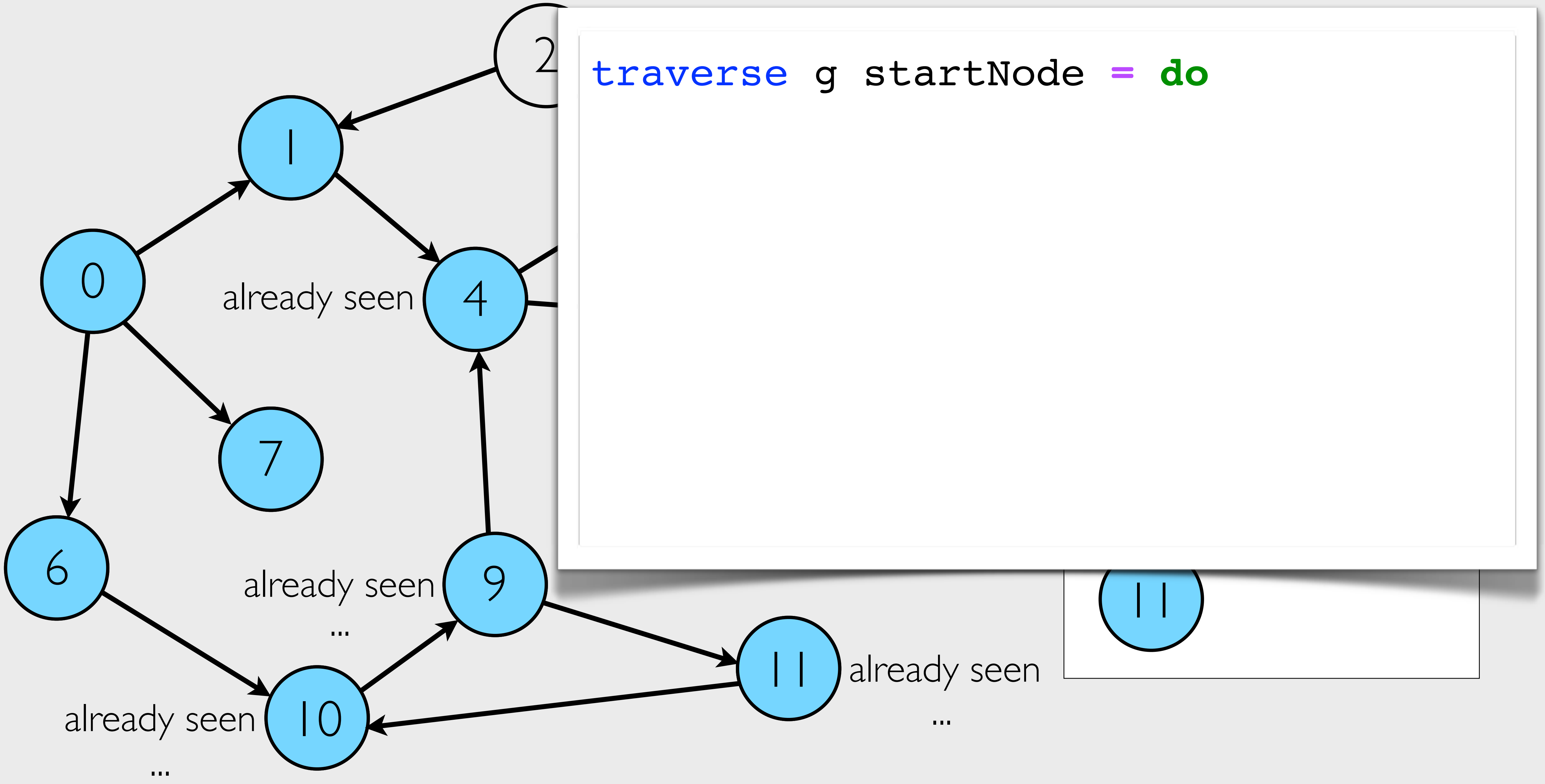
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response



Events are updates that change an LVar's state

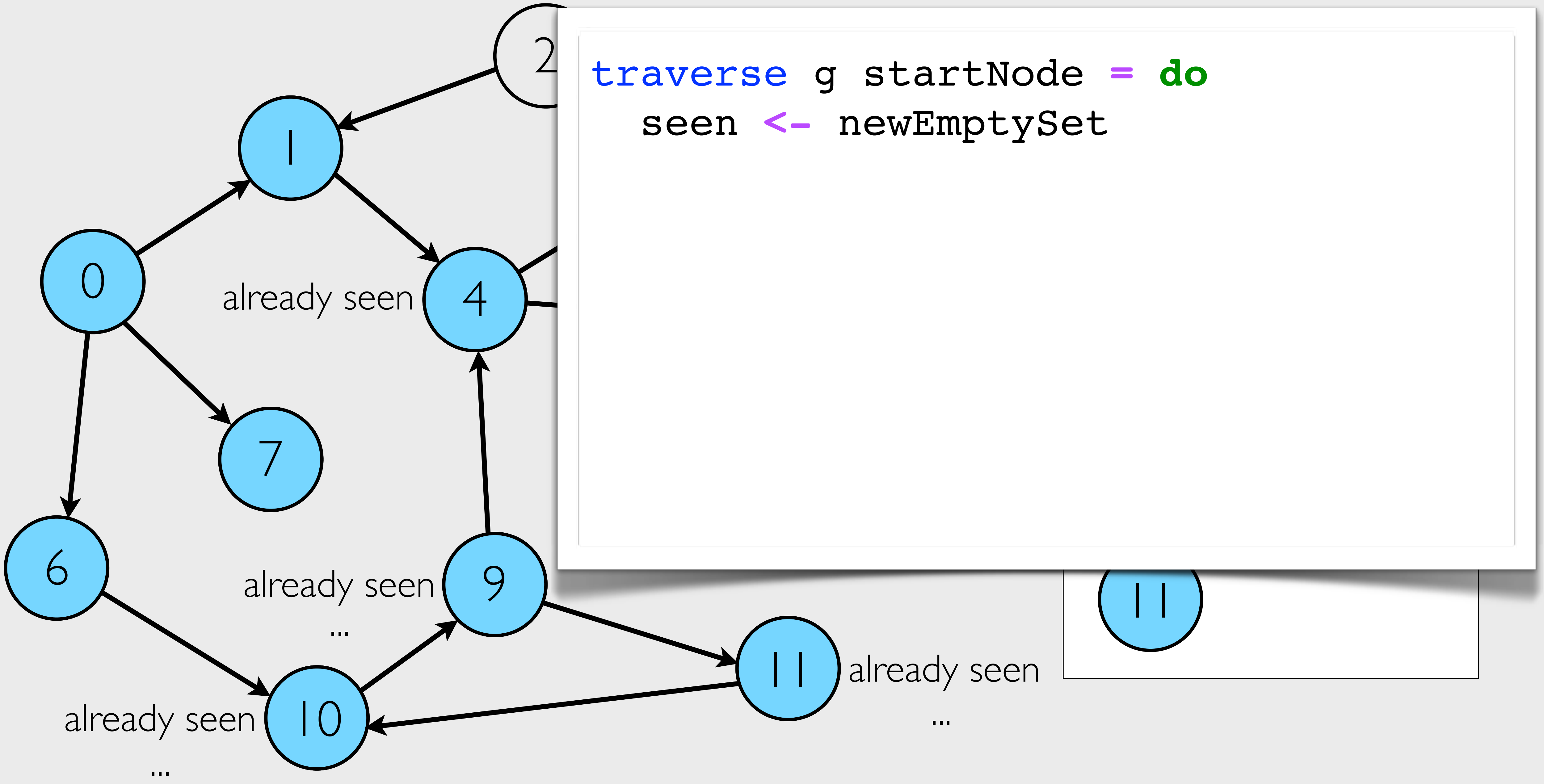
Event handlers listen for events and launch callbacks in response



```
traverse g startNode = do
```

Events are updates that change an LVar's state

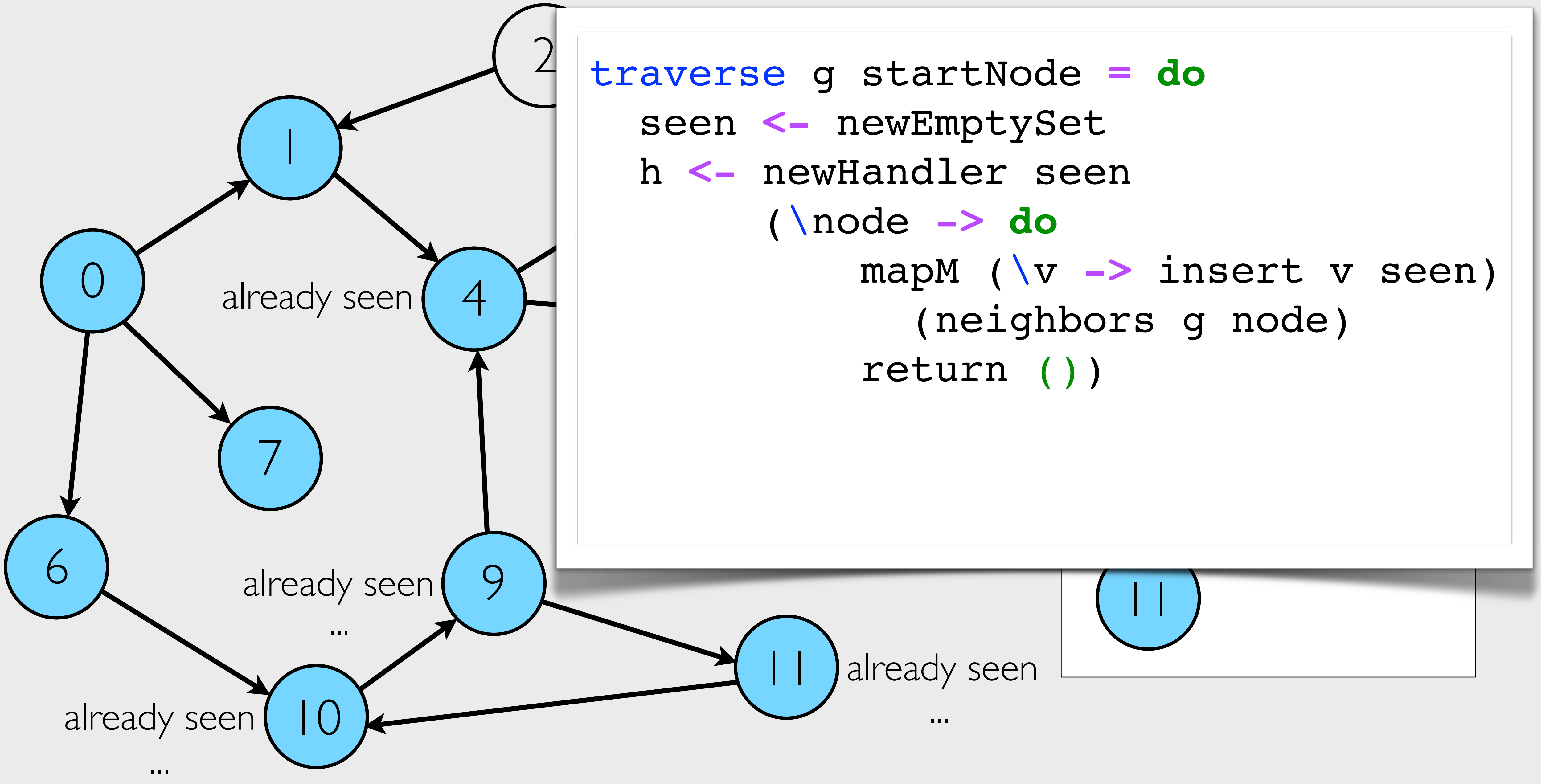
Event handlers listen for events and launch callbacks in response



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

Events are updates that change an LVar's state

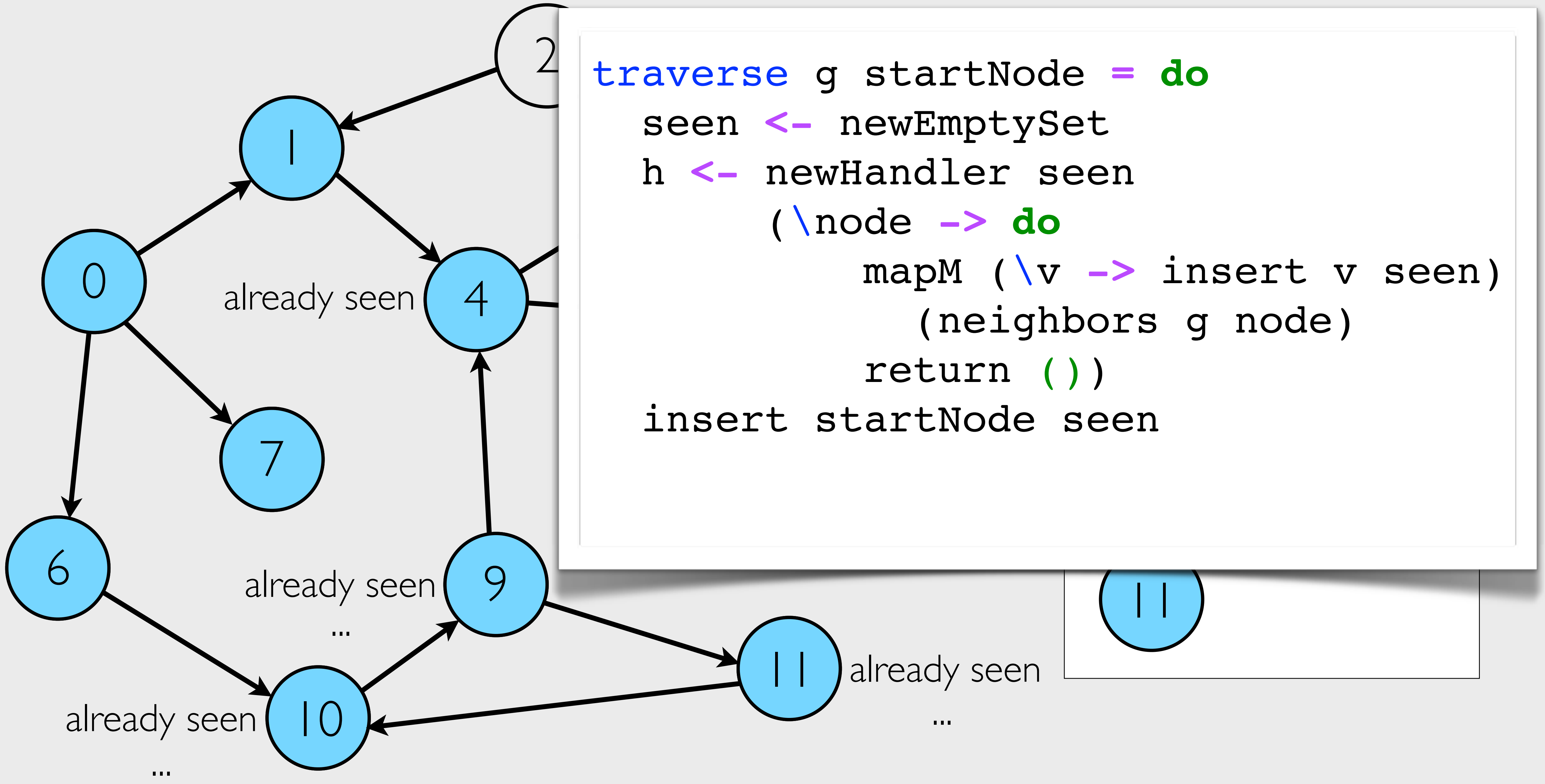
Event handlers listen for events and launch callbacks in response



```
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

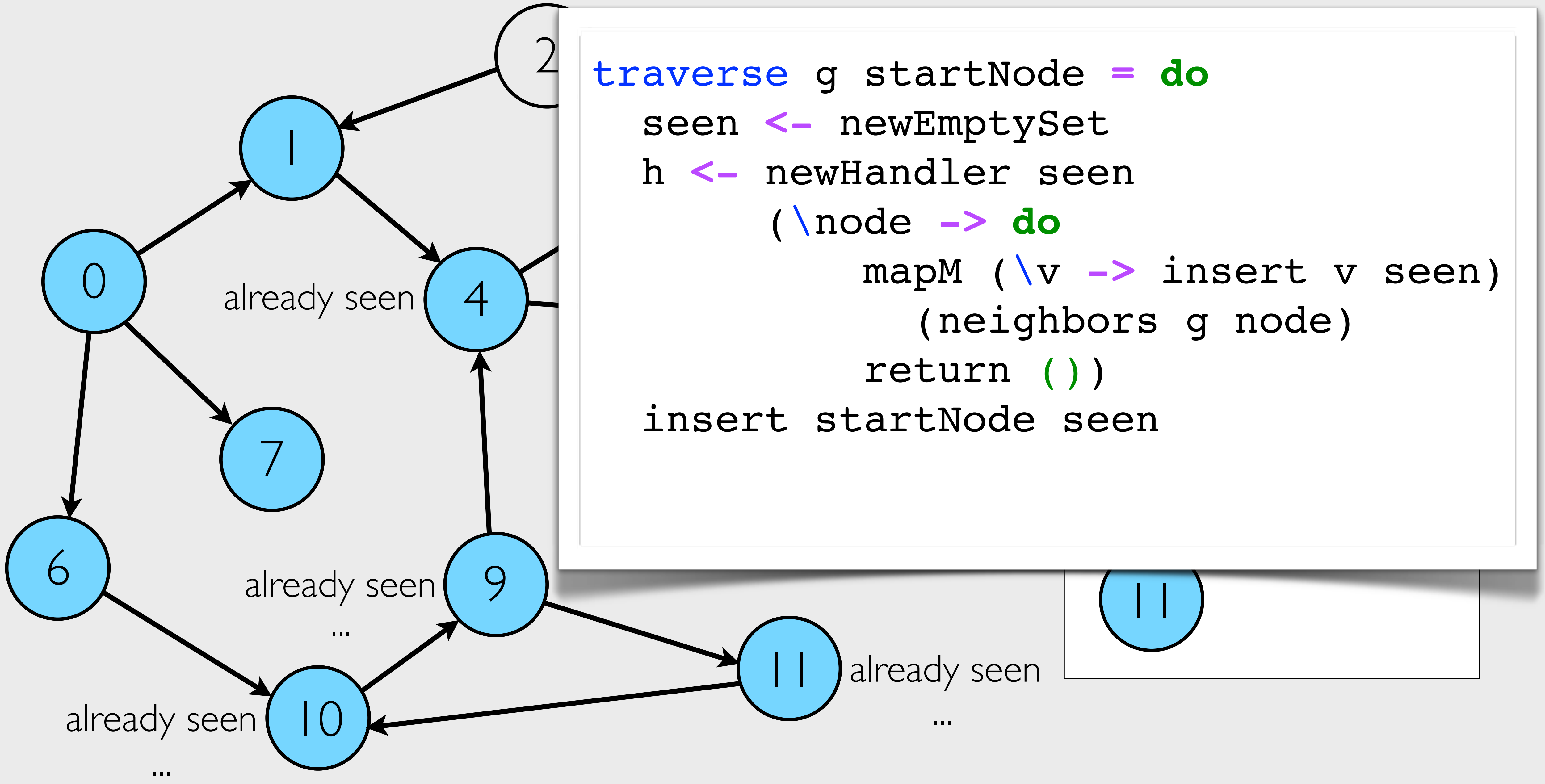


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
         (\node -> do
              mapM (\v -> insert v seen)
                (neighbors g node)
              return ())
  insert startNode seen
```

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

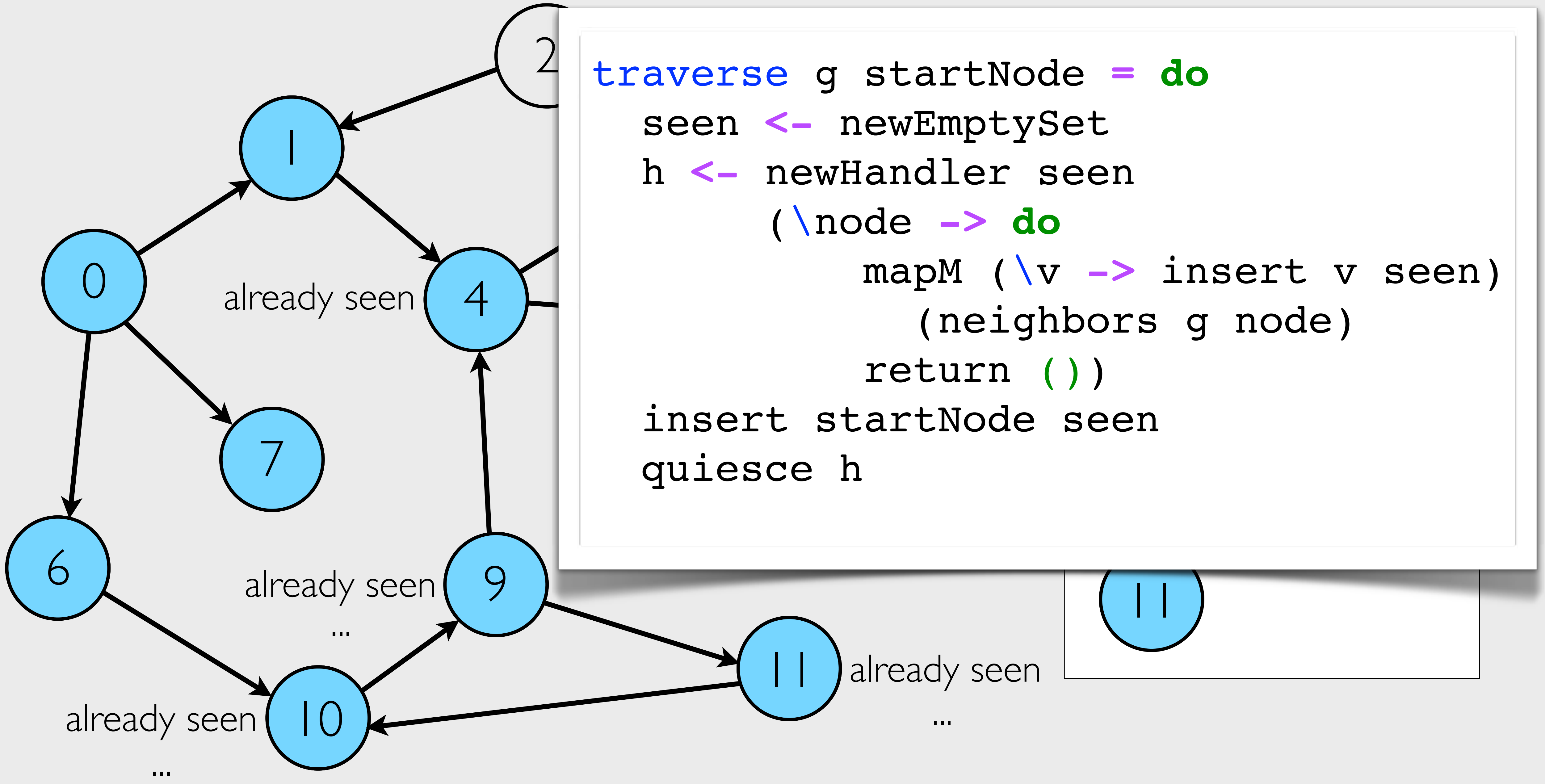


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
         (\node -> do
              mapM (\v -> insert v seen)
                (neighbors g node)
              return ())
  insert startNode seen
```

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

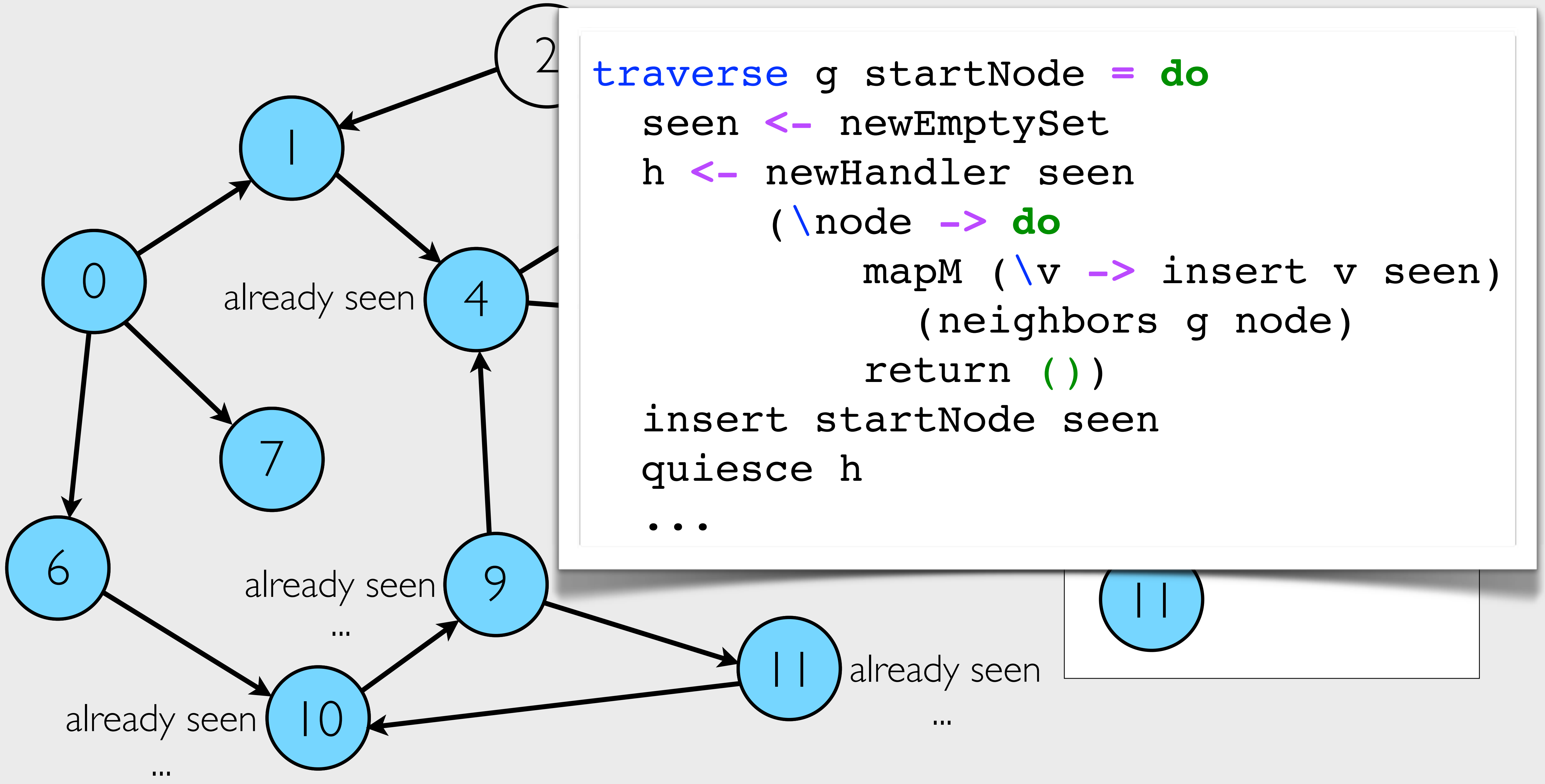


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
```

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
  ...
```

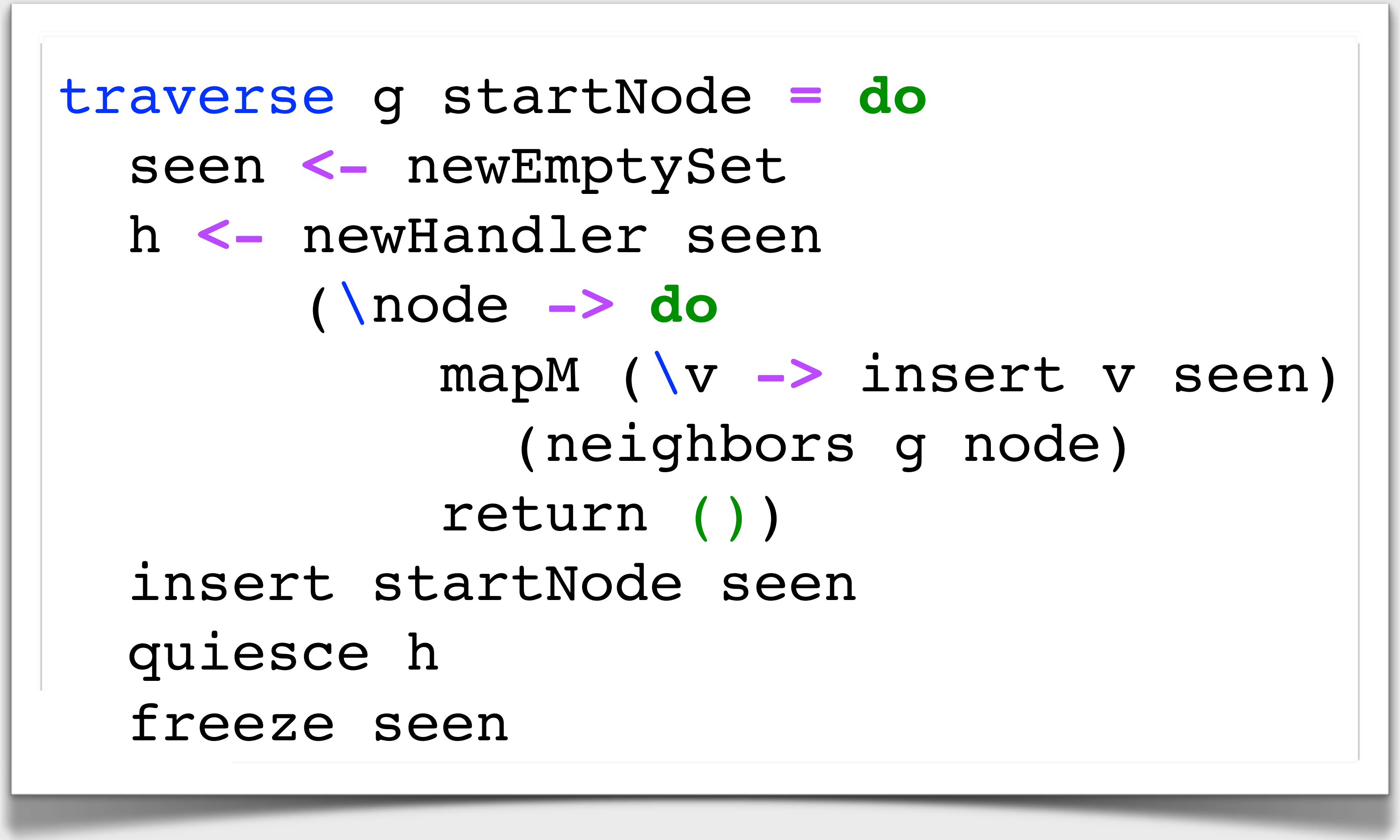
`freeze`: exact non-blocking read

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

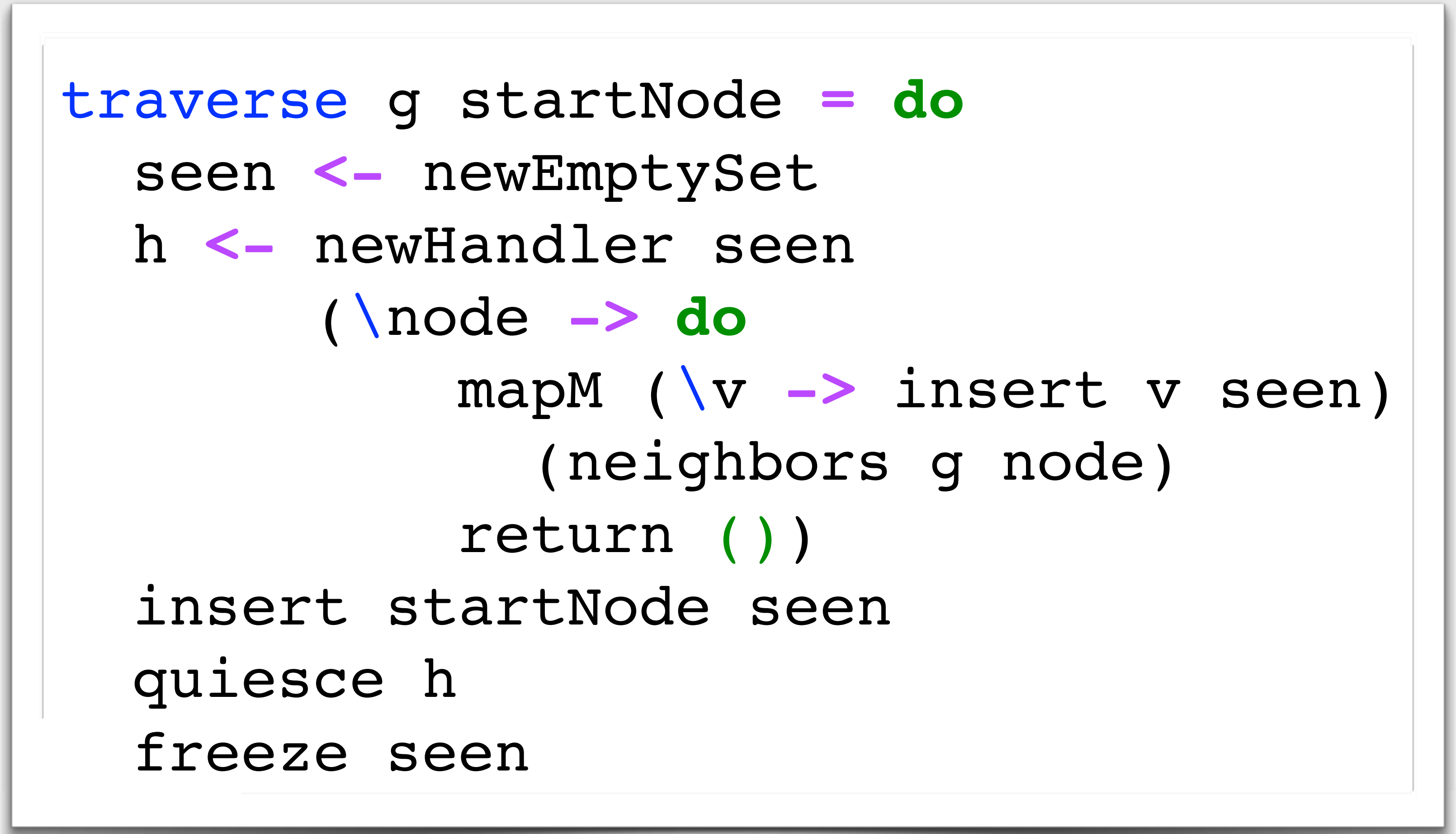


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`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

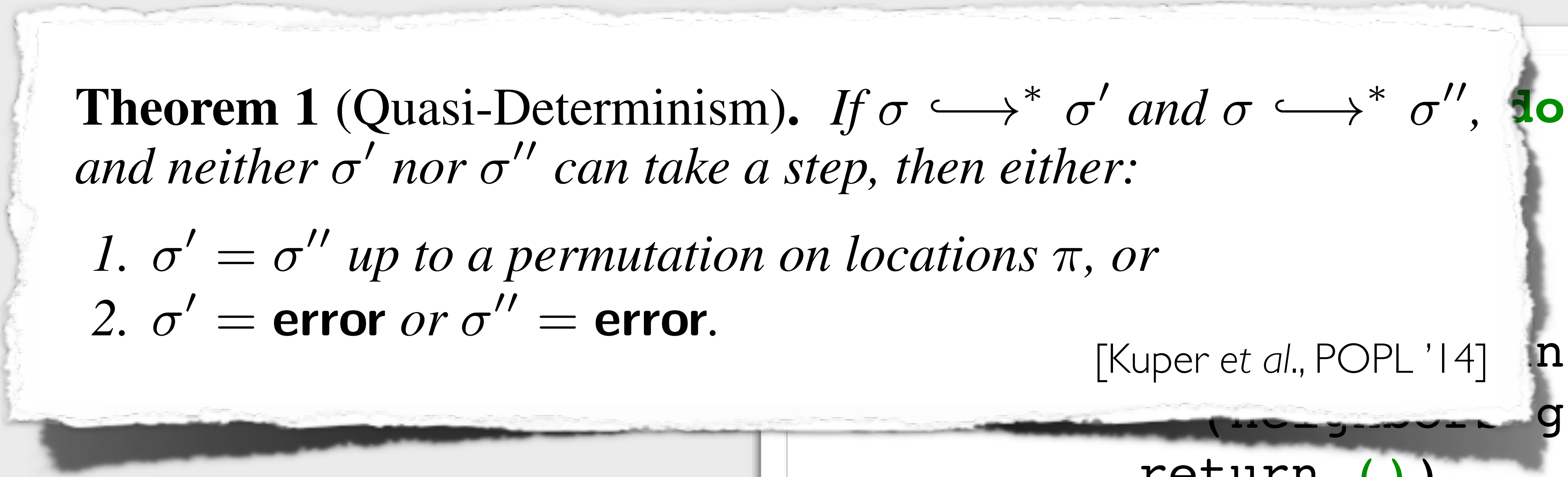


```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

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

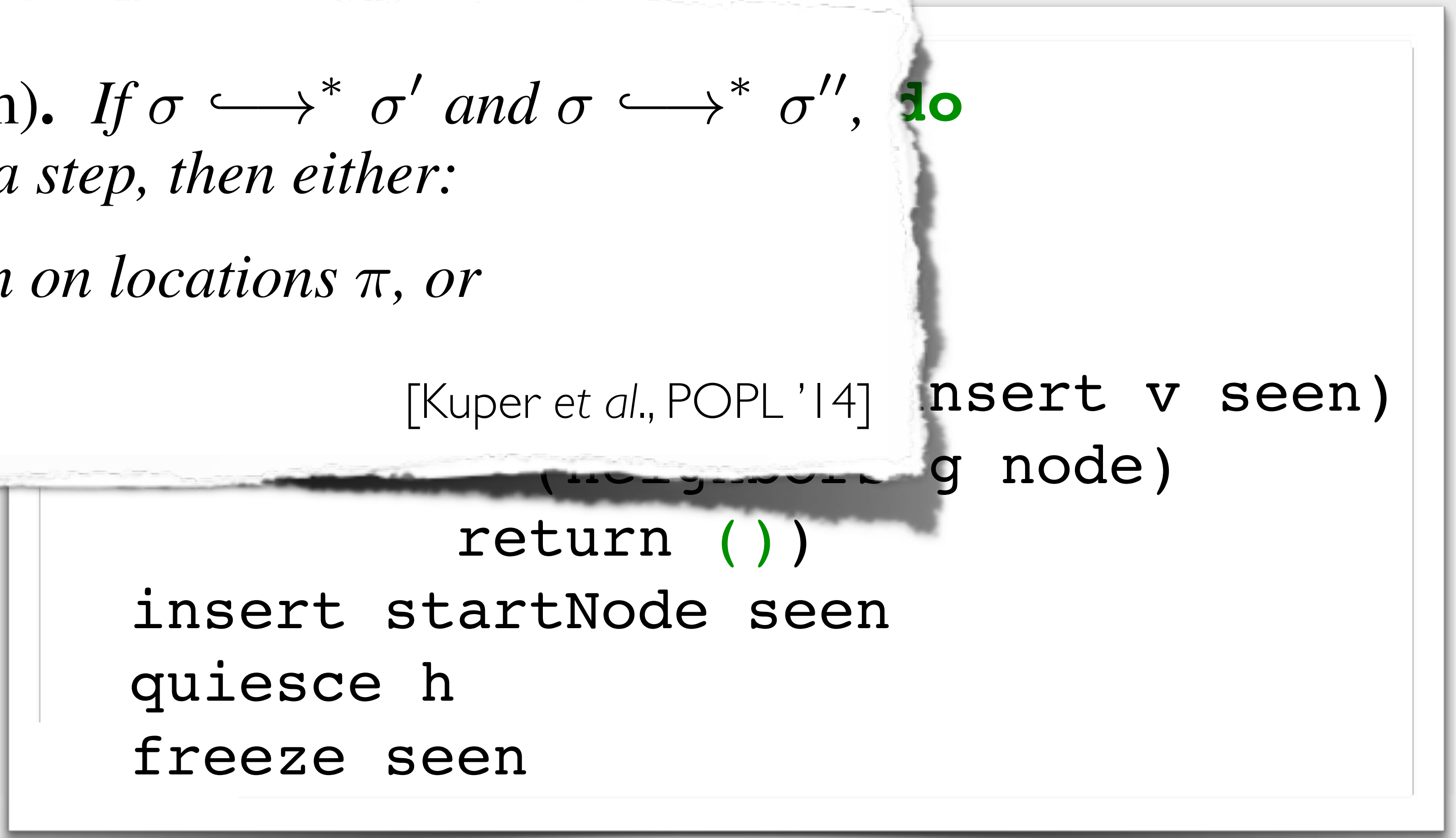
Two possible outcomes: either the same final value or an exception



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**.*

[Kuper et *al.*, POPL '14]

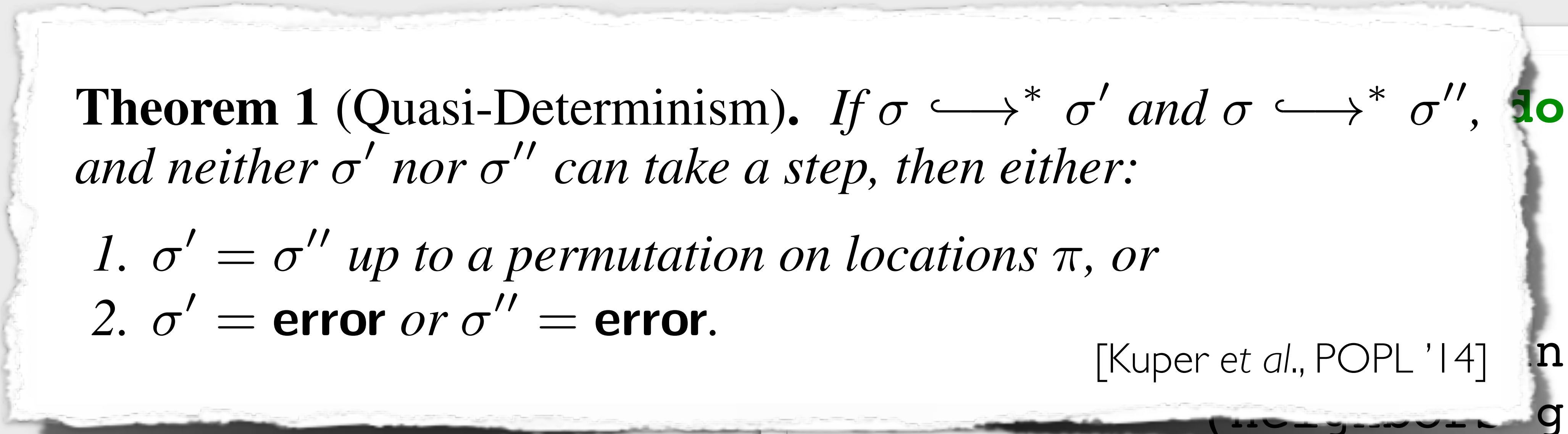


```
                                      do



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

`freeze`: exact non-blocking read

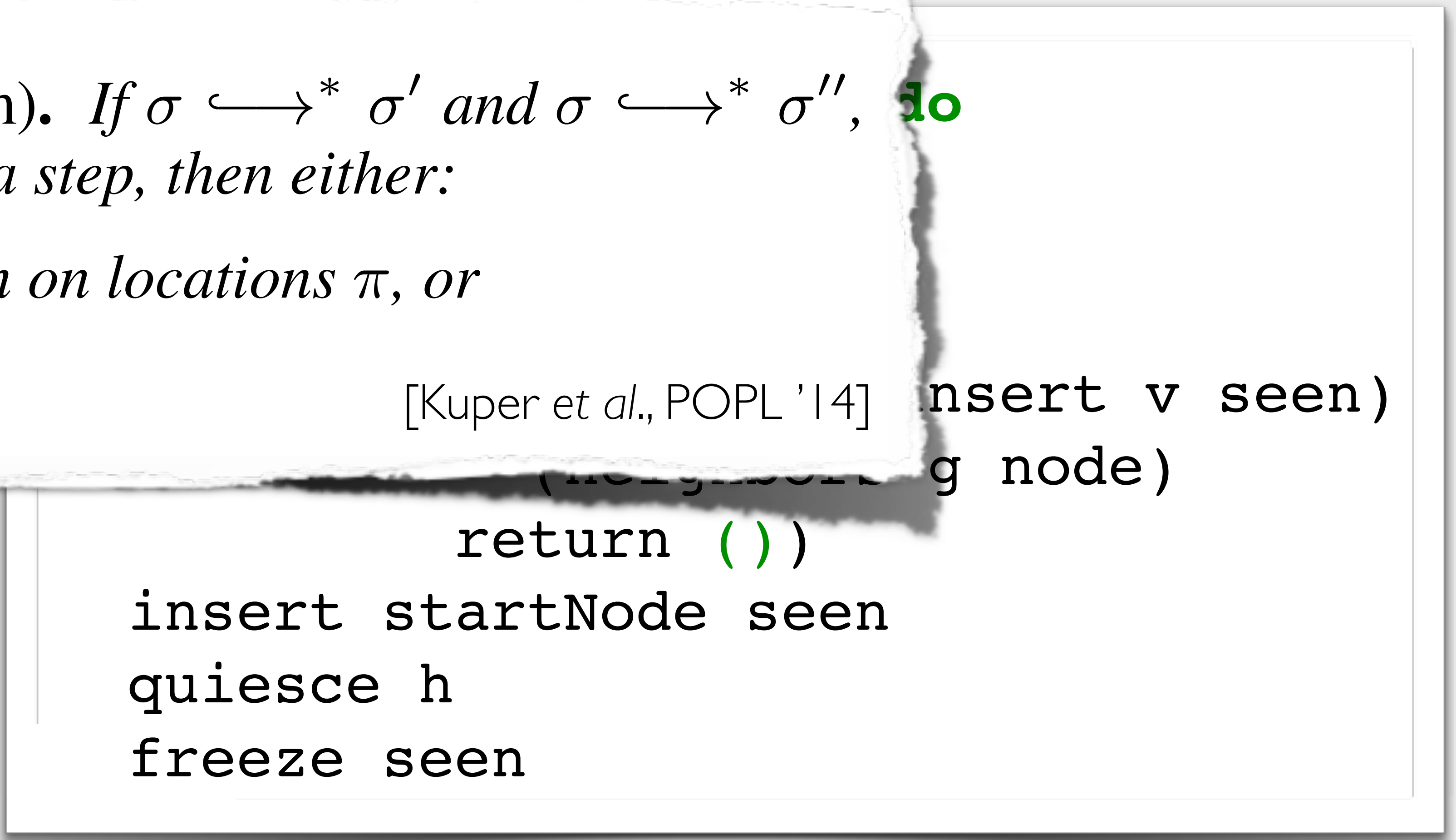
Attempts to write to a frozen LVar raise a write-after-`freeze` exception

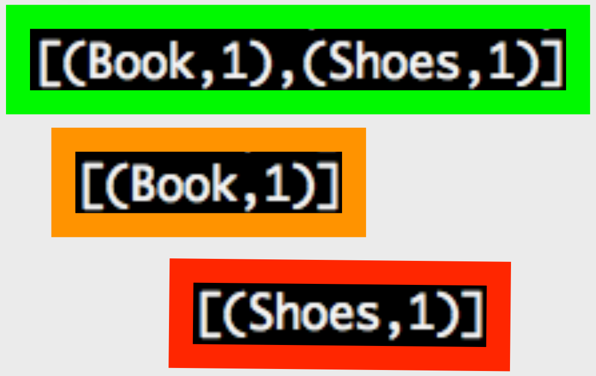
Two possible outcomes: either the same final value or an exception


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**.

[Kuper et al., POPL '14]

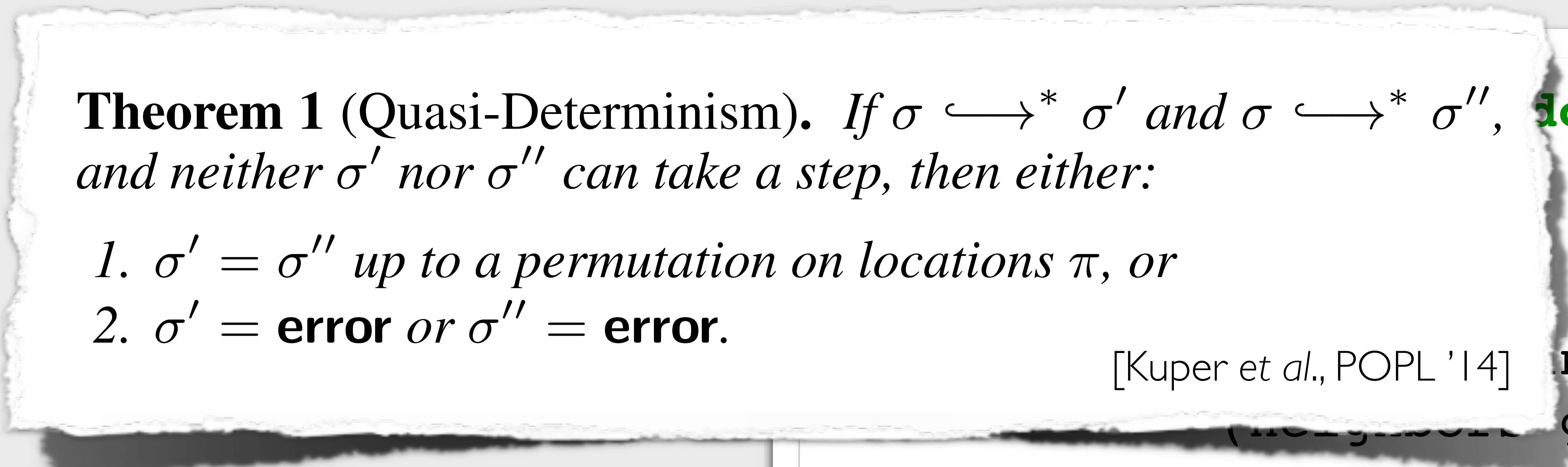

```
                                                          do
                                             nsert v seen)
                                             g node)
          return ())
insert startNode seen
quiesce h
freeze seen
```


```
[(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

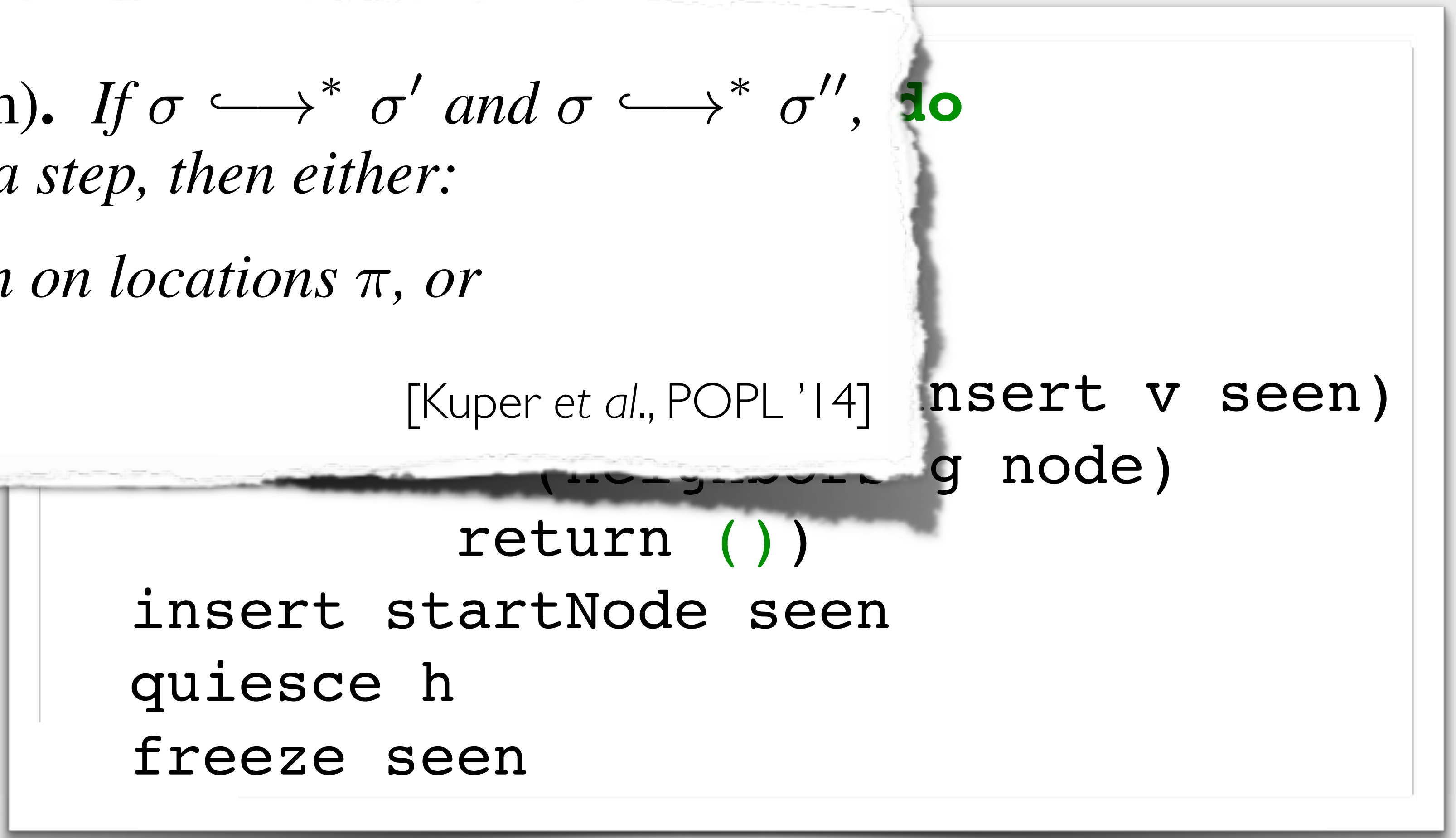
Two possible outcomes: either the same final value or an exception



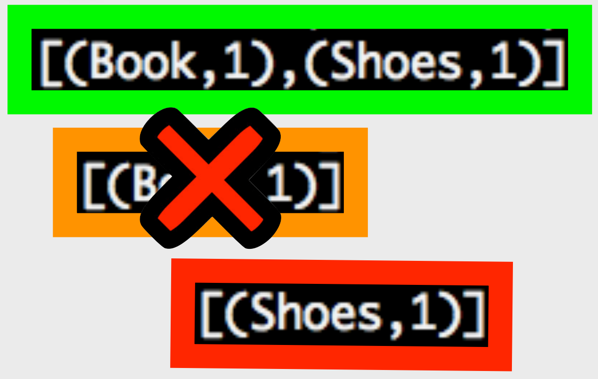
Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

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

[Kuper *et al.*, POPL '14]



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

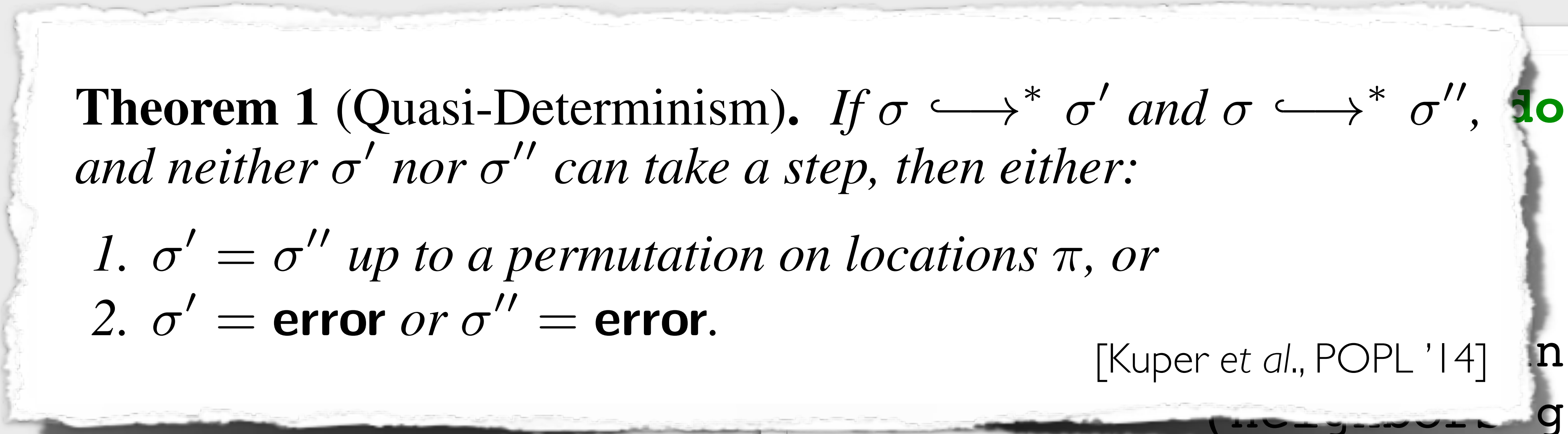


```
[(Book,1),(Shoes,1)]
[(B    1)]
[(Shoes,1)]
```

`freeze`: exact non-blocking read

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

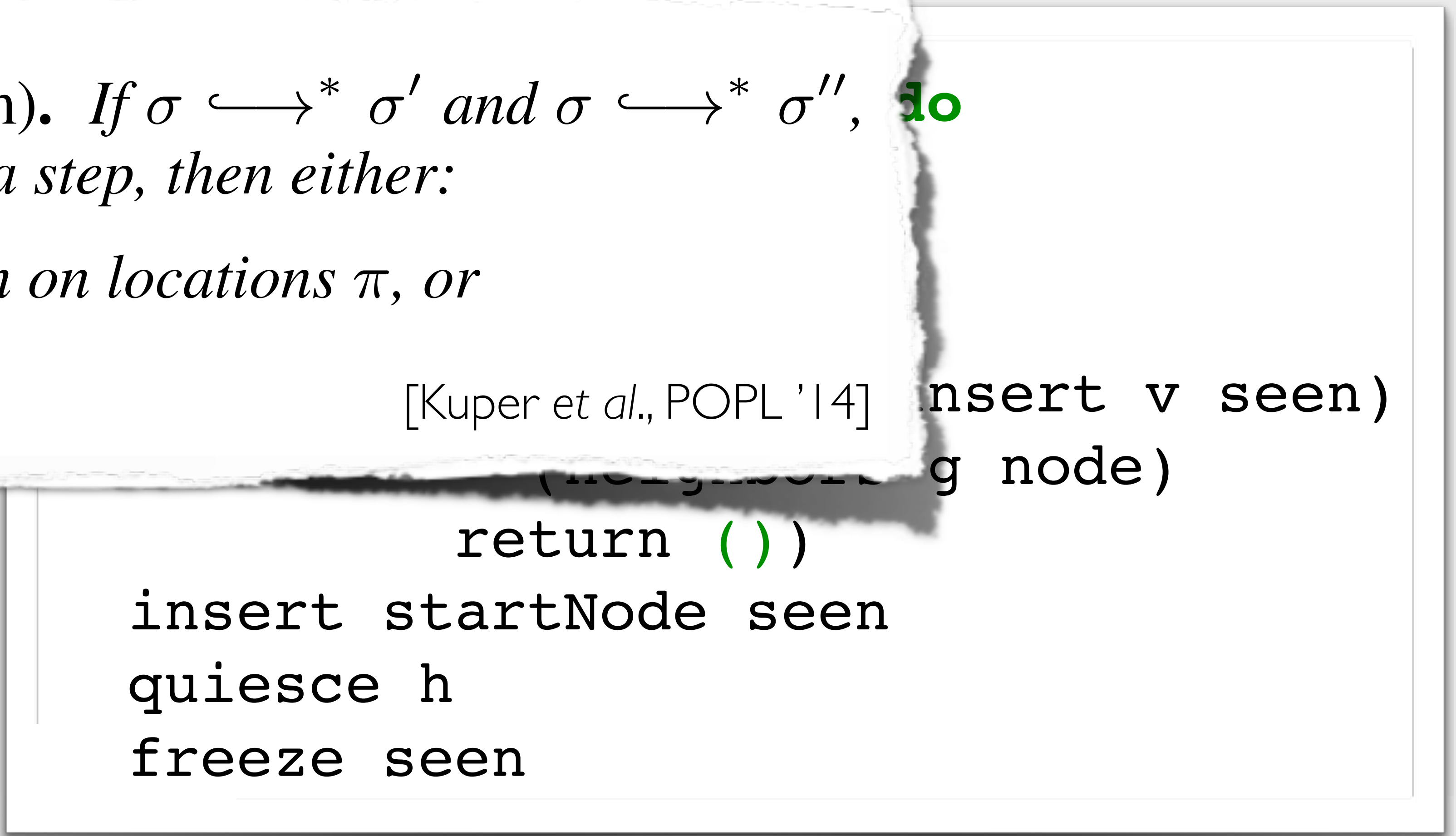
Two possible outcomes: either the same final value or an exception



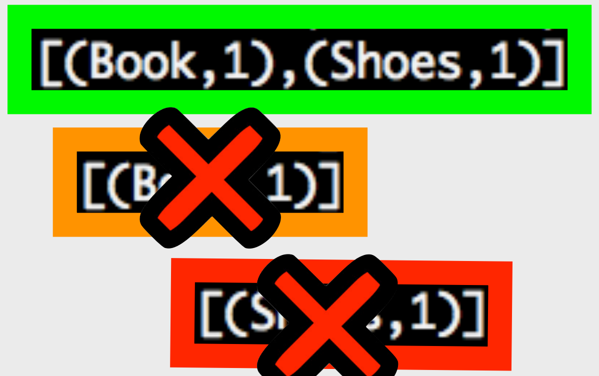
Theorem 1 (Quasi-Determinism). *If* $\sigma \hookrightarrow^* \sigma'$ *and* $\sigma \hookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

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

[Kuper et *al.*, POPL '14]



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

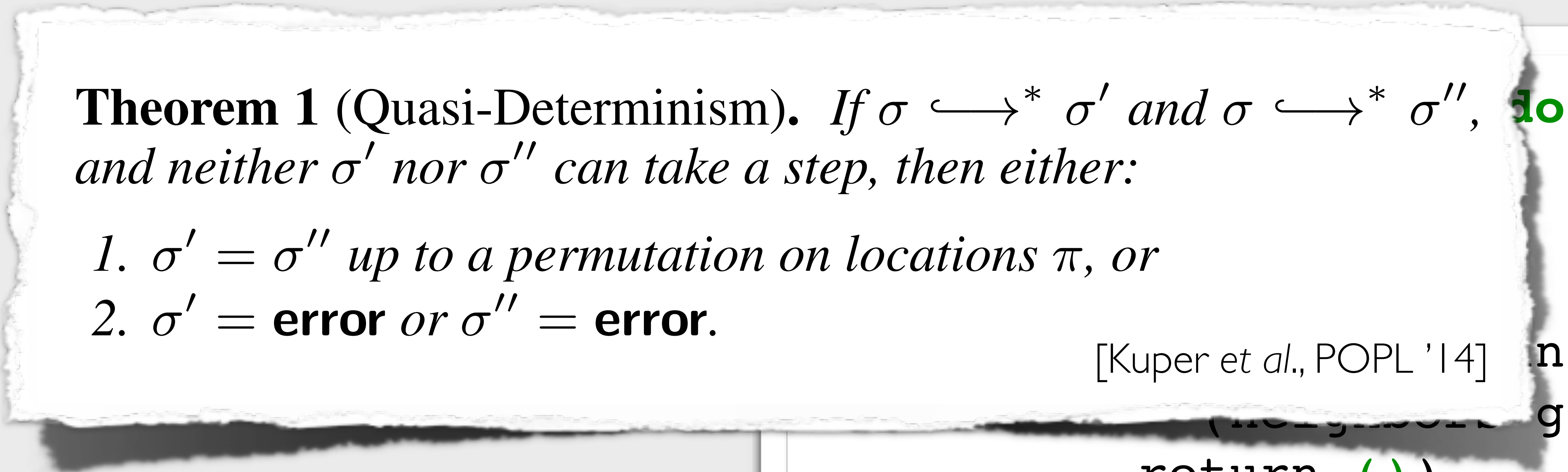


```
[(Book,1),(Shoes,1)]
[(B   1)]
[(S   ,1)]
```

`freeze`: exact non-blocking read

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

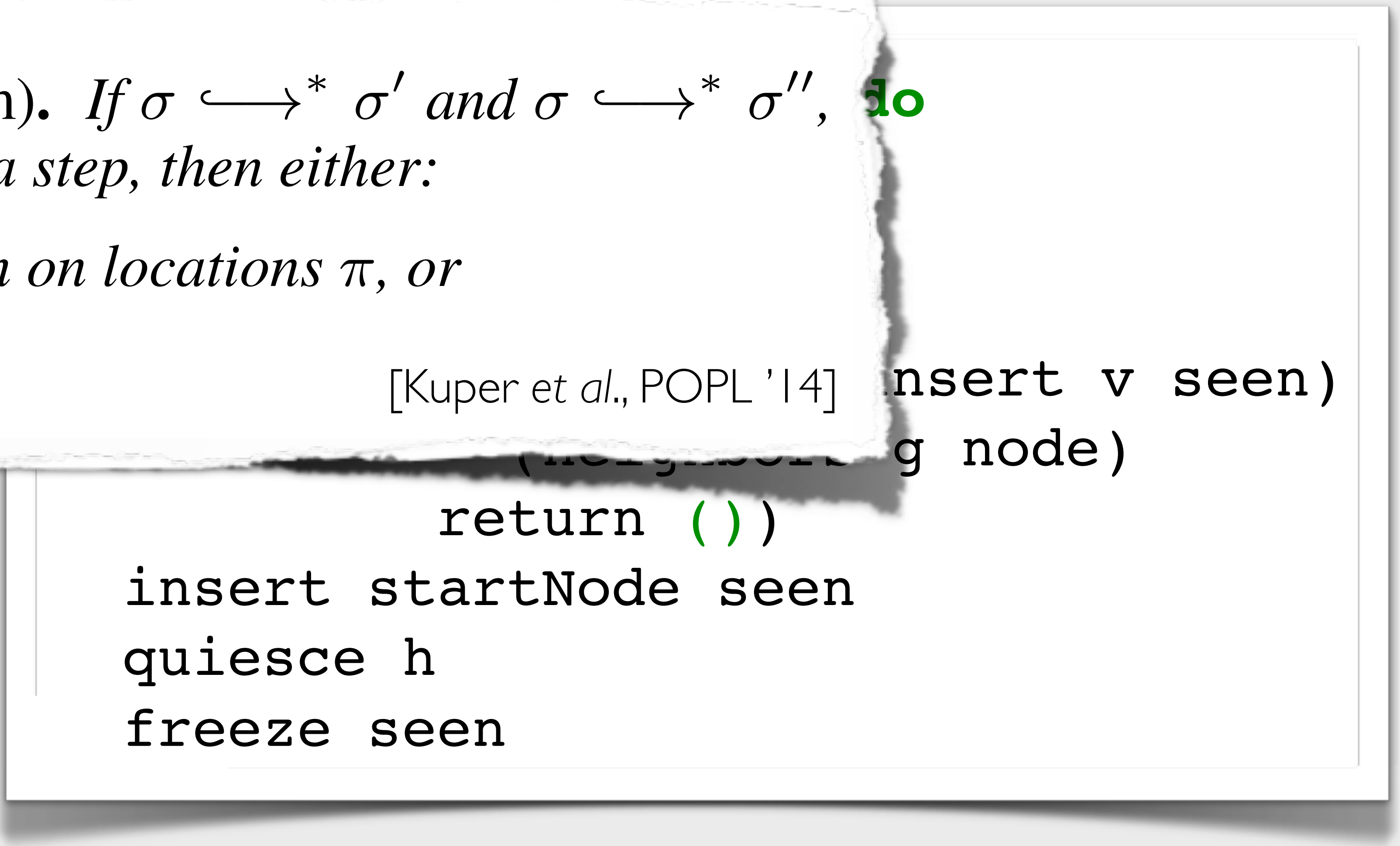
Two possible outcomes: either the same final value or an exception



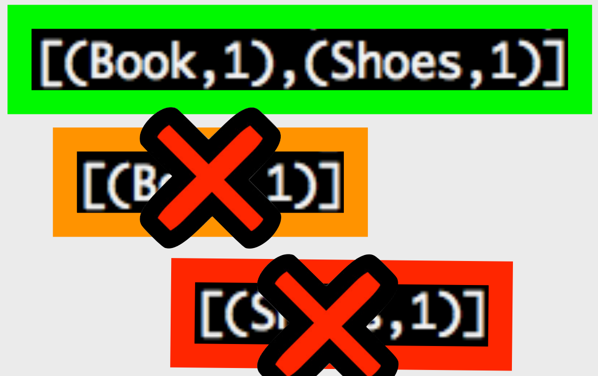
Theorem 1 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

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

[Kuper *et al.*, POPL '14]



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



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

or error.

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

Quasi-Determinism

$$\begin{tikzcd}
 & \sigma \arrow[dl, "*"] \arrow[dr, "*"] & \\
\sigma' & = & \sigma''
\end{tikzcd}$$

Quasi-
Determinism

$$\begin{tikzcd}[column sep=small] & \sigma \arrow[dl, "*"'] \arrow[dr, "*"] & \\ \sigma' & = & \sigma'' \end{tikzcd}$$

Strong
Local Quasi-
Confluence

$$\begin{tikzcd} & \sigma \arrow[dl] \arrow[dr] & \\ \sigma_a \arrow[dr, "\leq 1"'] & & \sigma_b \arrow[dl, "\leq 1"] \\ & \sigma_c & \end{tikzcd}$$

Independence

$$\frac{\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle}{\langle S \sqcup_S S'';\, e\rangle \hookrightarrow \langle S' \sqcup_S S'';\, e'\rangle}$$

Strong Local Quasi-Confluence

$$\begin{tikzcd}
& \sigma \arrow[dl] \arrow[dr] & \\
\sigma_a \arrow[dr, "\leq 1"'] & & \sigma_b \arrow[dl, "\leq 1"] \\
& \sigma_c &
\end{tikzcd}$$

Quasi-Determinism

$$\begin{tikzcd}
& \sigma \arrow[dl, "*"] \arrow[dr, "*"] & \\
\sigma' & = & \sigma''
\end{tikzcd}$$

Frame rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

[O'Hearn *et al.*, 2001]

Independence

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle}$$

Frame rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

[O'Hearn *et al.*, 2001]

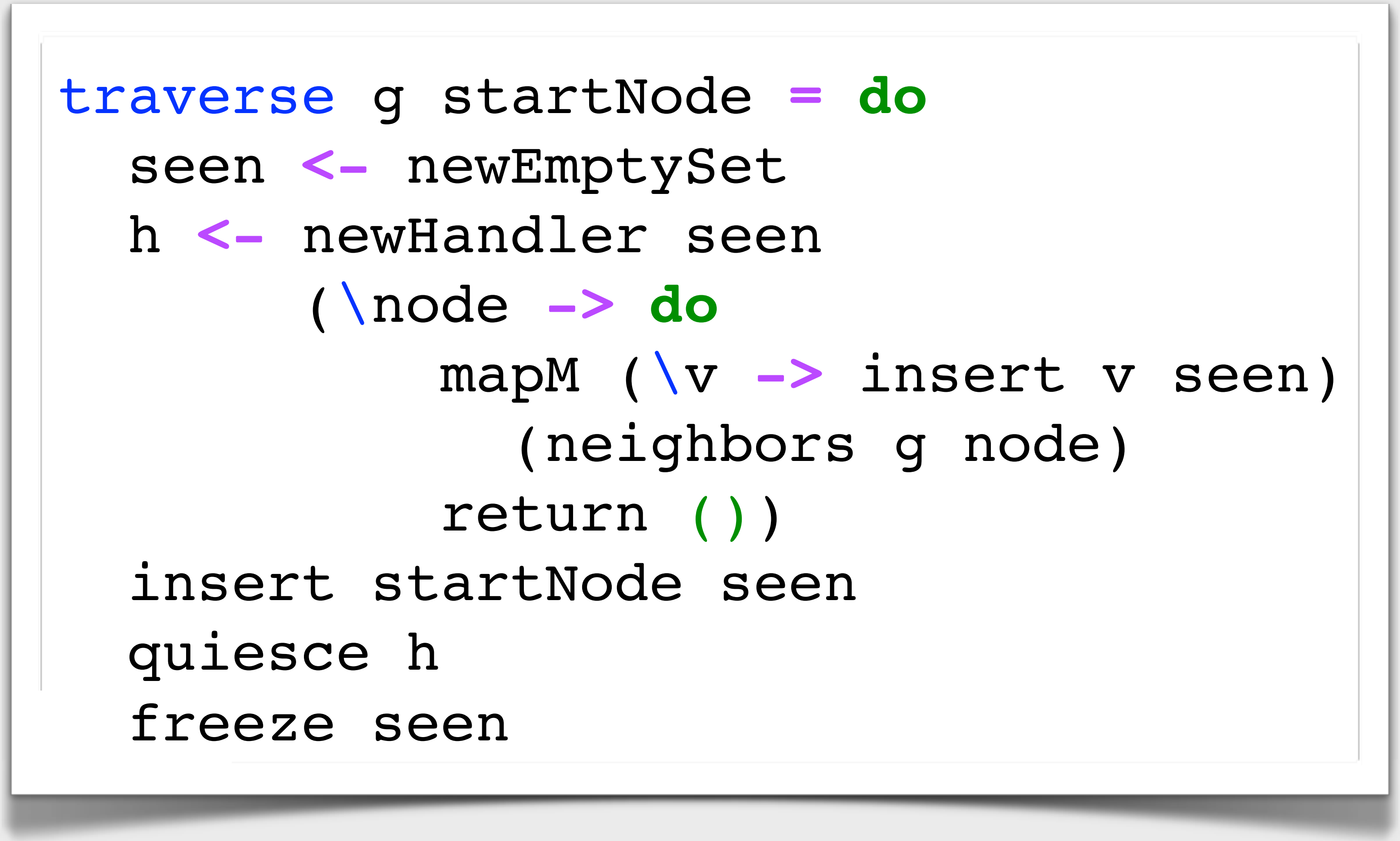
Independence

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle}$$

`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

```
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:

```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

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

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

main = do
  putStr (show (toList (fromIMap
    (runParThenFreeze p))))
```

LVish

a Haskell library for parallel programming with LVars

LVar operations run in **`Par`** computations

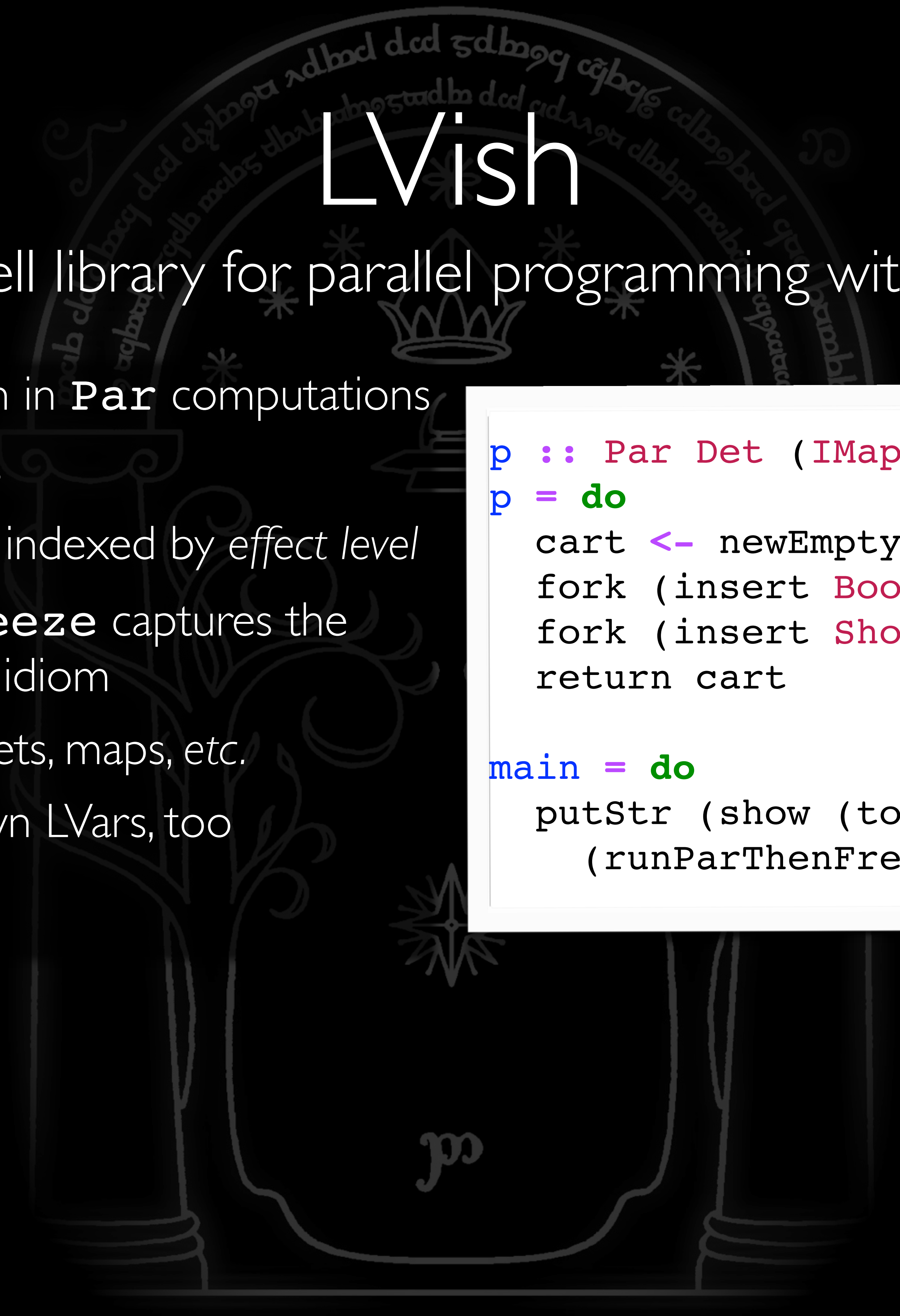
Lightweight threads

`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 = do
  putStr (show (toList (fromIMap
    (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 = do
  putStr (show (toList (fromIMap
    (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*

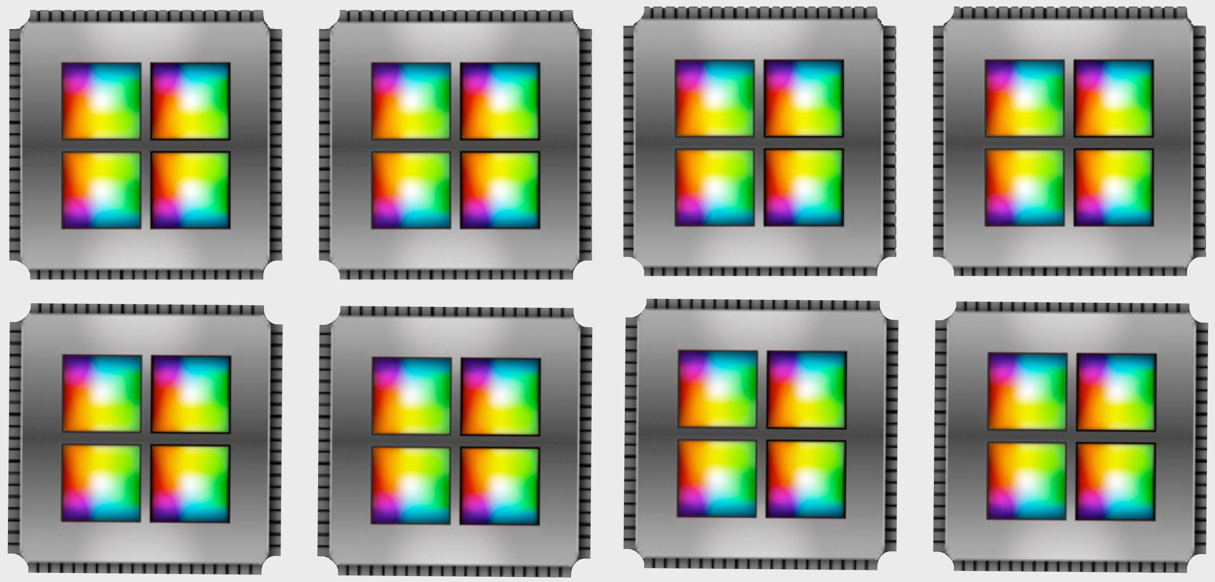
`runParThenFreeze` captures the freeze-after-writing idiom

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

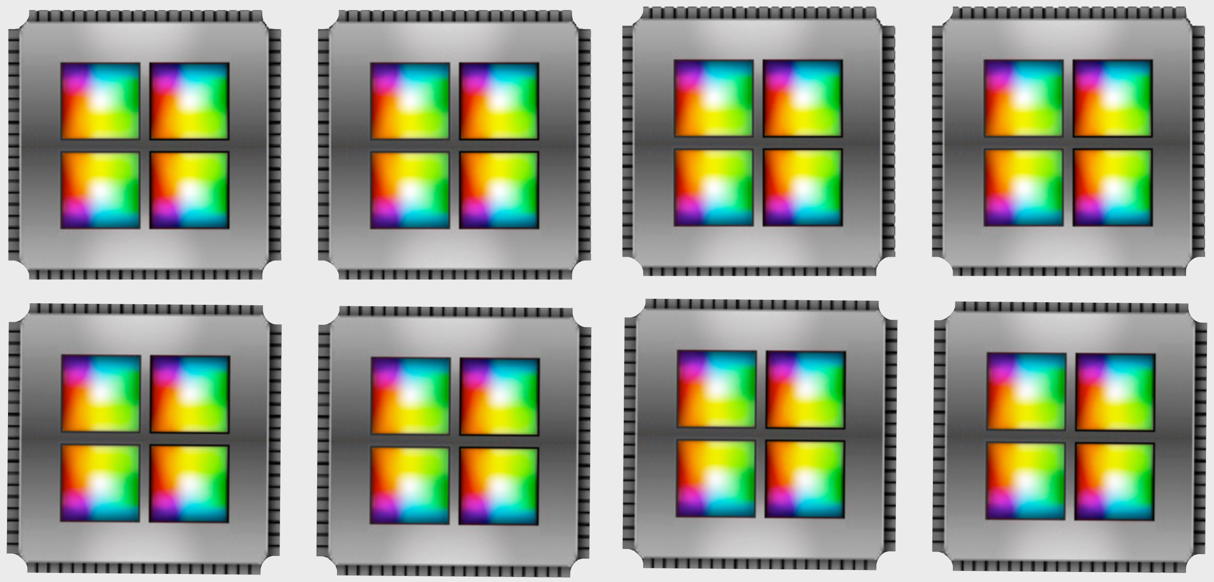
Implement your own LVars, too

`cabal install lvish` today!

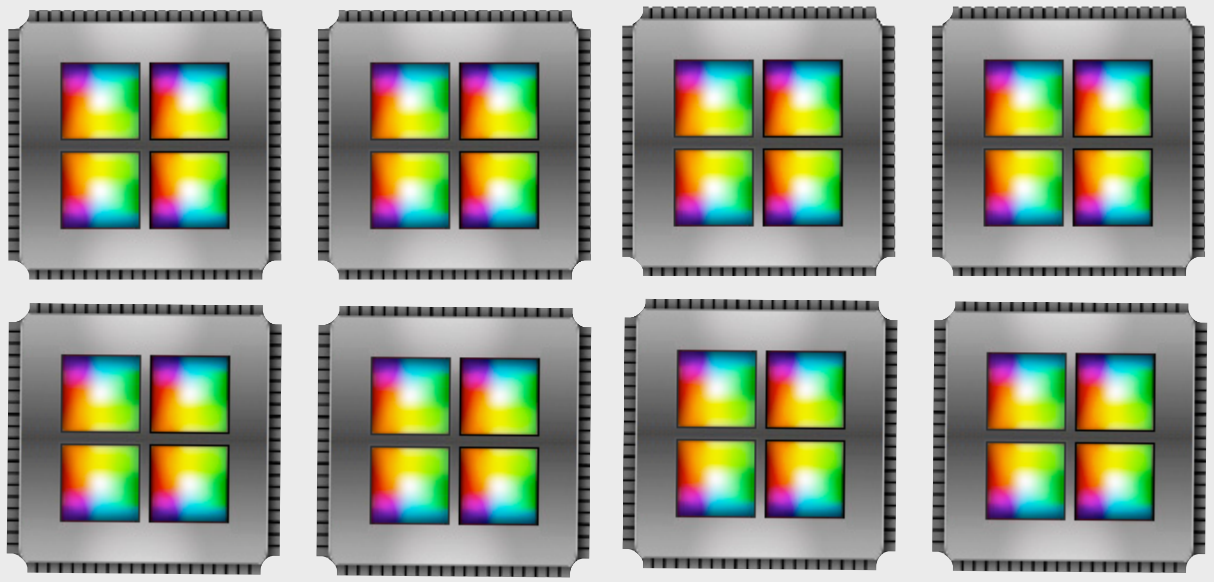
```haskell
p :: Par Det (IMap Item Int)
p = do
  cart <- newEmptyMap
  fork (insert Book 1 cart)
  fork (insert Shoes 1 cart)
  return cart

main = do
  putStr (show (toList (fromIMap
    (runParThenFreeze p))))
```



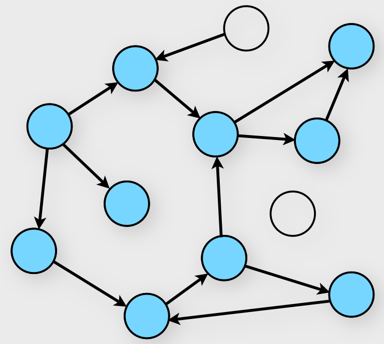
Deterministic Parallel Programming



(observably)
Deterministic Parallel Programming

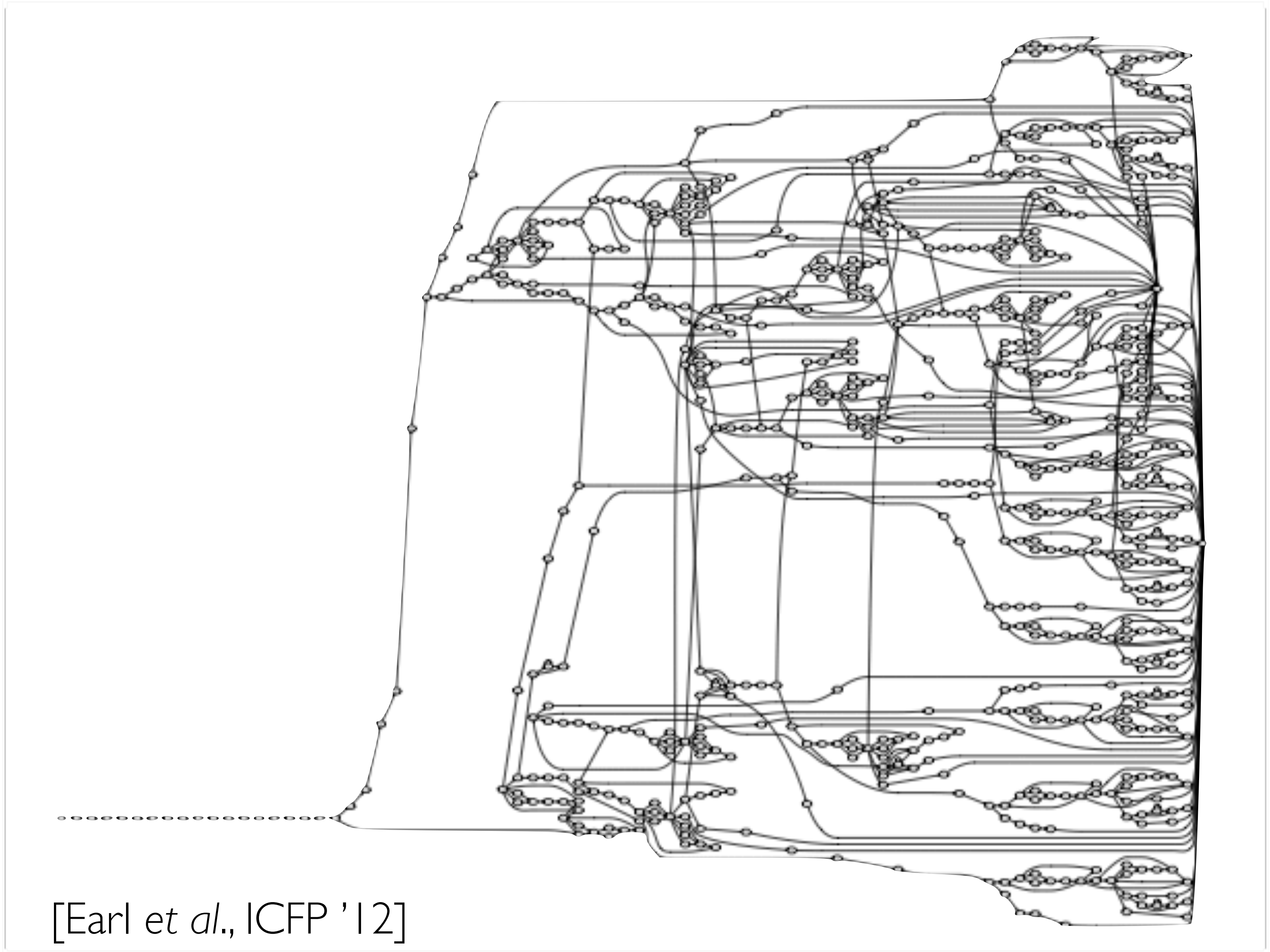


(observably) *(irregular)*
Deterministic Parallel Programming



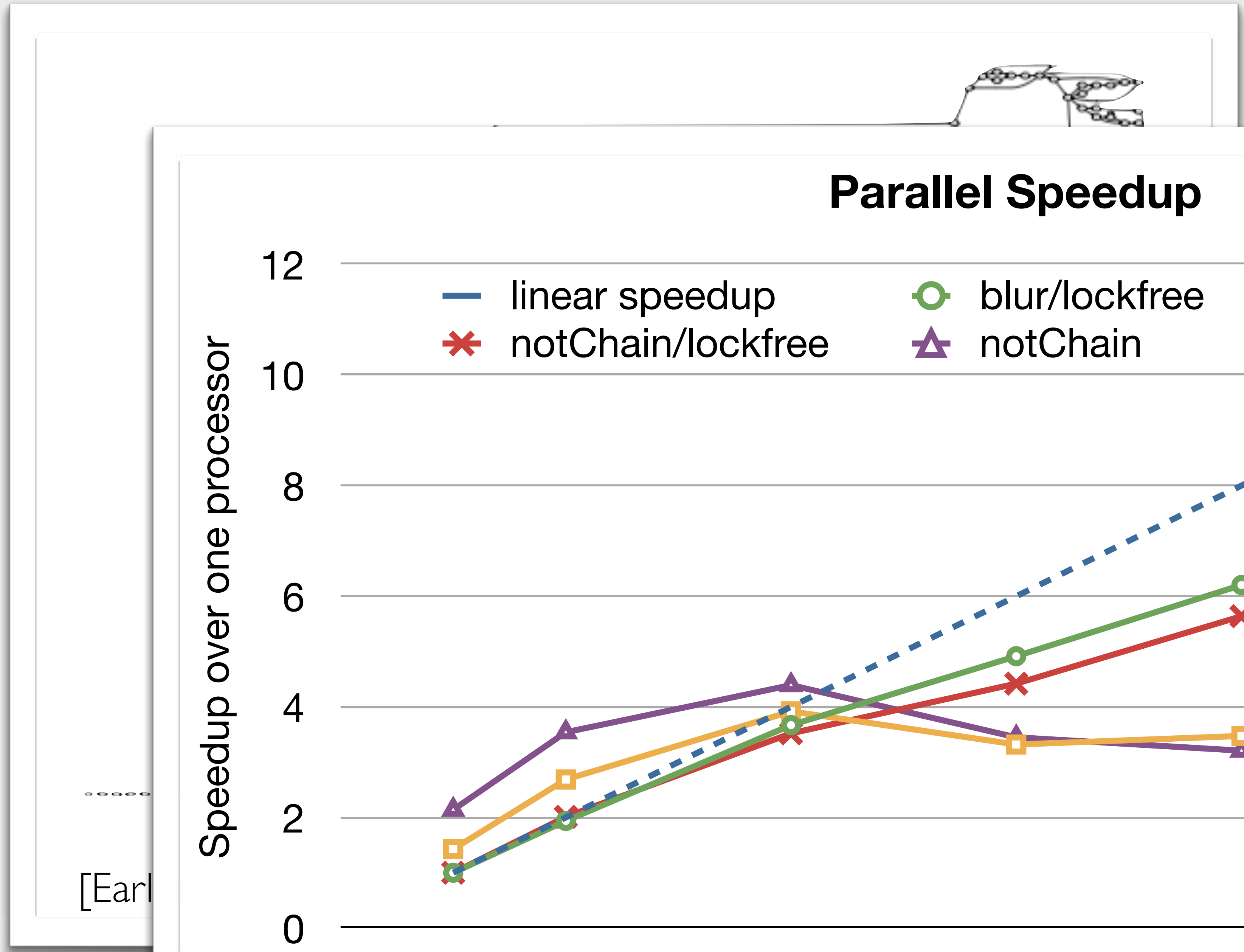
Case study:
k-CFA static analysis parallelized with LVish

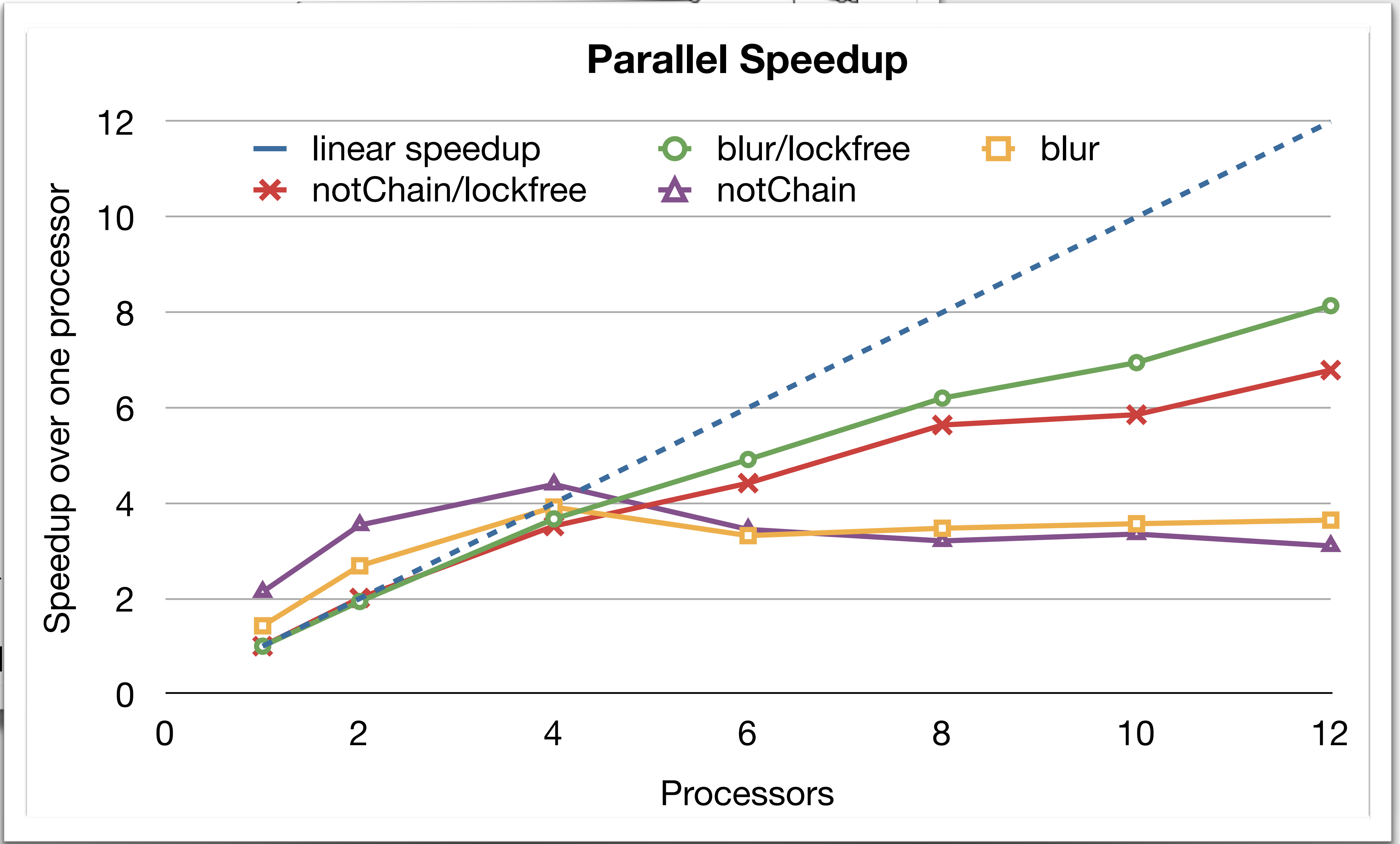
Case study:
k-CFA static analysis parallelized with LVish

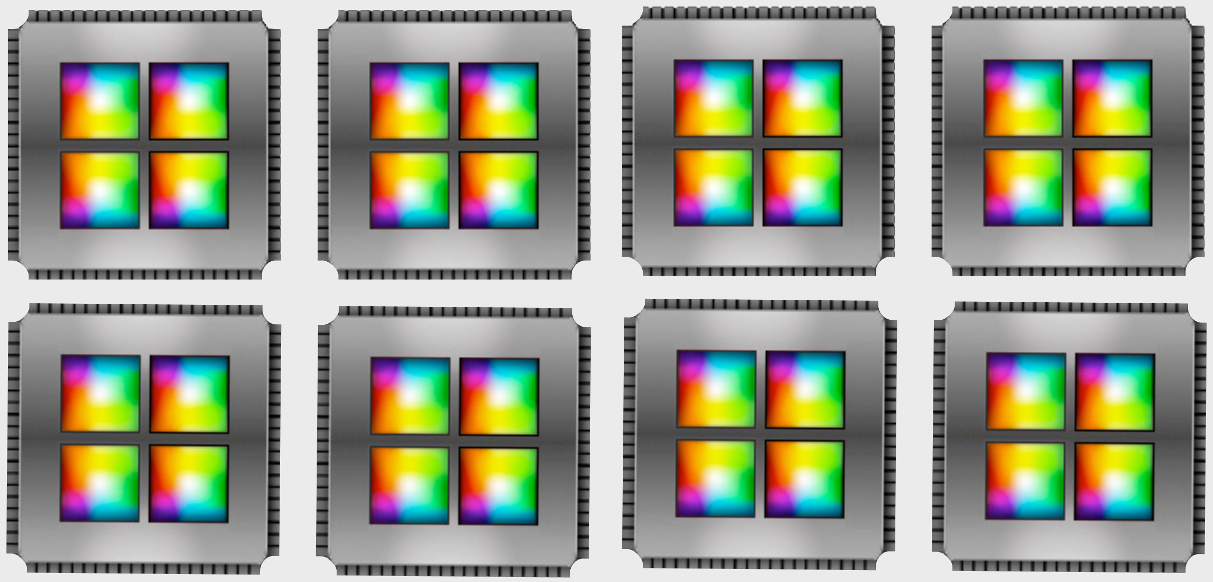


[Earl *et al.*, ICFP '12]

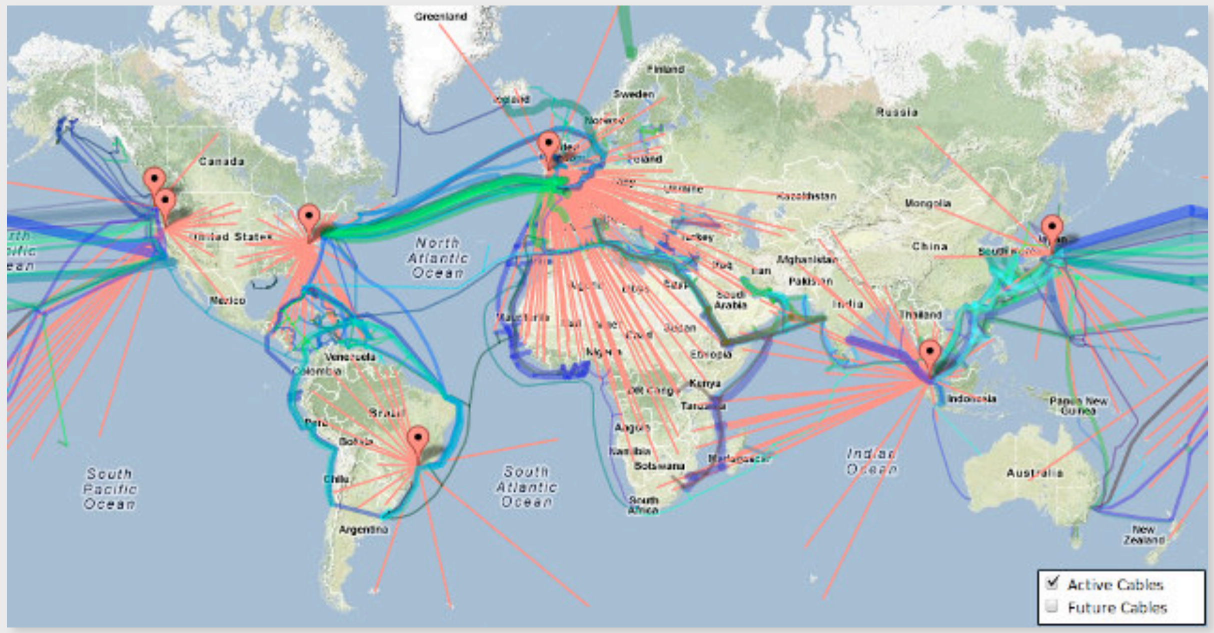
Case study:
k-CFA static analysis parallelized with LVish


[Earl

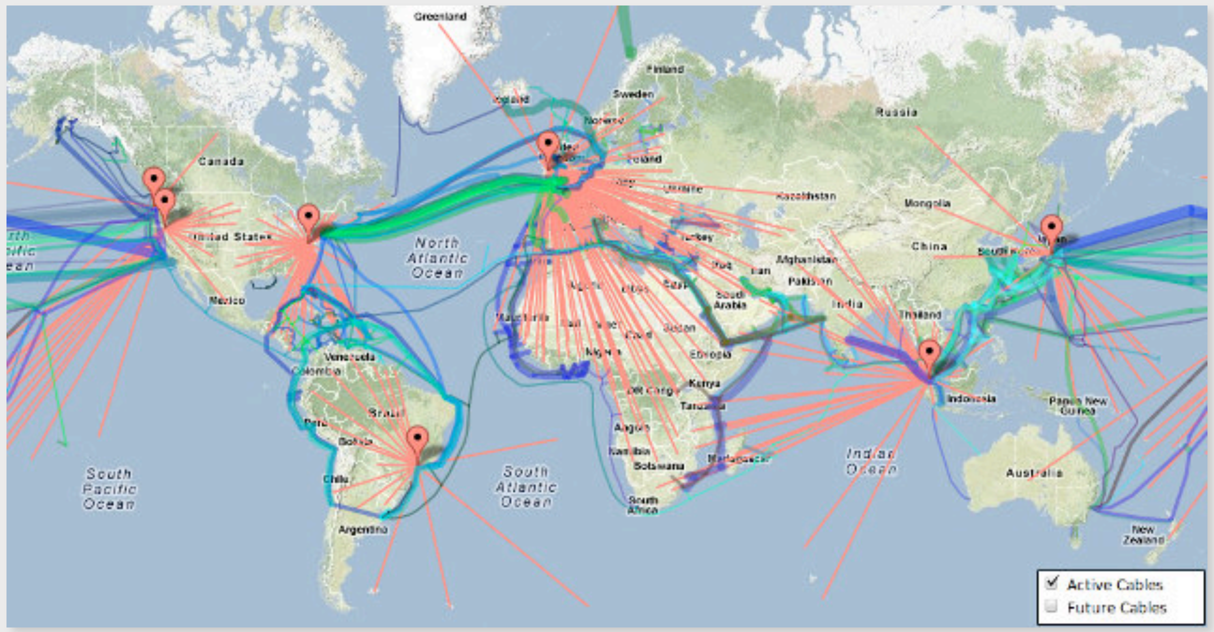




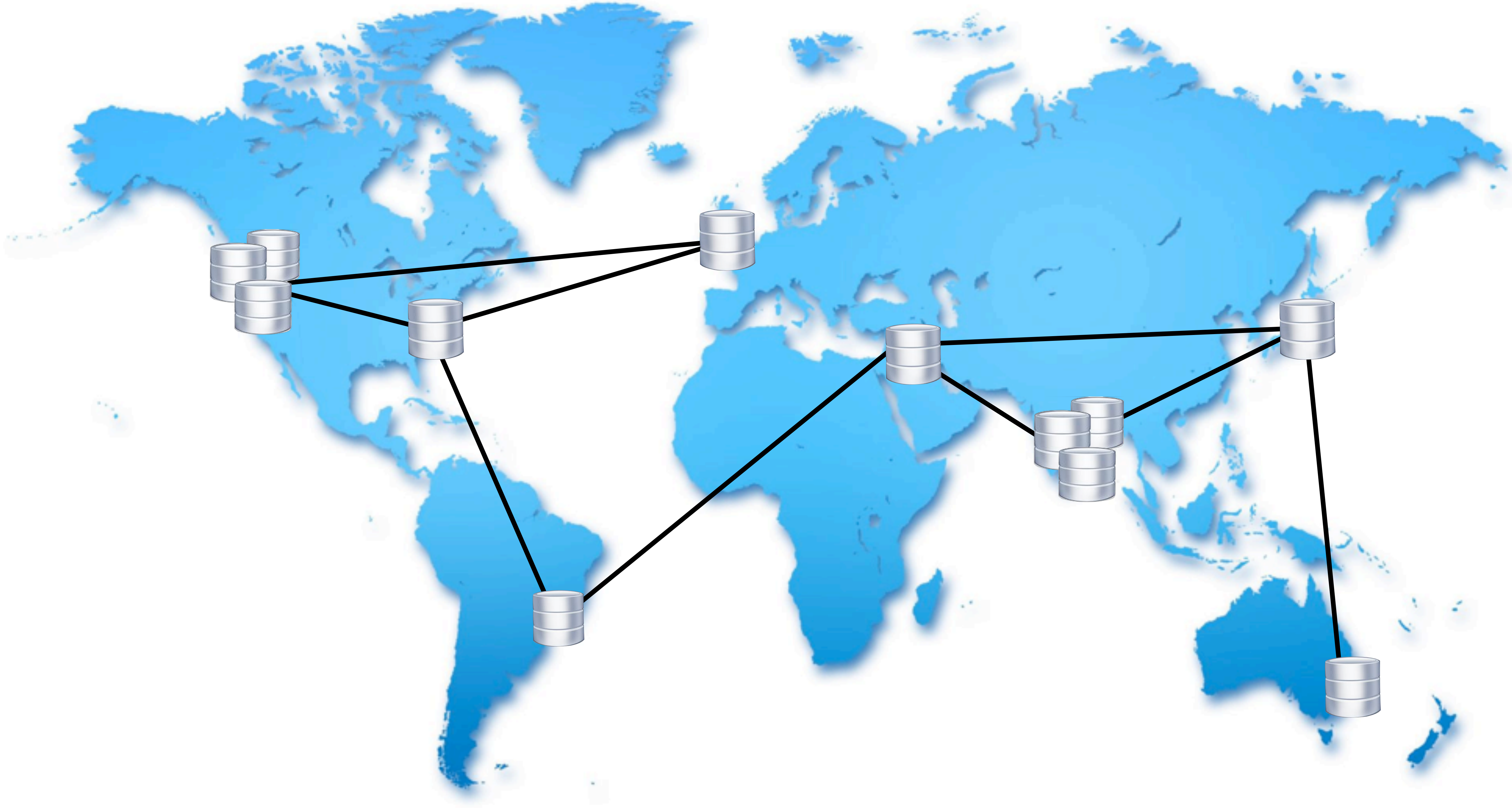
Parallel systems

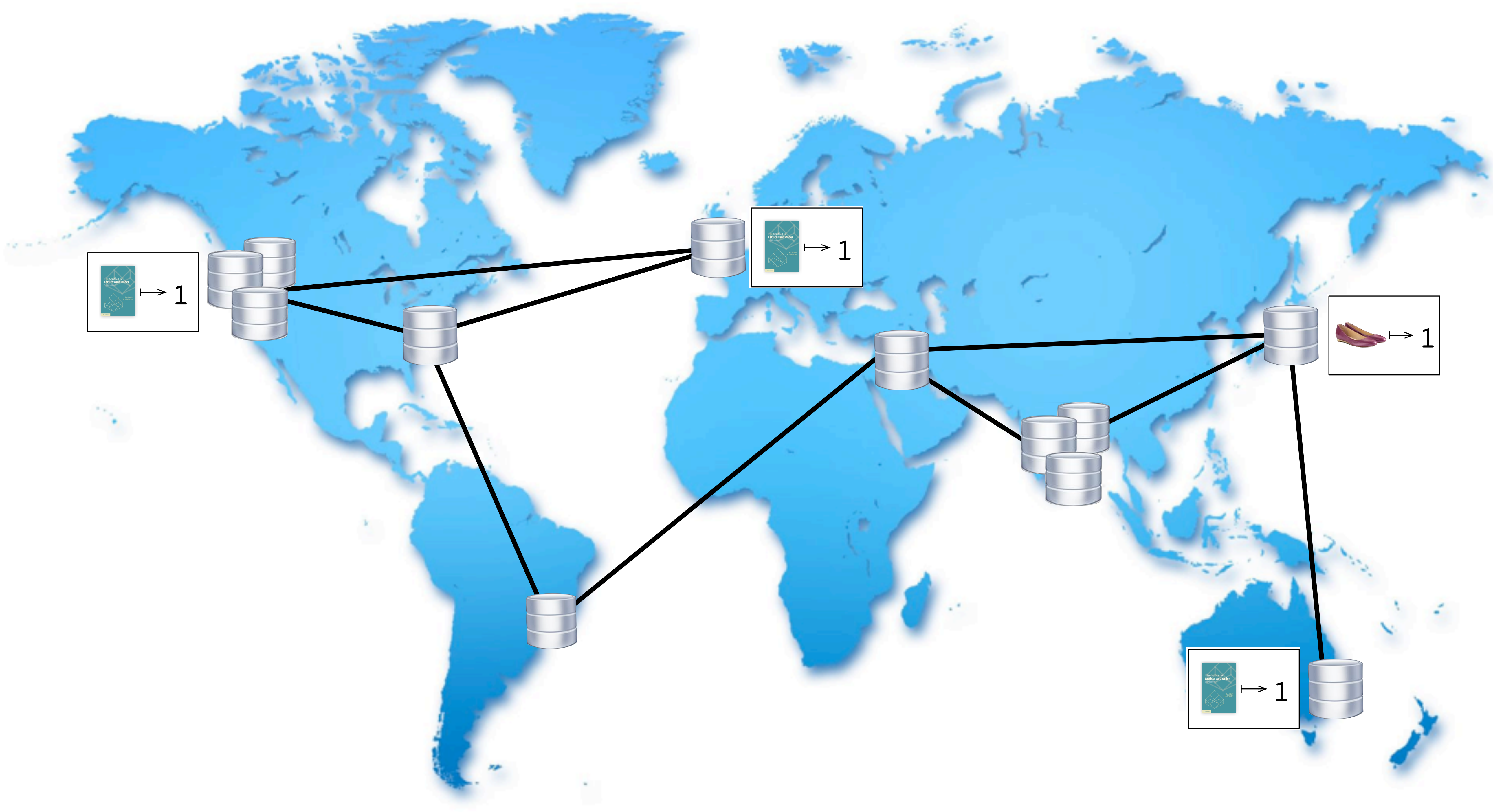


Distributed systems

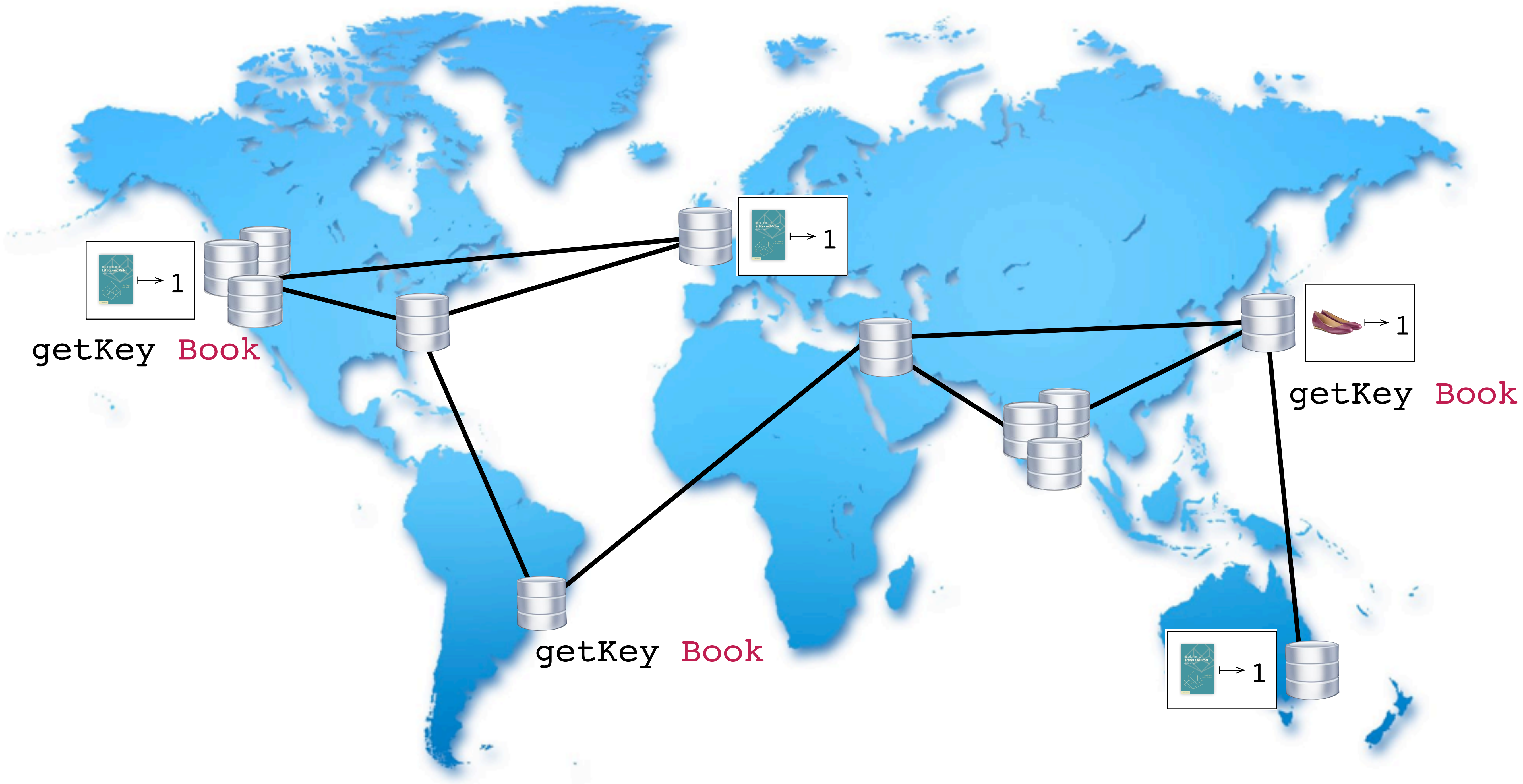


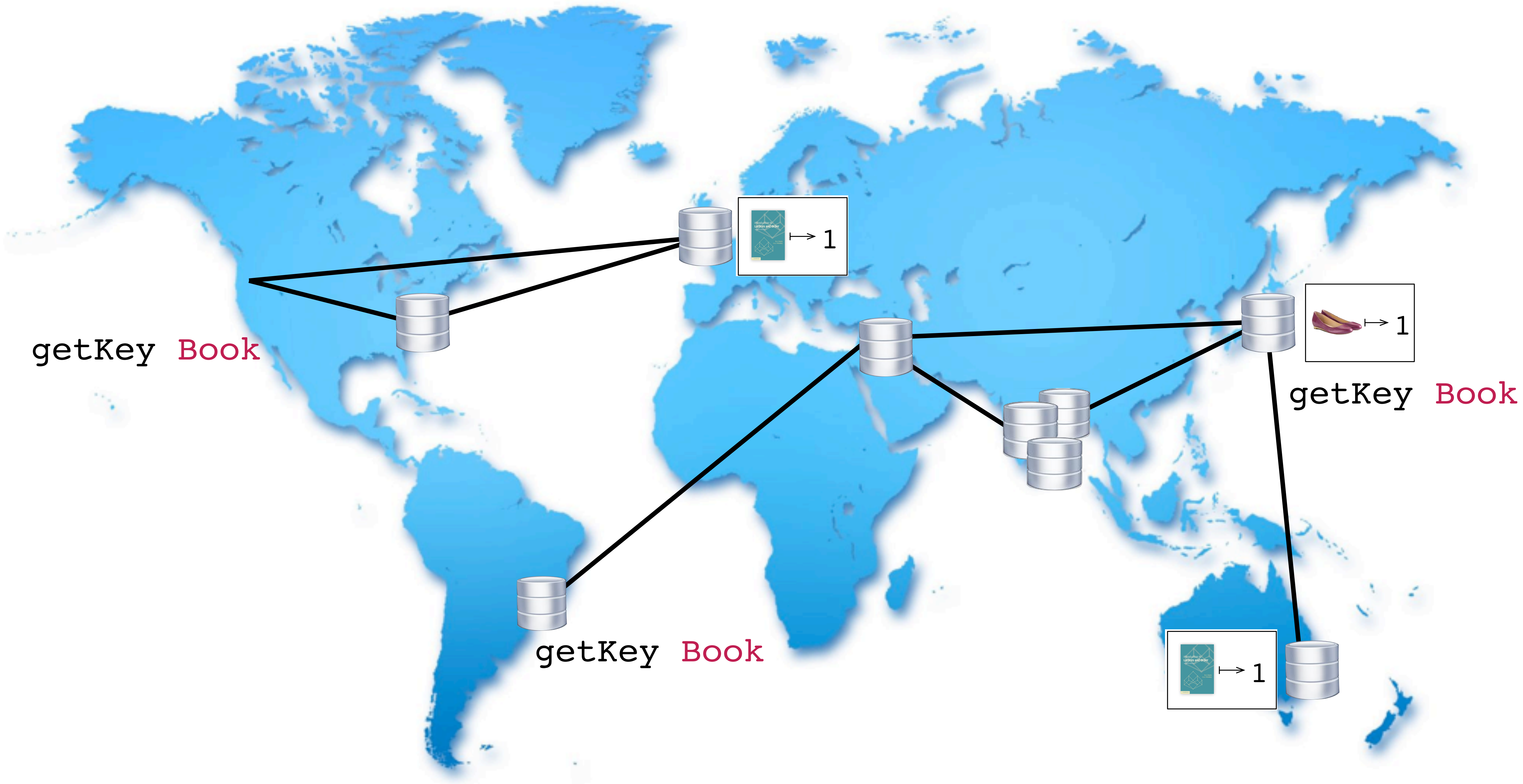
Distributed systems

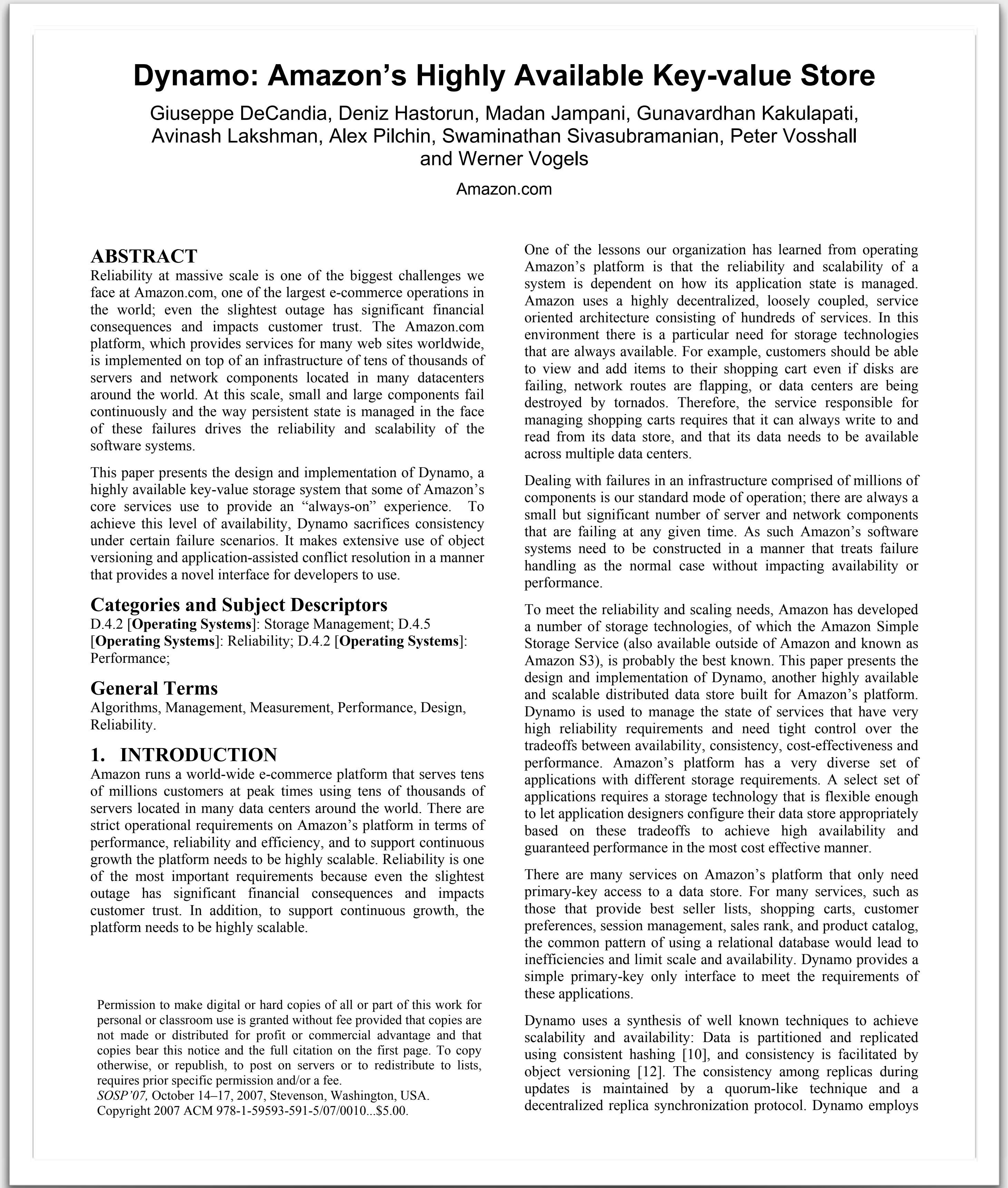




$\mapsto 1$
$\mapsto 1$
$\mapsto 1$
$\mapsto 1$


↦ 1
getKey Book
↦ 1
↦ 1
getKey Book
getKey Book
↦ 1


↦ 1
↦ 1
↦ 1
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.

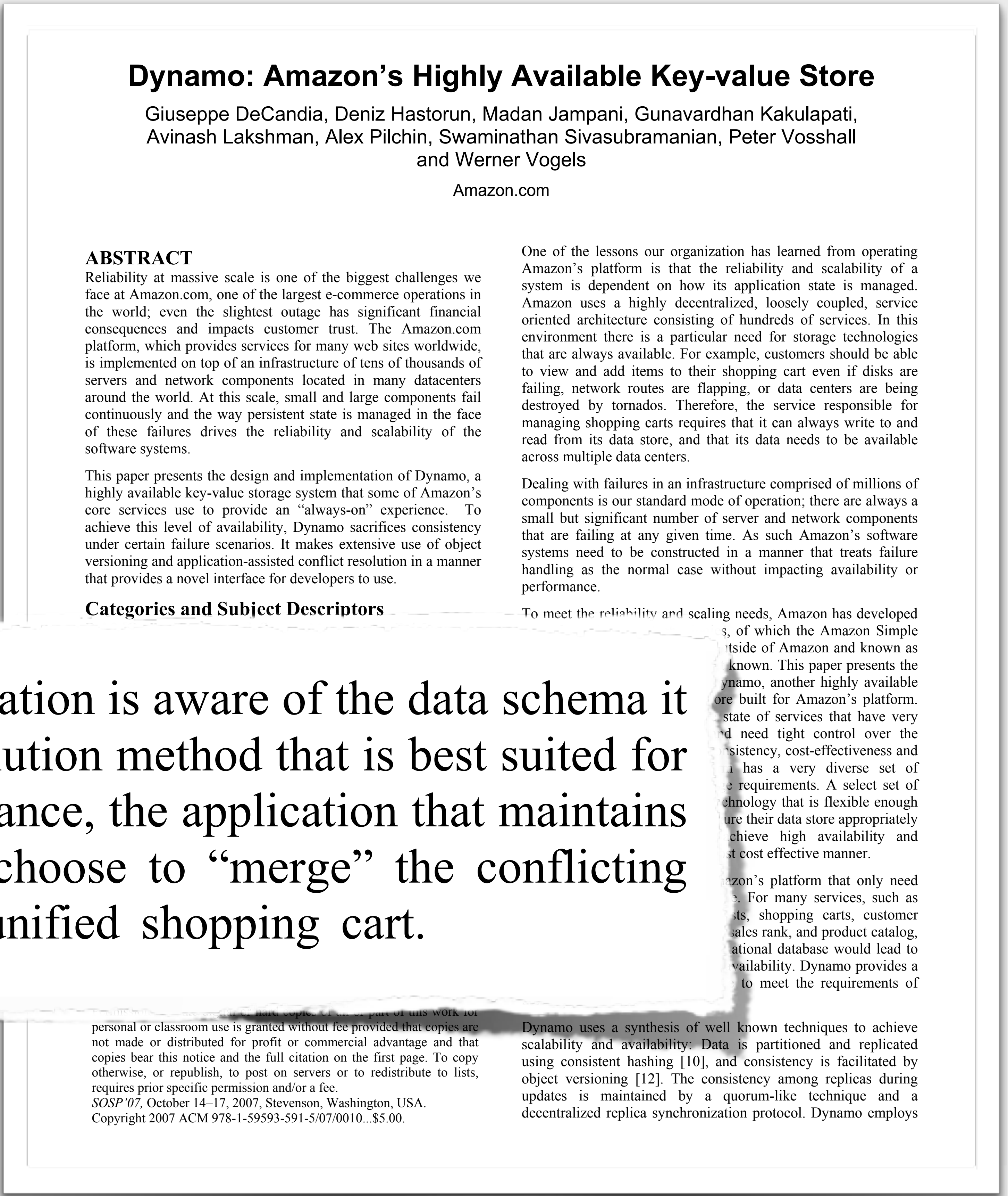
Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that 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

[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

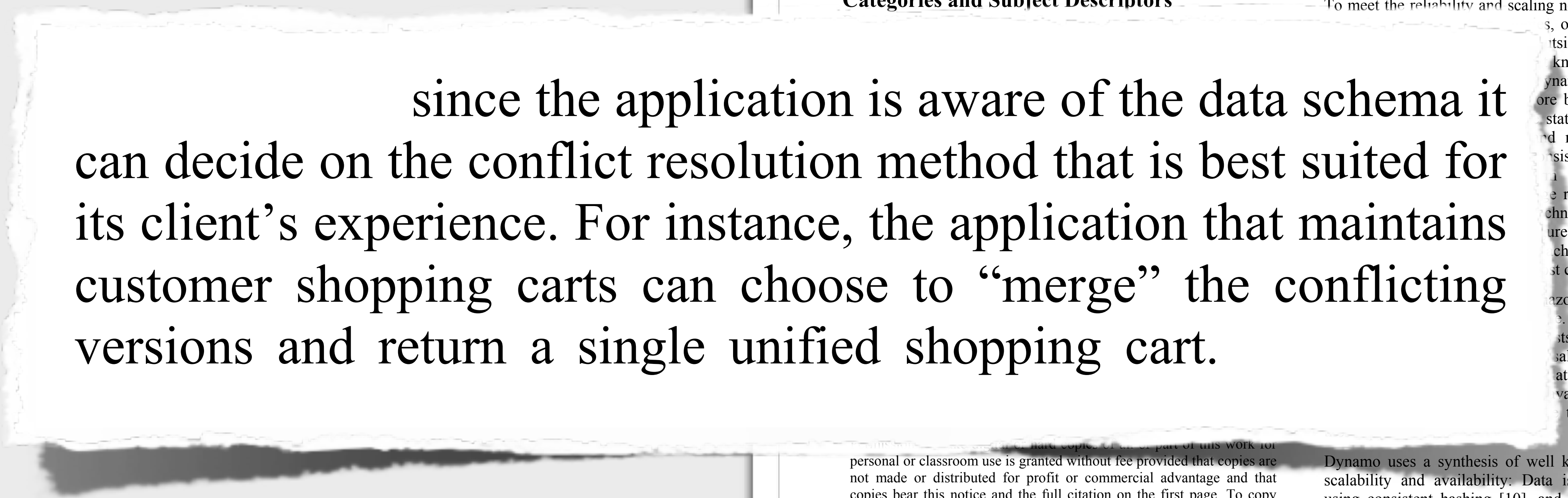
One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. 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
t cost effective manner.

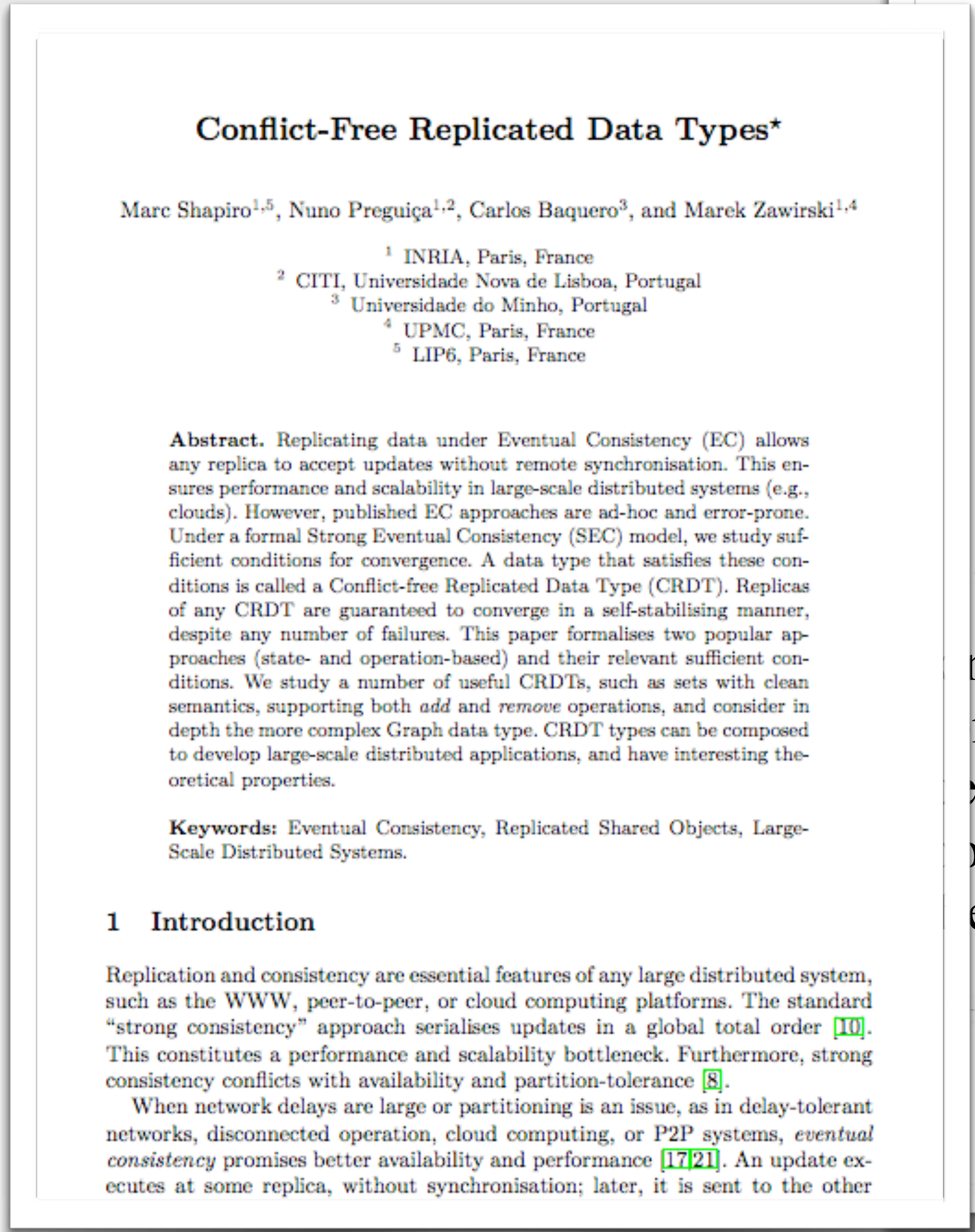
zon's platform that only need
. For many services, such as
sts, shopping carts, customer
sales rank, and product catalog,
ational database would lead to
vailability. Dynamo provides a
to meet the requirements of

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



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.

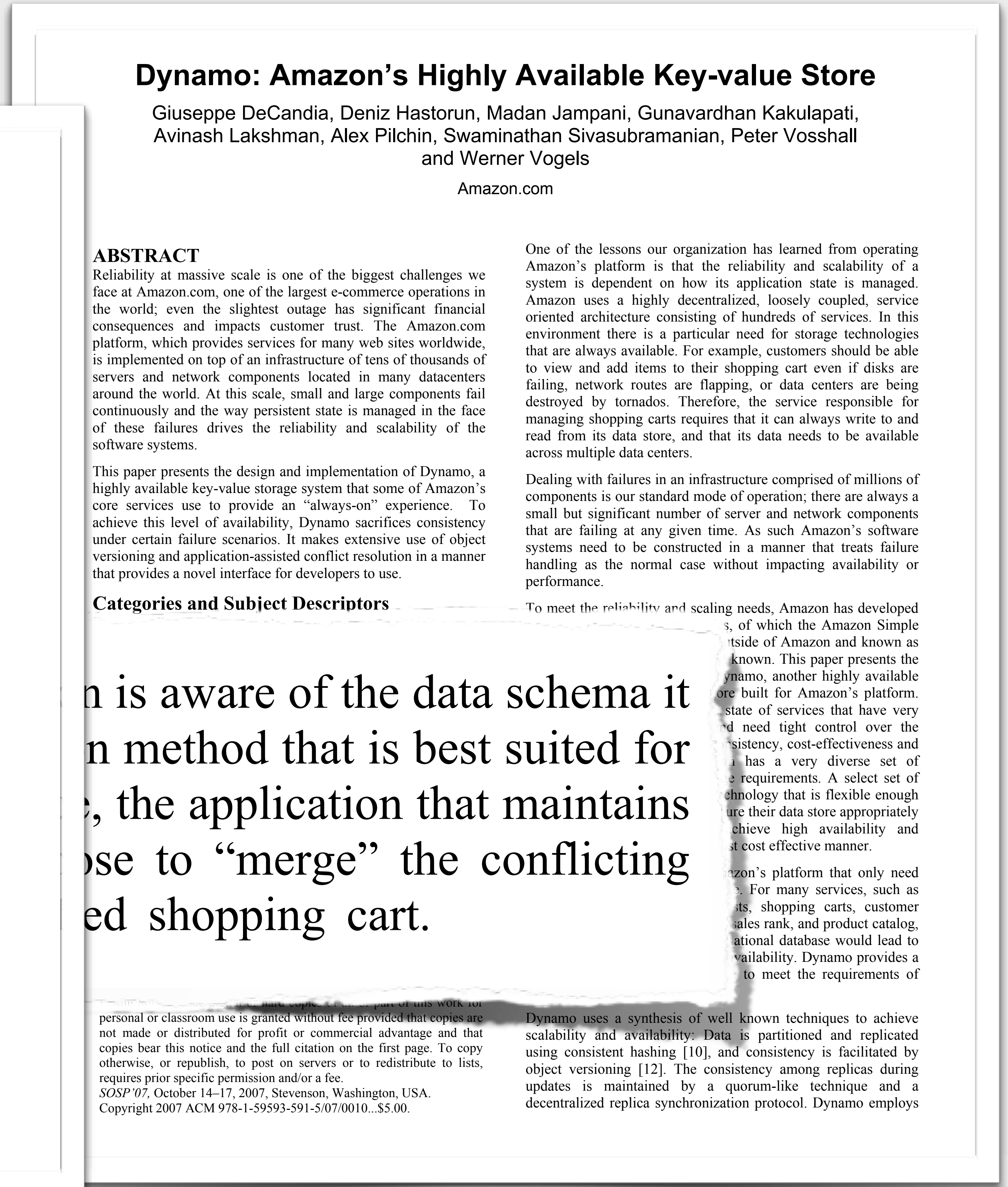
Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

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

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 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.

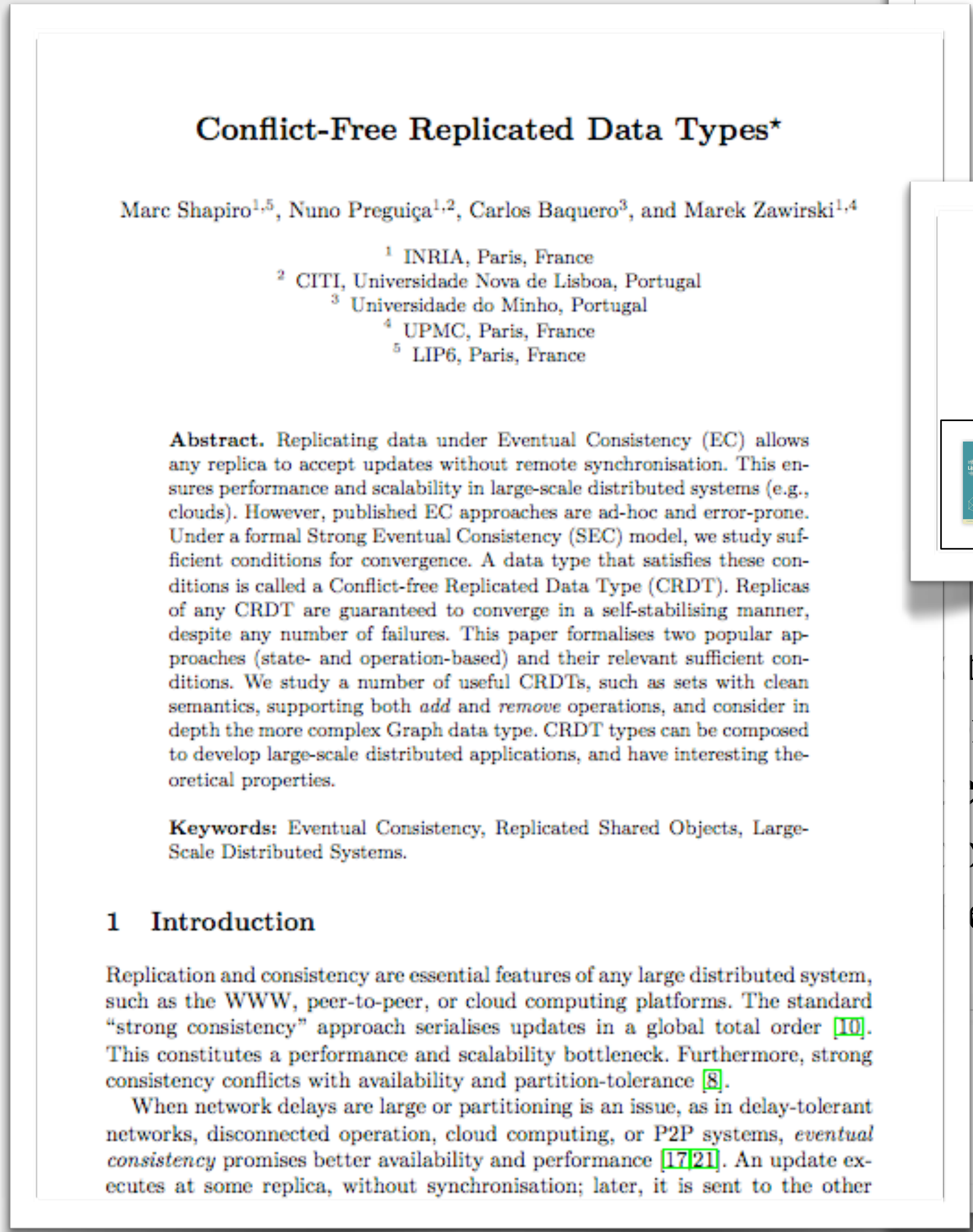
To meet the reliability and scaling needs, Amazon has developed
s, of which the Amazon Simple
tside of Amazon and known as
known. This paper presents the
ynamo, another highly available
ore built for Amazon's platform.
state of services that have very
d need tight control over the
sistency, cost-effectiveness 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

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]



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.

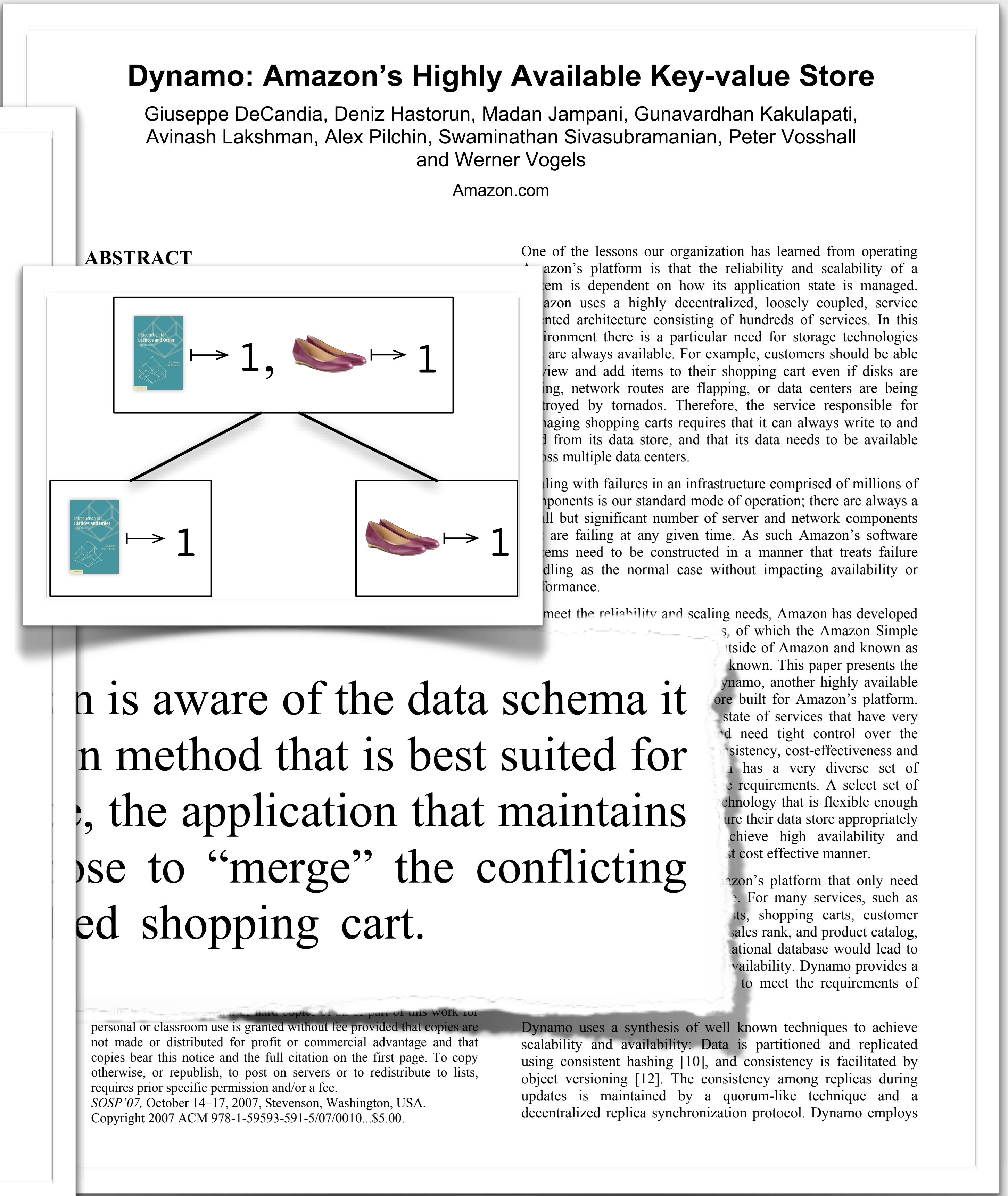
Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

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]

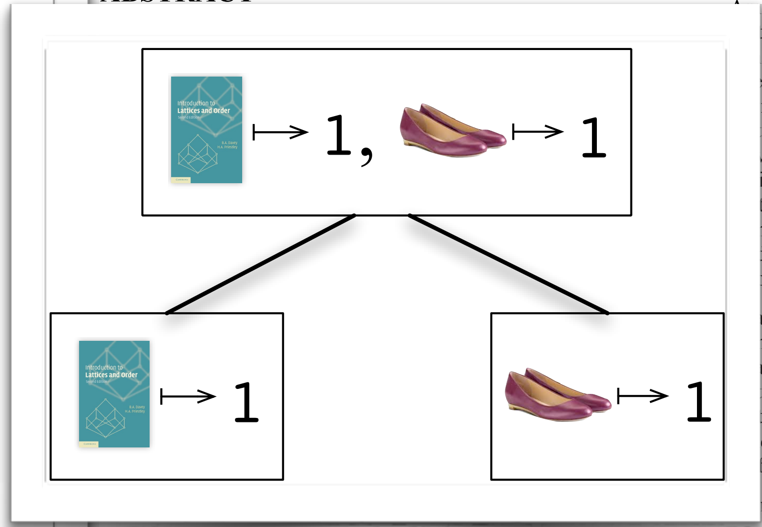


Dynamo: Amazon's Highly Available Key-value Store

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

Amazon.com

ABSTRACT

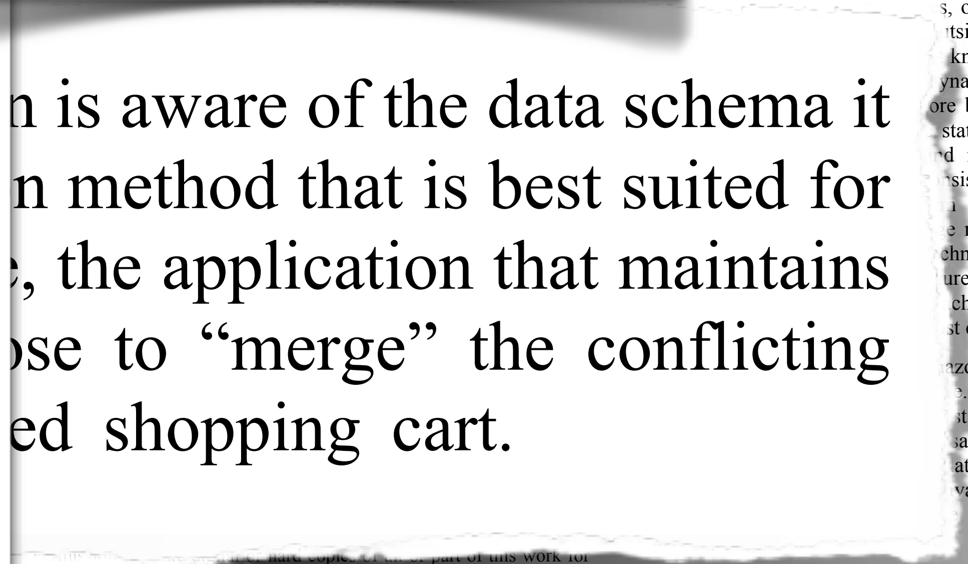


One of the lessons our organization has learned from operating azon's platform is that the reliability and scalability of a em is dependent on how its application state is managed. azon uses a highly decentralized, loosely coupled, service nted architecture consisting of hundreds of services. In this 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 l from its data store, and that its data needs to be available ss multiple data centers.

ling with failures in an infrastructure comprised of millions of nponents 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



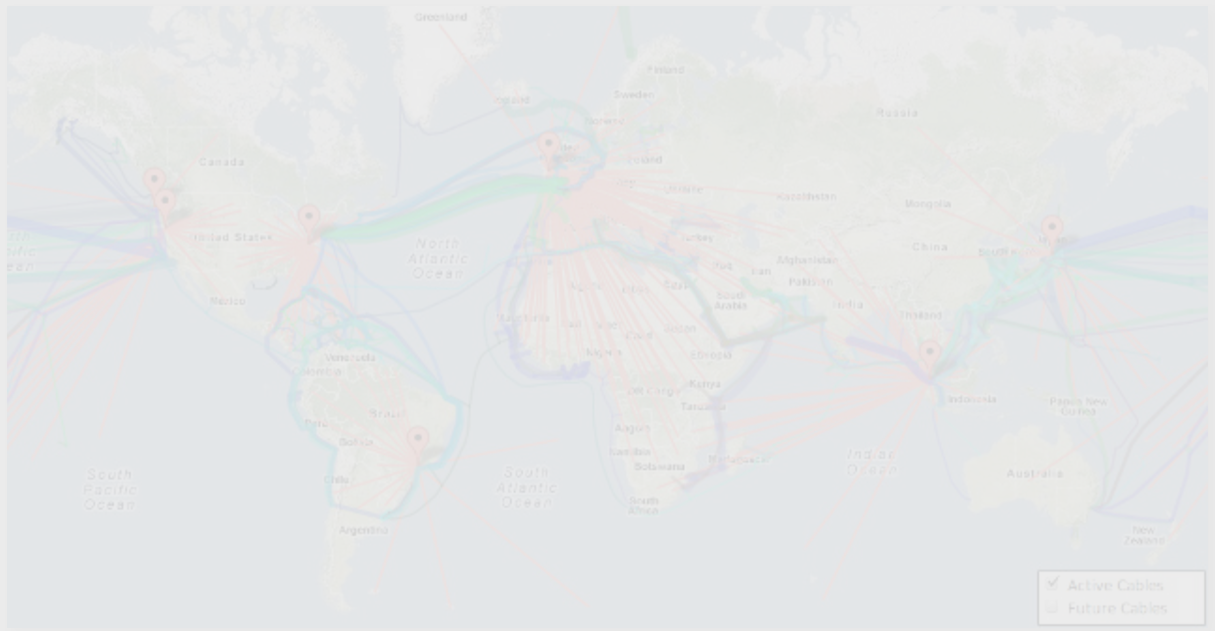
n is aware of the data schema it n method that is best suited for , the application that maintains se to "merge" the conflicting ed shopping cart.

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]

Threshold reads of CRDTs

Consistency choices at the granularity of *queries*, not that of databases

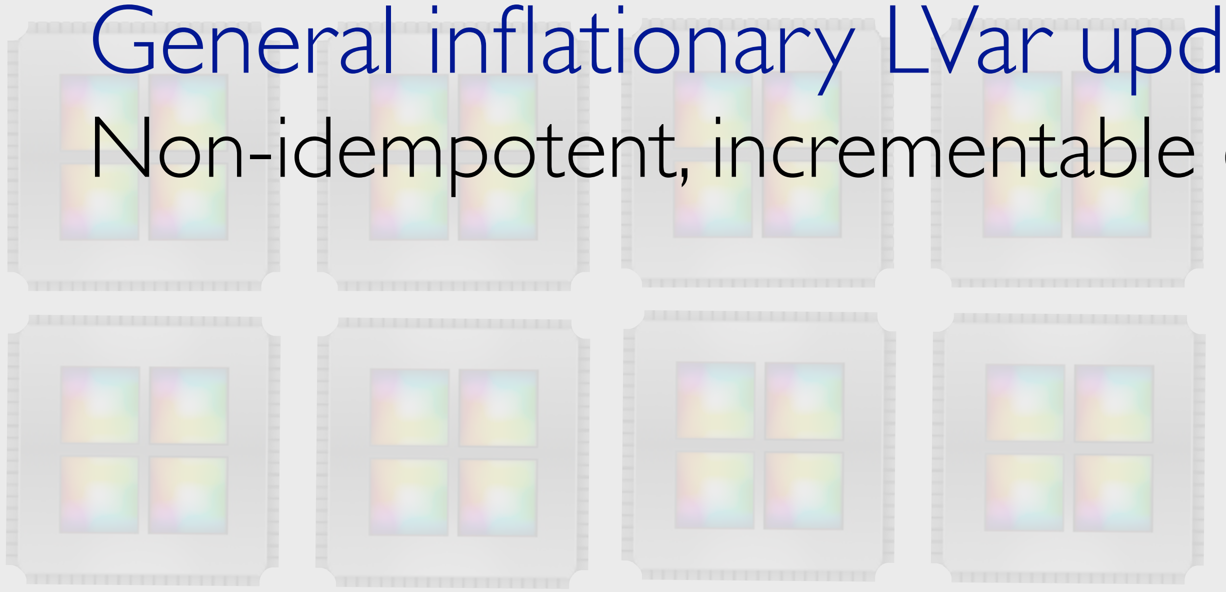


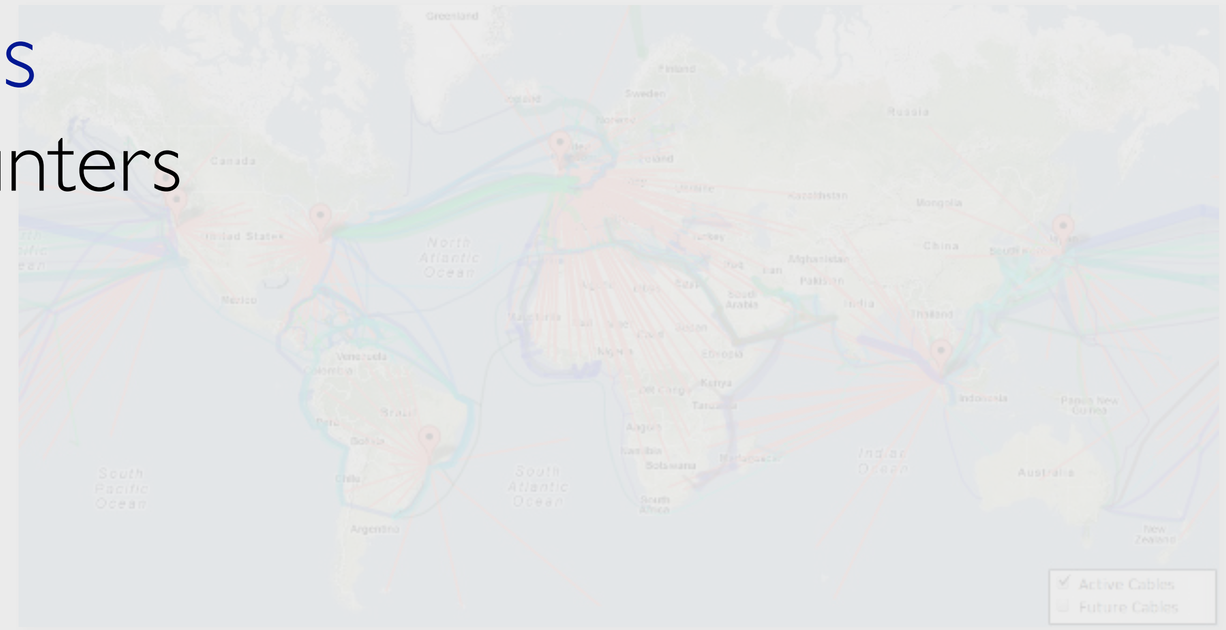
Threshold reads of CRDTs

Consistency choices at the granularity of *queries*, not that of databases

General inflationary LVar updates

Non-idempotent, incrementable counters





Threshold reads of CRDTs

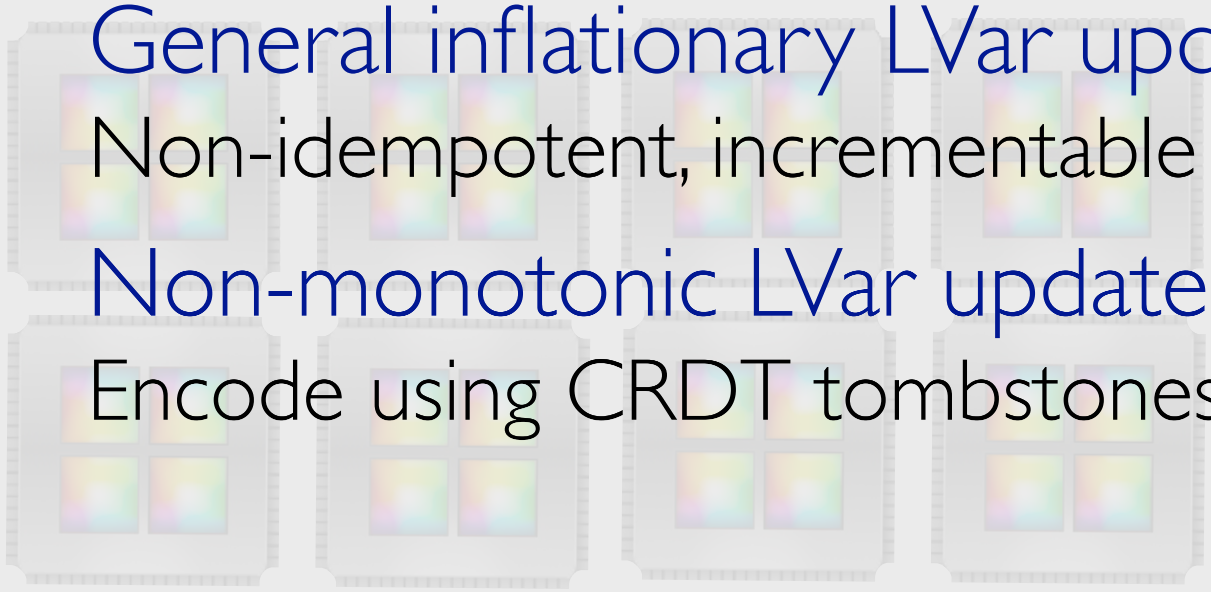
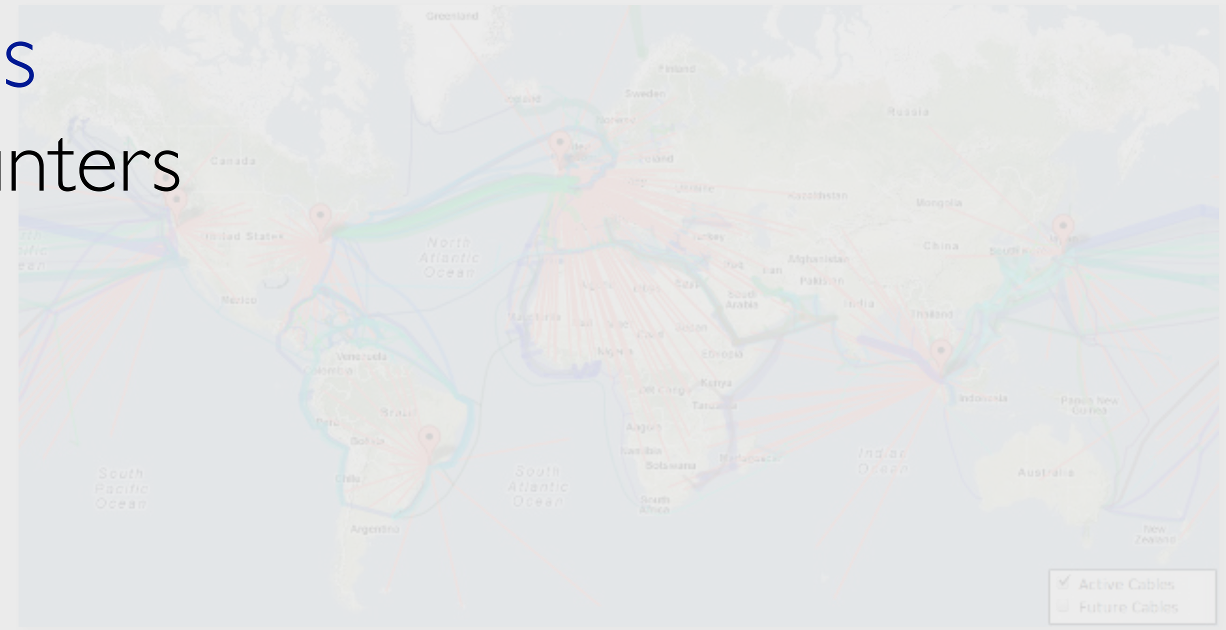
Consistency choices at the granularity of *queries*, not that of databases

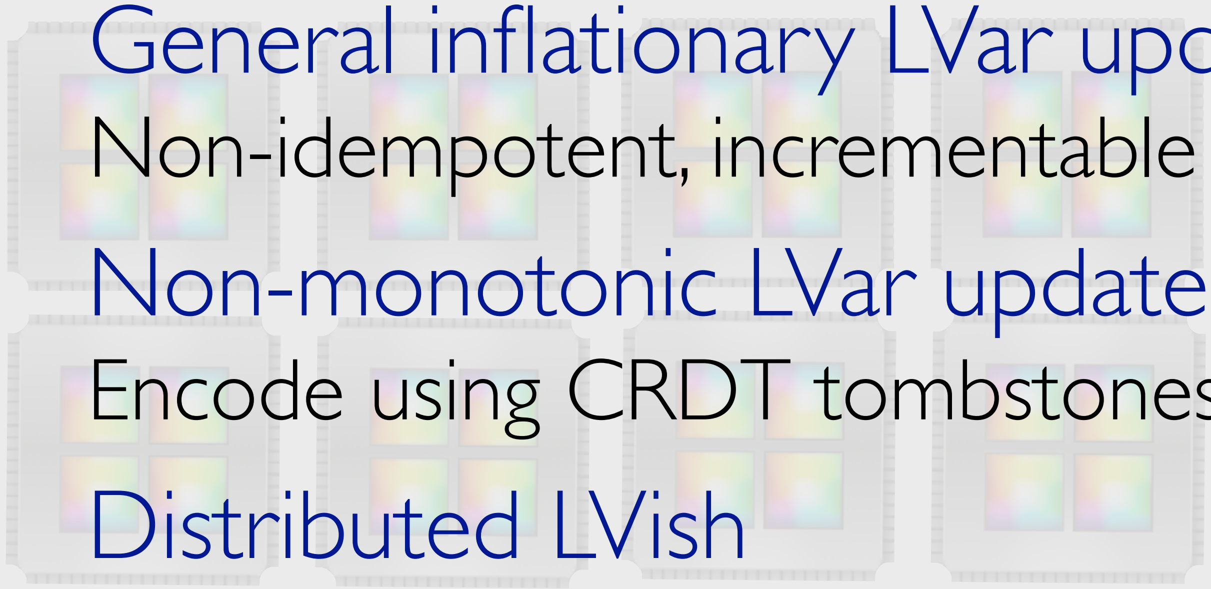
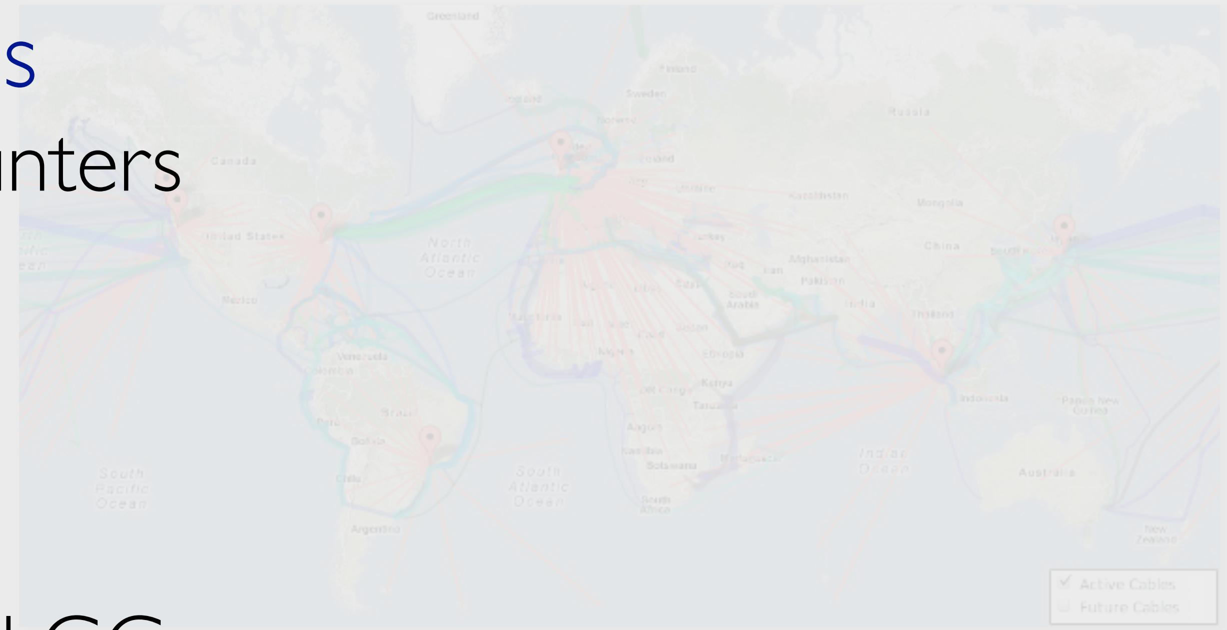
General inflationary LVar updates

Non-idempotent, incrementable counters

Non-monotonic LVar updates

Encode using CRDT tombstones

Threshold reads of CRDTs

Consistency choices at the granularity of *queries*, not that of databases

General inflationary LVar updates

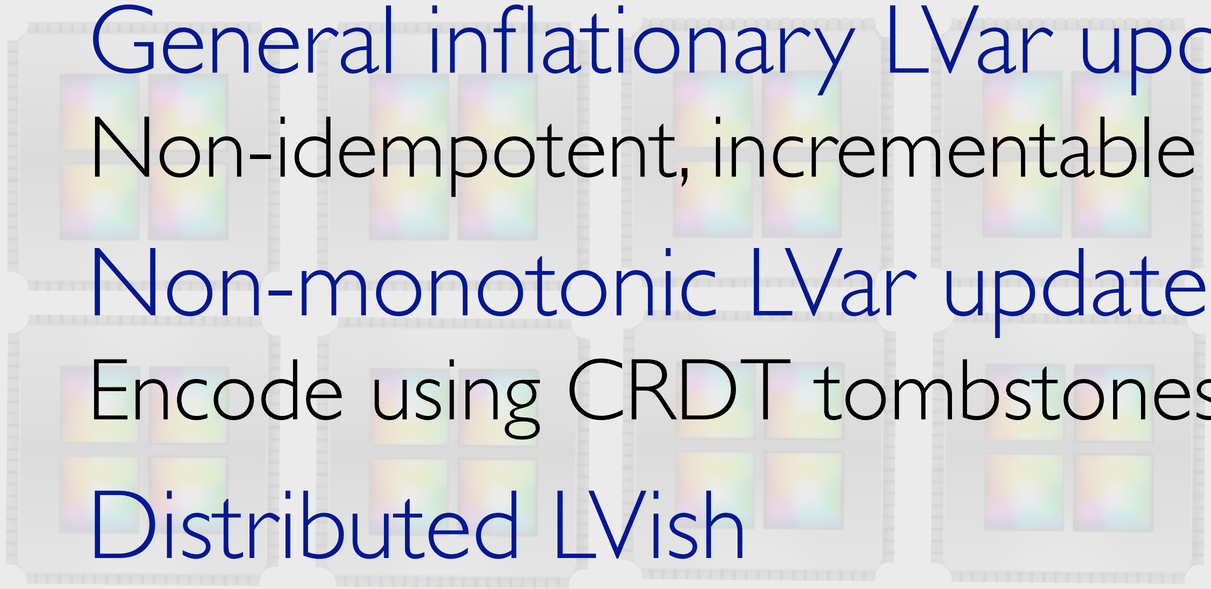
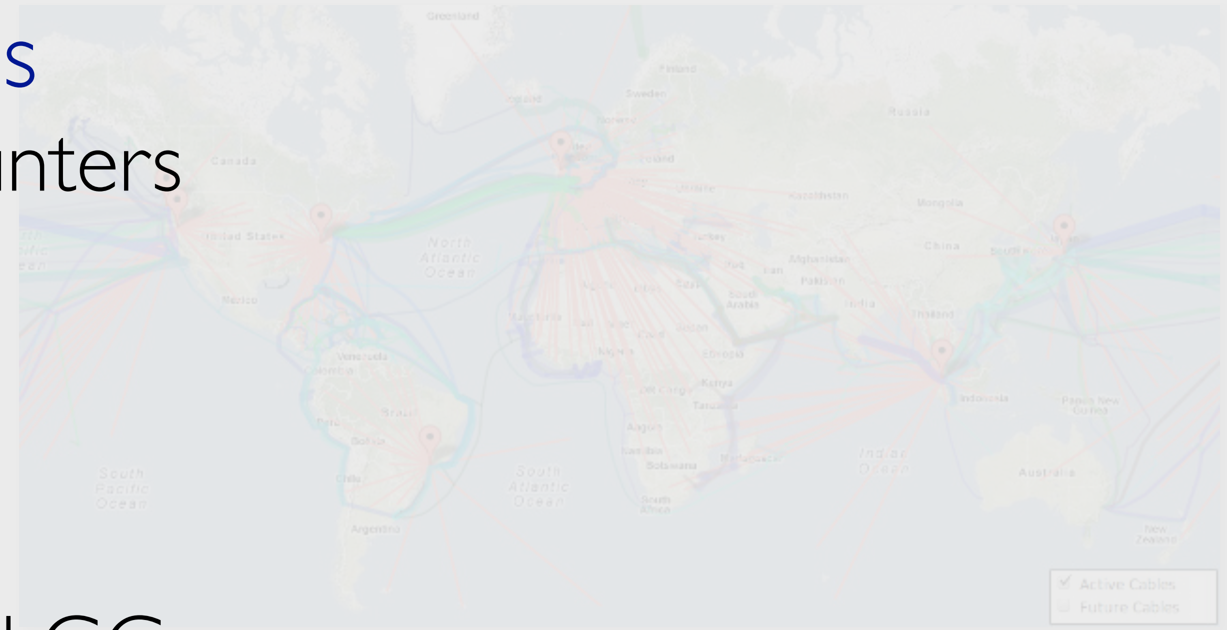
Non-idempotent, incrementable counters

Non-monotonic LVar updates

Encode using CRDT tombstones

Distributed LVish

Distributed work-stealing, distributed GC

Threshold reads of CRDTs

Consistency choices at the granularity of *queries*, not that of databases

General inflationary LVar updates

Non-idempotent, incrementable counters

Non-monotonic LVar updates

Encode using CRDT tombstones

Distributed LVish

Distributed work-stealing, distributed GC

Differential dataflow for LVars

Handling continuous input

- Threshold reads of CRDTs

  Consistency choices at the granularity of *queries*, not that of databases

- General inflationary LVar updates

  Non-idempotent, incrementable counters

  Non-monotonic LVar updates

  Encode using CRDT tombstones

  Distributed LVish

  Distributed work-stealing, distributed GC

  Differential dataflow for LVars

  Handling continuous input

Joining Forces

Toward a Unified Account of LVars and Convergent Replicated Data Types

Lindsey Kuper   Ryan R. Newton
Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract

LVars—shared memory locations whose semantics are defined in terms of an application-specific lattice—offer a principled approach to deterministic-by-construction, shared-state parallel programming: writes to an LVar take the join of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. This semantics guarantees that programs have a deterministic outcome, despite parallel execution and schedule nondeterminism.

LVars have a close cousin in the distributed systems literature: *convergent replicated data types* (CvRDTs), which leverage lattice properties to guarantee that all replicas of a distributed object (for instance, in a distributed database) are *eventually consistent*. Unlike LVars, in which all updates are joins, CvRDTs allow updates that are inflationary with respect to the lattice but do not compute a join. Moreover, CvRDTs differ from LVars in that they allow intermediate states to be observed: if two replicas of an object are updated independently, reads of those replicas may disagree until a (least-upper-bound) merge operation takes place.

Although CvRDTs and LVars were developed independently, LVars ensure determinism under parallel execution by leveraging the same lattice properties that CvRDTs use to ensure eventual consistency. Therefore, a sensible next research question is: how can we take inspiration from CvRDTs to improve the LVars model, and vice versa? In this paper, we take steps toward answering that question in both directions: we consider both how to extend CvRDTs with LVar-style threshold reads and how to extend LVars with CvRDT-style inflationary updates, and we advocate for the usefulness of these extensions.

1. Introduction

Deterministic-by-construction parallel programming models ensure that all programs written using the model have the same observable behavior every time they are run, offering freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. Ideally, a deterministic-by-construction parallel program will run faster when more parallel resources are available, and so we do *not* want our model to require that exact scheduling behavior is deterministic; only a program's *outcome* should be preserved across multiple runs. Indeed, we want to specifically *allow* tasks to be scheduled dynamically and unpredictably, in order to handle irregular parallel applications, but without allowing such *schedule nondeterminism* to affect the outcome of a program.

In earlier work [9, 10], we proposed *LVars* as a principled approach to shared-state parallel programming that guarantees observably deterministic outcomes. An LVar is a memory location that can be shared among multiple threads and accessed through `put` (write) and `get` (read) operations. Unlike a typical shared mutable location, though, the values an LVar can take on are elements of an application-specific *lattice*. This application-specific lattice determines the semantics of the `put` and `get` operations that comprise the interface to LVars:

- `put` operations can only change an LVar's state in a way that is *monotonically increasing* with respect to the application-specific lattice, because it updates the LVar to the *join*, or least upper bound, of the old state and the new state.
- `get` operations allow only limited observations of the state of an LVar. A `get` operation requires the programmer to specify a *threshold set* of minimum values that can be read from the LVar, where every two elements in the threshold set must have the lattice's greatest element $\top$ as their join. A call to a `get` operation blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns *that value*, rather than the LVar's exact contents.

Together, monotonically increasing writes via `put` and threshold reads via `get` yield a deterministic-by-construction programming model. That is, a program in which `put`s and `get`s on LVars are the only side effects will have the same observable result on every run, in spite of parallel execution and schedule nondeterminism [9].

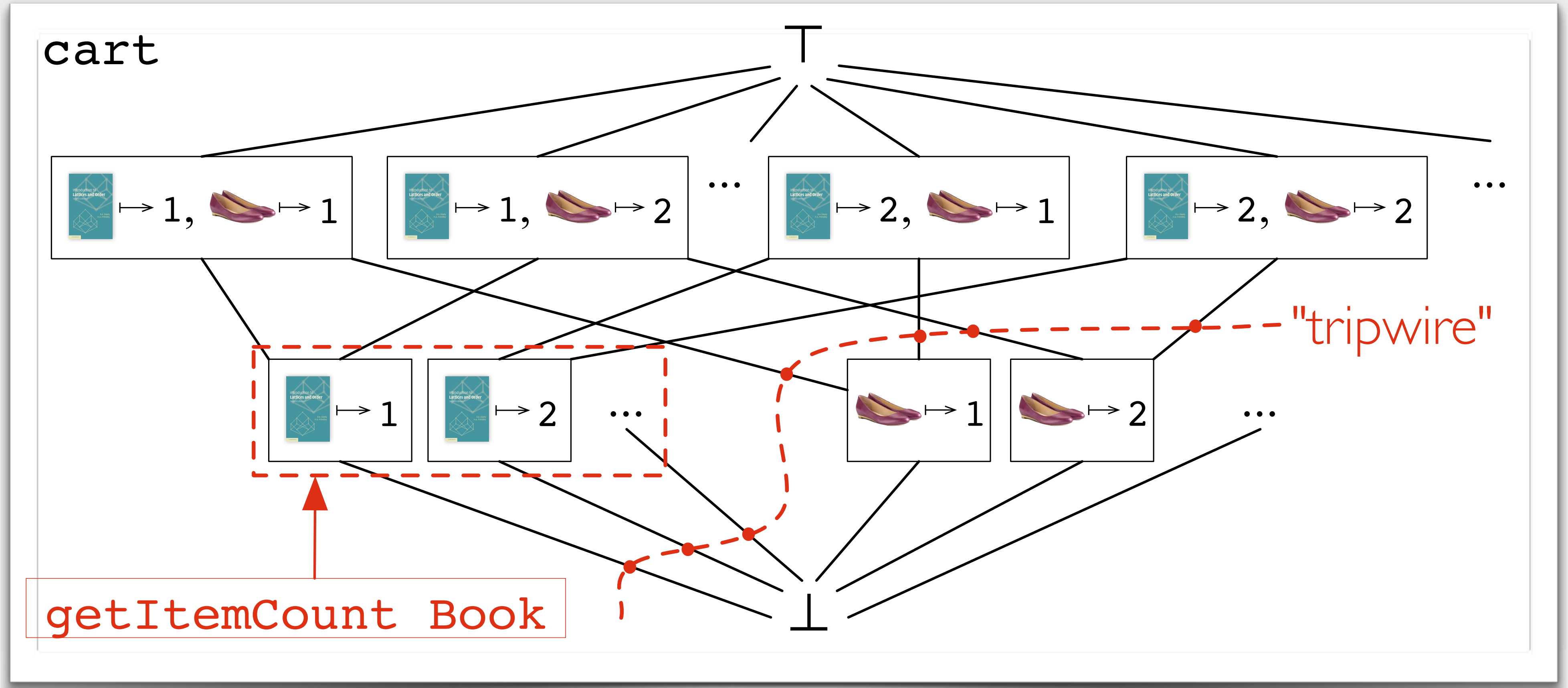
Lattices for eventual consistency The problem of ensuring determinism of parallel programs is closely related to the problem of ensuring the *eventual consistency* [14] of replicated objects in a distributed system. Consider, for example, an object representing the contents of a shopping cart, replicated across a number of physical locations. If two replicas disagree on the contents of the cart—for instance, if one replica sees only that item a has been added to the cart, while another sees only item b—how do we know what the "real" cart contents are? One option is to give every write a timestamp and allow the last-written replica to overrule the others, but such a "last-write-wins" policy does not necessarily make sense from a semantic point of view [7]. In the particular case of the shopping cart, we might instead want to resolve the conflict by taking the set union $\{a, b\}$ of the two replicas' contents; for some other application, a different policy might be more appropriate.

This notion of application-specific conflict resolution, long used by, for instance, the Amazon Dynamo key-value store [7], has recently been formalized in the setting of *convergent replicated data types* (CvRDTs) [12, 13]. A CvRDT is a replicated object in which the states that replicas can take on can be viewed as elements of a join-semilattice. While at any given time, replicas may differ, conflicts between replicas can always be deterministically resolved by a *merge* operation that computes the join of the two replicas' states. As long as all replicas merge with one another periodically, eventual consistency is guaranteed.

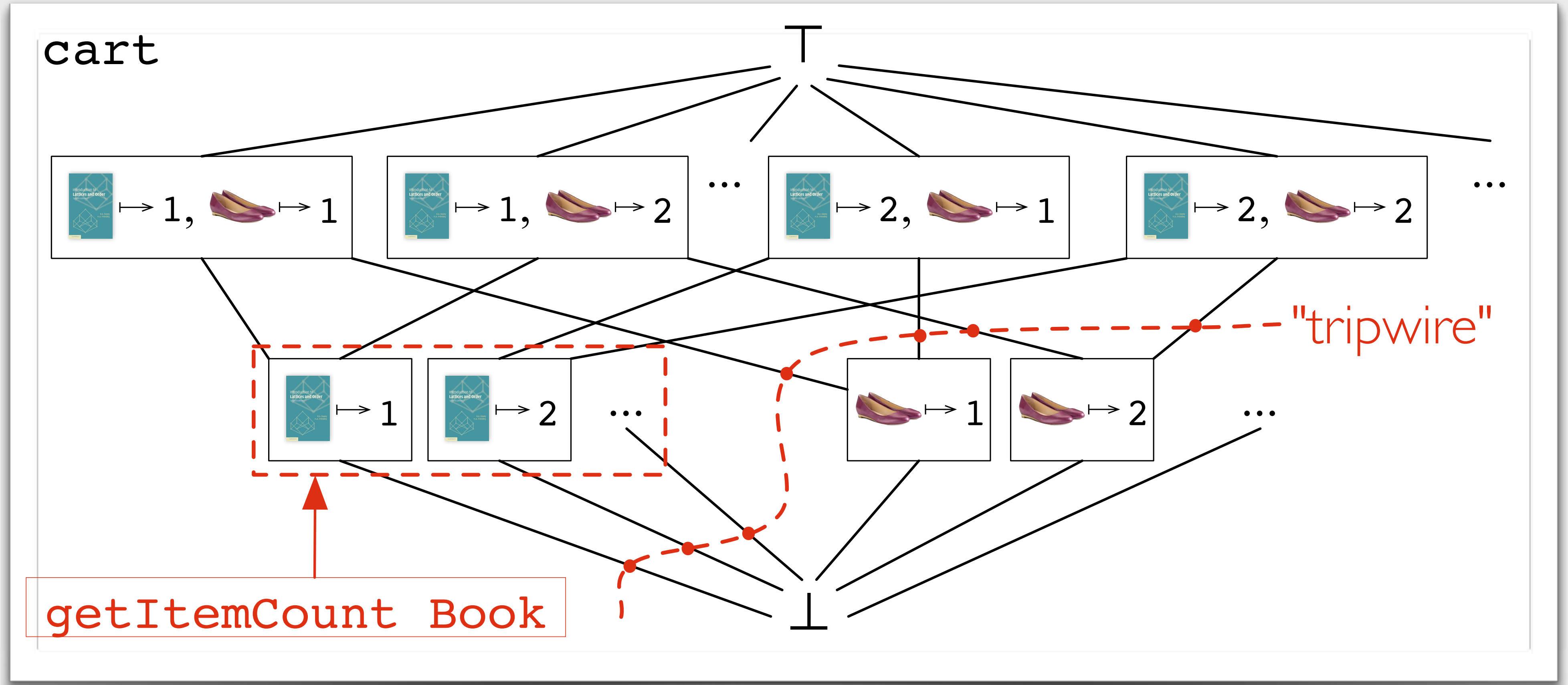
Joining forces Although LVars and CvRDTs were developed independently, both models leverage the mathematical properties of join-semilattices to ensure that a property of the model holds—

Thank you!

Email: lkuper@cs.indiana.edu
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

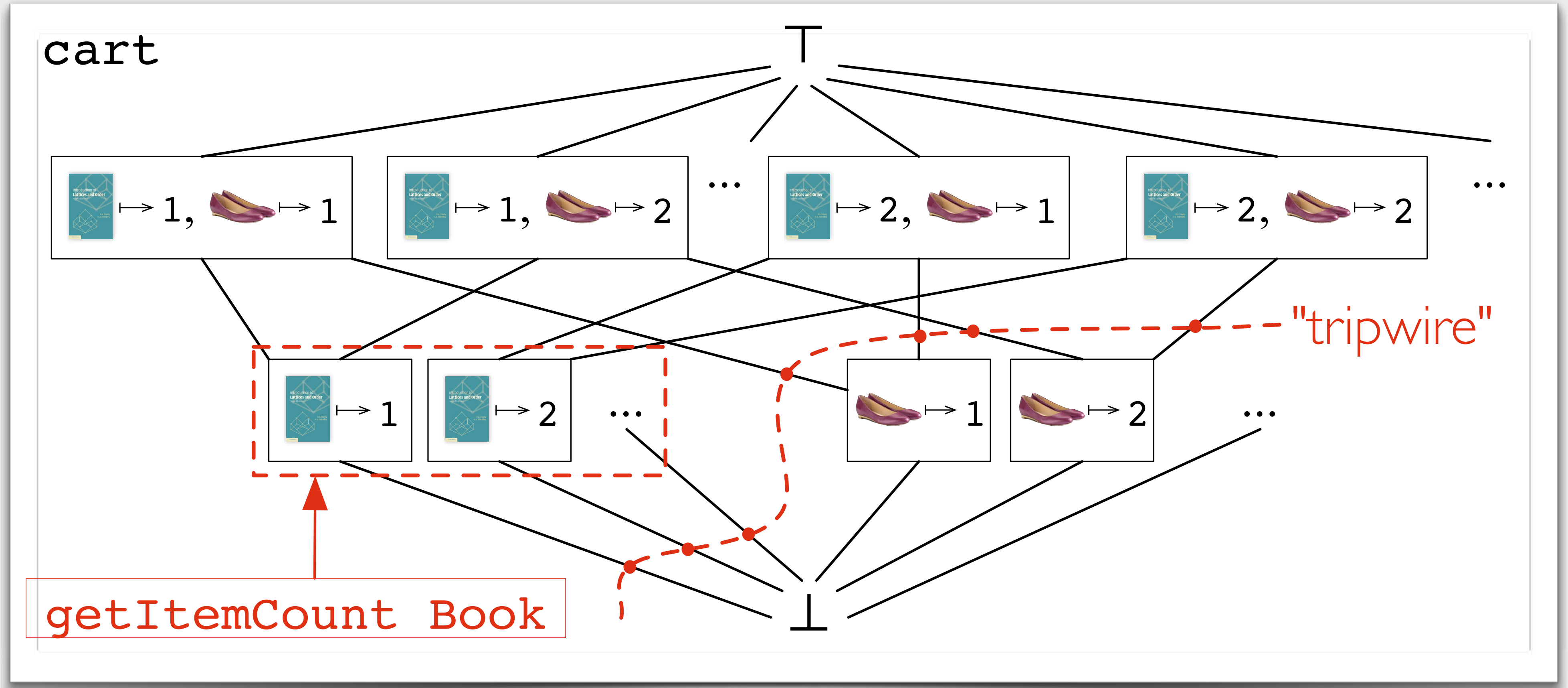


✖ Can't see the exact, complete contents of the cart

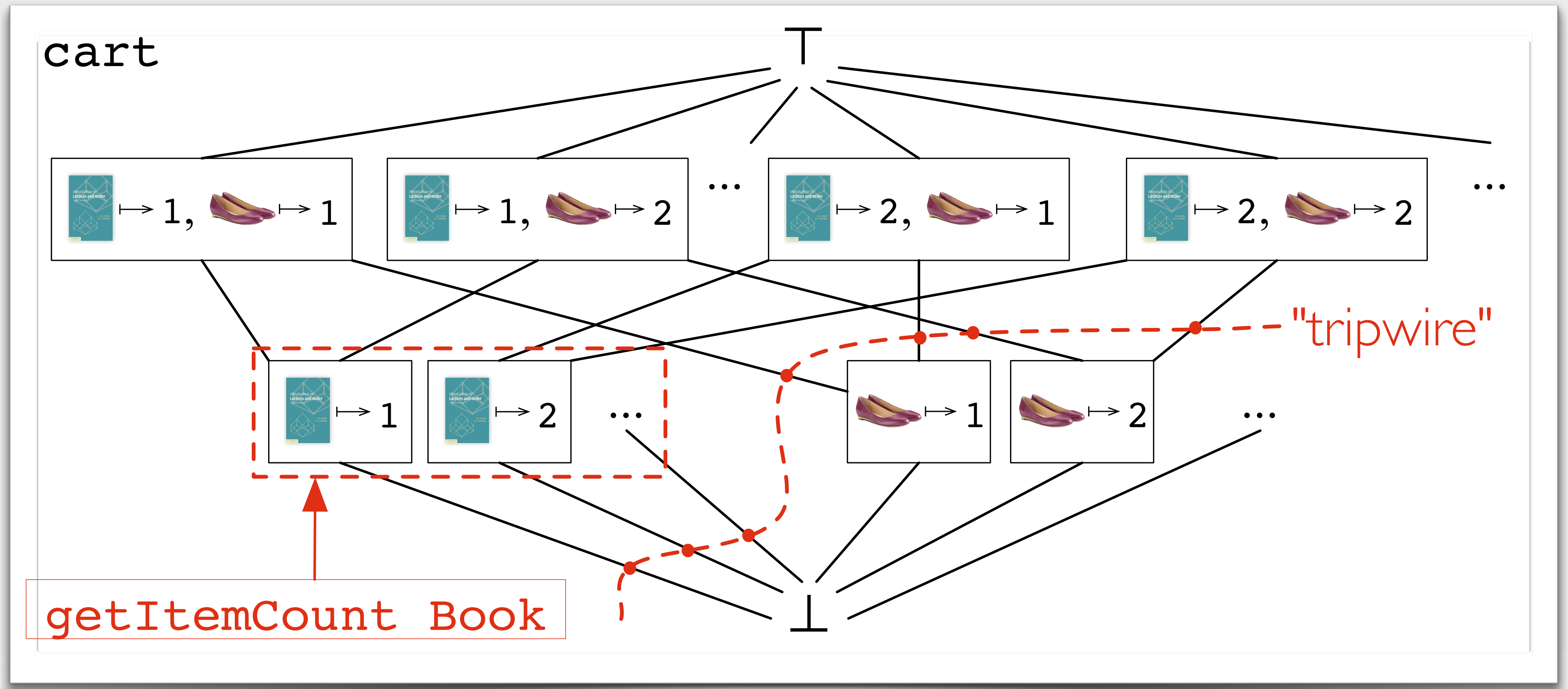


❌ Can't see the exact, complete contents of the cart

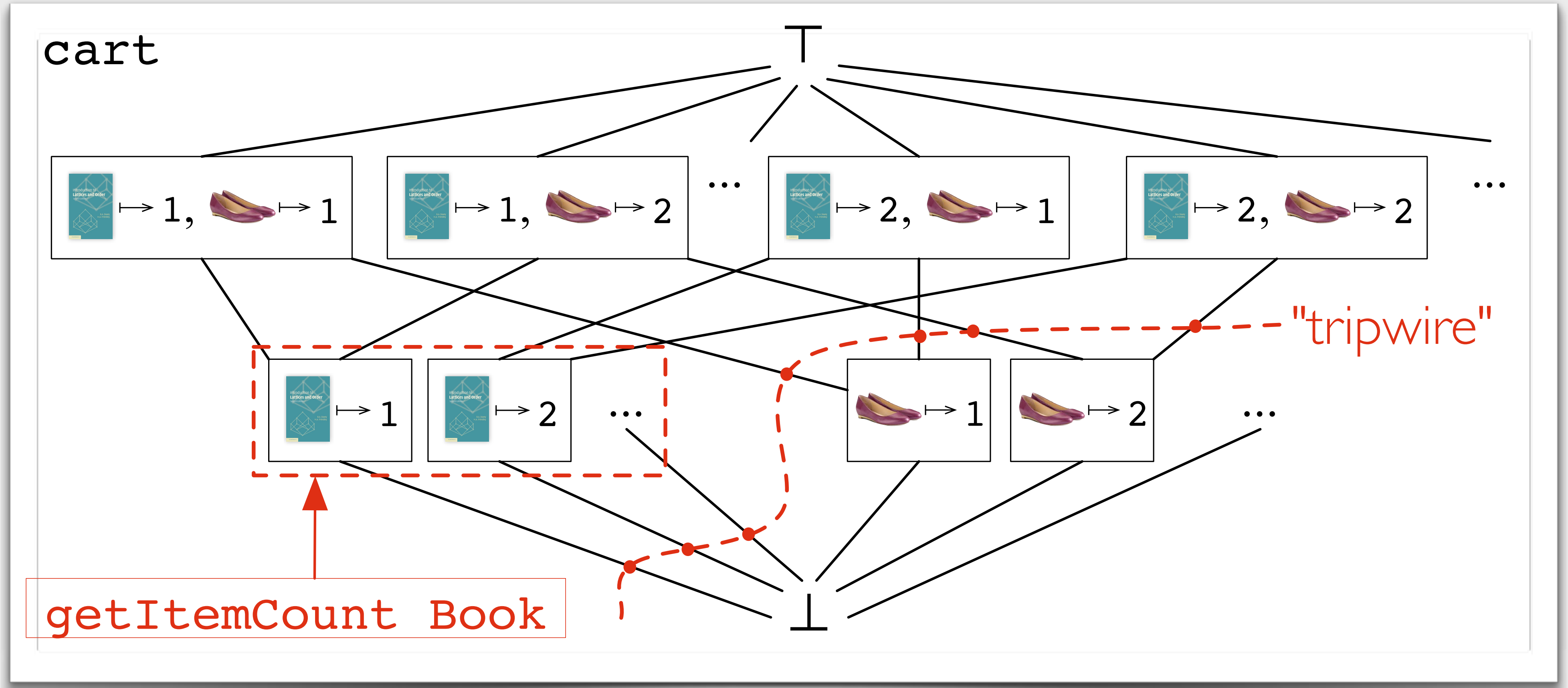
❌ Can't iterate over the items in the cart



- ✖ Can't see the exact, complete contents of the cart
- ✖ Can't iterate over the items in the cart
- ✖ Can't determine if an item *isn't* in the cart



- ✖ Can't see the exact, complete contents of the cart
- ✖ Can't iterate over the items in the cart
- ✖ Can't determine if an item *isn't* in the cart
- ✖ Can't react to writes that we weren't expecting

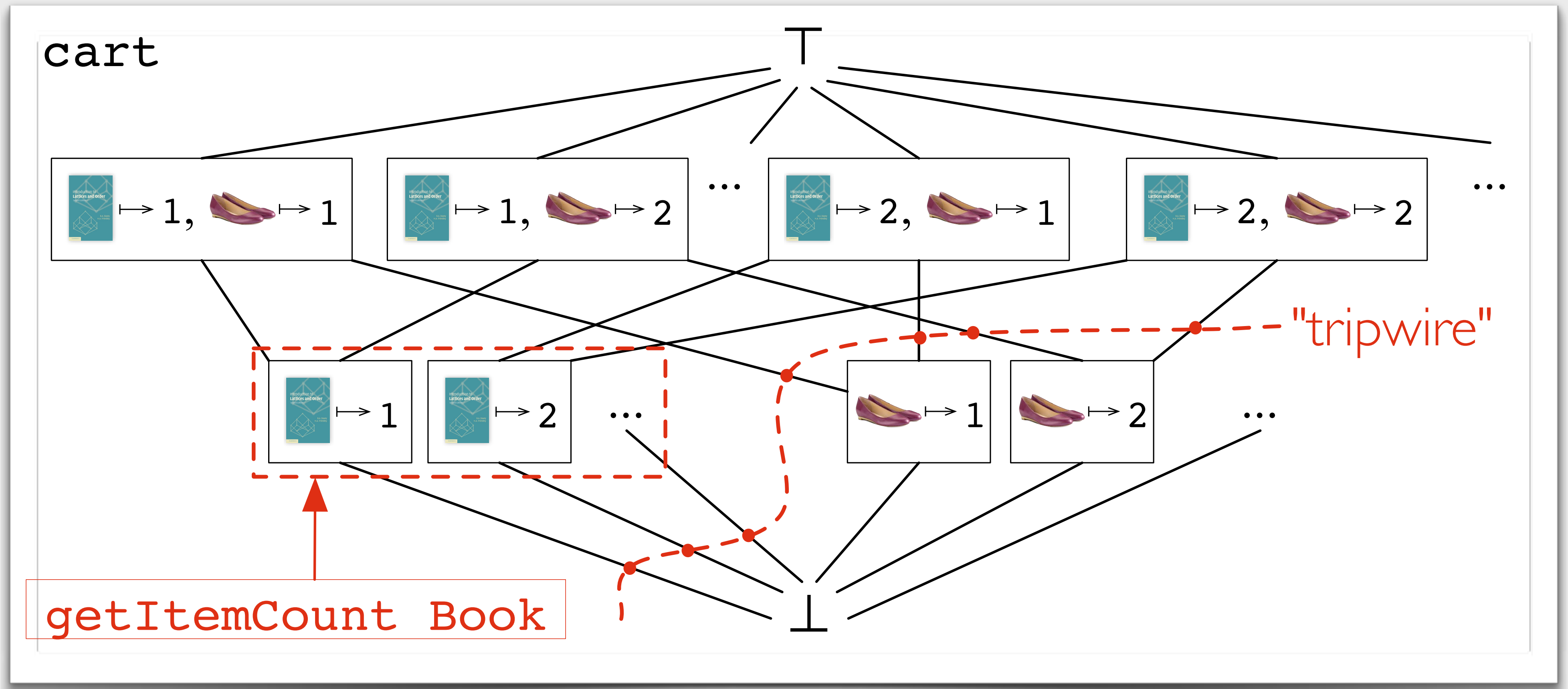


✓ Can see the exact, complete contents of the cart

✓ Can iterate over the items in the cart

✓ Can determine if an item *isn't* in the cart

✓ Can react to writes that we weren't expecting



✓ Can see the exact, complete contents of the cart
✓ Can iterate over the items in the cart
✓ Can determine if an item *isn't* in the cart
✓ Can react to writes that we weren't expecting

} handlers, quiescence, freezing