Utrecht University

B3CC: Concurrency

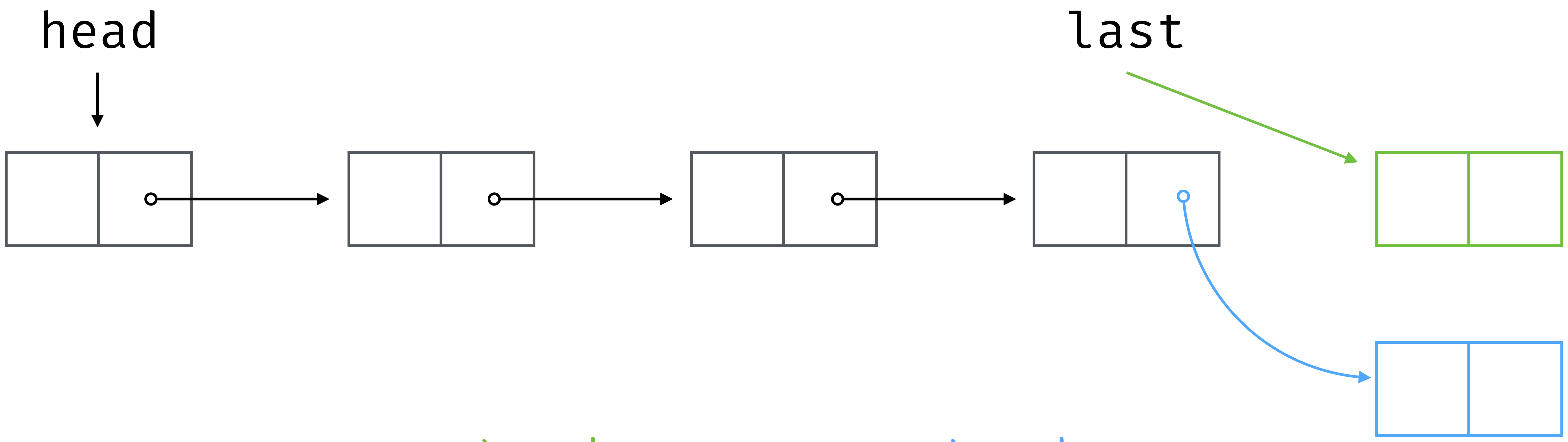
06: Software Transactional Memory (1)

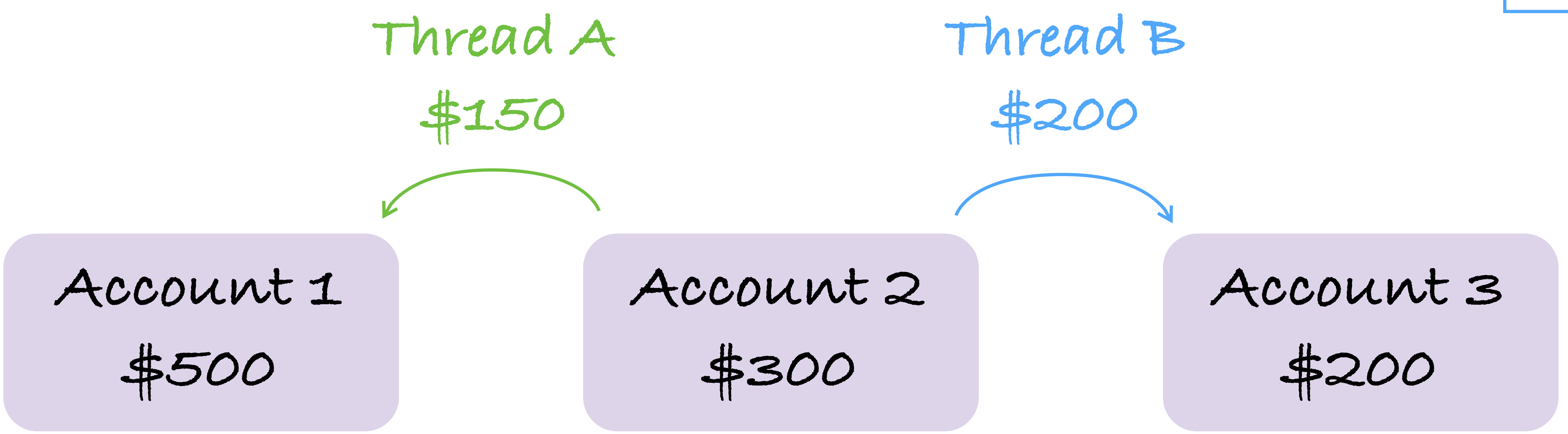
Tom Smeding

Recap

Why?

- Concurrency control required for safe access to shared state between threads
  - Examples we've seen previously:





Attempt #4

- Take locks in an a fixed (but arbitrary) order; release in the opposite order

```
struct Account {
  int balance;
  Mutex lock;
};
```

```
void transfer(int amount, Account *from, Account *to) {
  if (from->accountNumber < to->accountNumber) {
    from->lock.acquireLock();
    to->lock.acquireLock();
    ...
    to->lock.releaseLock();
    from->lock.releaseLock();
  } else {
    to->lock.acquireLock();
    from->lock.acquireLock();
    ...
    from->lock.releaseLock();
    to->lock.releaseLock();
  }
}
```

LAST WEEK

Why?

- Concurrency control
  - *Mutual exclusion:* critical resources => critical section
    - Only one process allowed in the critical section at once
  - *Deadlock*
  - *Starvation*

Review

- What are the requirements for *implementing* mutual exclusion?
- What are the requirements for *using* critical sections?

Review

- Using critical sections
  - Threads should stay in the critical section for as little time as possible
  - What is the consequence of taking locks for too long?

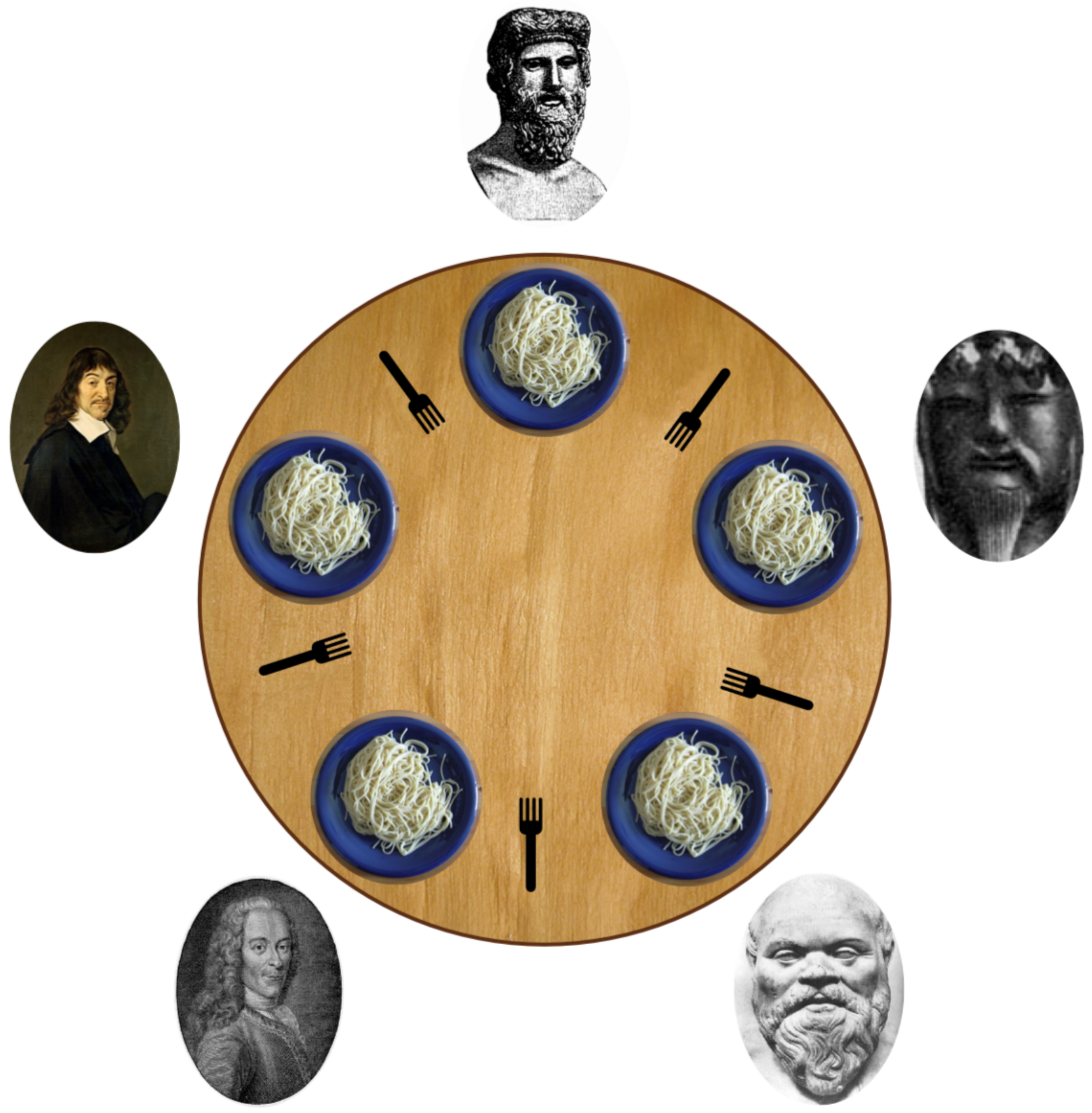
```
countMode :: MVar Int -> [Int] -> IO ()
countMode var accounts =

            sum [ 1 | a <- accounts, mtest a ]
```

🤔

Dining philosophers

- Canonical example of synchronisation issues and how to resolve them
  - Philosophers alternatively think and eat
  - Require both forks to start eating
  - Each fork is held by one philosopher at a time



https://en.wikipedia.org/wiki/Dining_philosophers_problem

Atomic blocks

An alternative

- The idea:
  - Garbage collectors allow us to program without `malloc()` and `free()`
    - Can we do the same for locks?
    - What would that look like?
  - Modular concurrency!
  - Locks are pessimistic; let’s be optimistic instead!

Software transactional memory

- A [programming languages/software-based] technique for implementing *atomic blocks*
  - Atomicity: effects become visible to other threads all at once
  - Isolation: cannot see the effects of other threads
  - Use a different type (STM) to wrap operations *whose effects can be undone if necessary* (more on this later)

```
import Control.Concurrent.STM

data STM a              -- abstract
instance Monad STM      -- among other things

atomically :: STM a -> IO a
```

Software transactional memory

- Sharing state
  - Instead of `IORef`, we use `TVar` as a *transactional variable*
  - Basic interface:

```
import Control.Concurrent.STM.TVar

newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Revisiting accounts

- STM actions are composed together in the same way as IO actions

```
void transfer(int amount, Account *from, Account
  if (from->accountNumber < to->accountNumber) {
    from->lock.acquireLock();
    to->lock.acquireLock();
    ...
    to->lock.releaseLock();
    from->lock.releaseLock();
  } else {
    to->lock.acquireLock();
    from->lock.acquireLock();
    ...
    from->lock.releaseLock();
    to->lock.releaseLock();
  }
}
```

```haskell
type Account = TVar Int

deposit :: Int -> Account -> STM ()
deposit amount account = do
  balance <- readTVar account
  writeTVar amount (balance + amount)

withdraw :: Int -> Account -> STM ()
withdraw amount = deposit (-amount)

transfer :: Int -> Account -> Account -> IO ()
transfer amount from to =
  atomically $ do
    withdraw amount from
    deposit amount to
```

STM

- Types are used to isolate transactional actions from arbitrary IO actions
  - To get from `STM` to `IO` we have to execute the entire action `atomically`
  - Can't mix monads!

```
bad :: Int -> Account -> ?? ()
bad amount account = do
  putStrLn “withdrawing!”
  withdraw amount account

good :: Int -> Account -> IO ()
good amount account = do
  putStrLn “withdrawing!”                -- :: IO ()
  atomically $ withdraw amount account   -- :: IO ()
```

Implementing transactional memory

- How to implement `atomically`
  - Single global lock?
  - Instead: optimistic execution, without taking any locks
- At the start of the atomic block begin a thread local *transaction log*
  - Each `writeTVar` records the address and the new value to the log
  - Each `readTVar` searches the log and
    - Takes the value of an earlier `writeTVar`; or
    - Reads the `TVar` and records the value into the log

Implementing transactional memory

- At the end of the atomic block the transaction log must be *validated*
  - Checks each `readTVar` in the log matches the current value
  - If successful all `writeTVar` recorded in the log are *committed* to the real `TVars`
  - The validate and commit steps together must be truly atomic

Implementing transactional memory

- What if validation fails?
  - The operation executed with an inconsistent view of memory
  - *Re-execute* the transaction with a new transaction log
    - Since none of the writes are committed to memory, this is safe to do
    - It is critical that the atomic block contains no actions other than reads and writes to `TVars`

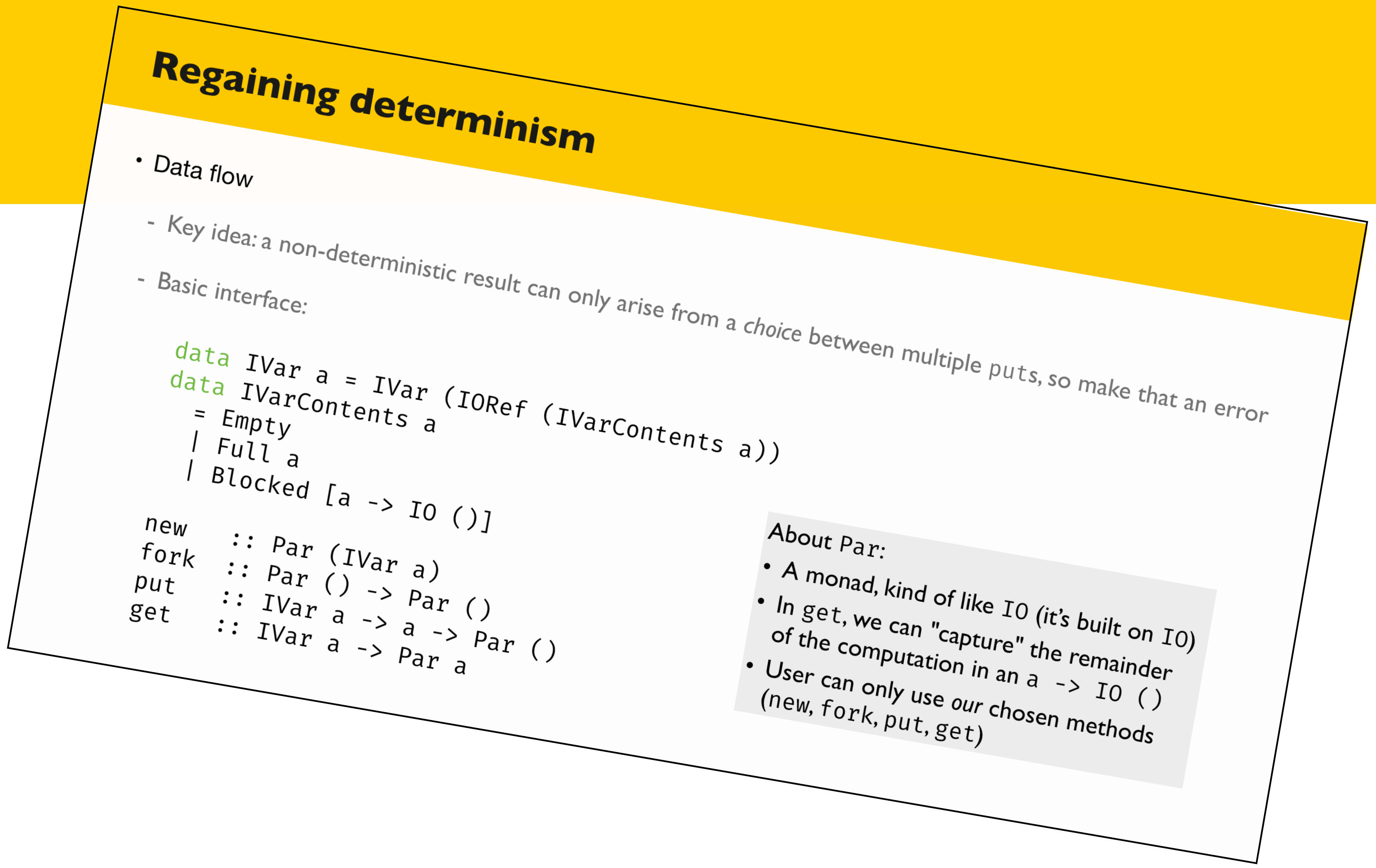
```
atomically $ do
  x <- readTVar xv
  y <- readTVar yv
  if x > y
    then brexit         -- :: IO () side effects!
    else return ()
```

Summary (so far)

- STM gives us:
  - Atomic transactions for shared memory
  - Encapsulation of concurrent code
  - Help avoid common locking problems
- Locks are pessimistic, STM is optimistic
- But…
  - Just like garbage collection, is no silver bullet
  - Can not solve all problems: e.g. starvation & contention

Blocking & Choice

Software transactional memory



- Sharing state
  - Instead of `MVar` we have an equivalent `TMVar`
  - A variable is either *full* or *empty*: threads wait for the appropriate state
  - Basic interface:

```
import Control.Concurrent.STM.TMVar

newTMVar      :: a -> STM (TMVar a)
newEmptyTMVar :: STM (TMVar a)
takeTMVar     :: TMVar a -> STM a
readTMVar     :: TMVar a -> STM a
putTMVar      :: TMVar a -> a -> STM ()
```

Blocking

- Wait for some condition to be true or a resource to become available
  - Abandon the current transaction and begin again
  - Only when the inputs change, to avoid busy waiting (how?)

```
retry :: STM a
```

Accounts, revisited

- Suppose we want to block if the account will be overdrawn
  - Because the transaction read `account` on the way to `retry`, the thread can wait until this variable changes before trying again

```
type Account = TVar Int

withdraw :: Int -> Account -> STM ()
withdraw amount account = do
  balance <- readTVar account
  if amount > 0 && amount > balance
     then retry
     else writeTVar account (balance + amount)
```

Example: TMVar

- Transactional equivalent of `MVar`
  - Shared variable which is either empty or full
  - Easy to implement in terms of TVar!

```
newtype TMVar a = TMVar (TVar (Maybe a))

newEmptyTMVar :: STM (TMVar a)
takeTMVar     :: TMVar a -> STM a
putTMVar      :: TMVar a -> a -> STM ()


newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = do
  t <- newTVar Nothing
  return (TMVar t)
```

TMVar

- Block if the desired variable is empty, and return the contents when it is full

```haskell
takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do
  m <- readTVar t
  case m of
    Nothing -> retry
    Just a  -> do
      writeTVar t Nothing
      return a
```

```haskell
newtype TMVar a = TMVar (TVar (Maybe a))
```

TMVar

- Block when the variable is full, update the contents when it is empty

```haskell
putTMVar :: TMVar a -> a -> STM a
putTMVar (TMVar t) a = do
  m <- readTVar t
  case m of
   Nothing -> writeTVar t (Just a)
   Just _   -> retry
```

```haskell
newtype TMVar a = TMVar (TVar (Maybe a))
```

Question

- Threads block on an `MVar` are woken up in FIFO order
  - This is the fairness guarantee
- When multiple threads are blocked on a `TVar`, which should be woken up?
  - Consider: who can make progress? Example:

```
do x <- takeTVar v
   when (x != 42) retry
```

  - All threads retrying on a variable are woken up

Choice

- Choose an alternative action if the first transaction calls `retry`
  - If the first action returns a result, that is the result of the `orElse`
  - If the first action retries, the second action runs
  - If the second action retries, the whole action retries
  - Since the result of `orElse` is also an `STM` action, you can ``a `orElse` (b `orElse` (c `orElse` …))``

```
orElse :: STM a -> STM a -> STM a
```

Accounts, re-revisited

- Suppose we want to withdraw from a second account if the first has insufficient funds

```
withdraw2 :: Int -> Account -> Account -> STM ()
withdraw2 amount account1 account2 =
  withdraw amount account1
  `orElse`
  withdraw amount account2
```

STM as a building block (1)

Asynchronous computations

Asynchronous computations, revisited

- The goal:
  - Run computations asynchronously and wait for the results
  - Cancel and *race* running computations
  - Interface:

```
data Async a

async  :: IO a -> IO (Async a)
wait   :: Async a -> IO a
poll   :: Async a -> IO (Maybe a)
cancel :: Async a -> IO ()
race   :: Async a -> Async b -> IO (Either a b)
```

async

- Perform an action *asynchronously* and later wait for the results

```
data Async a = Async ThreadId (TMVar a)

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyTMVarIO
  tid <- forkIO $ do
    res <- action
    atomically $ putTMVar var res

  return (Async tid var)
```

wait

- Wait for the computation to complete

```
waitSTM :: Async a -> STM a
waitSTM (Async _ var) = readTMVar var

wait :: Async a -> IO a
wait a =
  atomically $ waitSTM a

race :: Async a -> Async b -> IO (Either a b)
race a b =
  atomically $
    fmap Left  (waitSTM a)
    `orElse`
    fmap Right (waitSTM b)
```

  - Exercise: write an alternative `race` that kills the losing thread

STM as a building block (II)

Concurrent Map

Key-value map

- The goal:
  - A key-value map that can be accessed concurrently by multiple threads
  - Basic interface:

```
data CMap k v

insert :: Ord k => k -> v -> CMap k v -> CMap k v
lookup :: Ord k => k -> CMap k v -> Maybe v
```

Option #1

- A regular (pure) key-value map in a mutable box
  - Simple, safe
  - No concurrency!

```haskell
import Control.Concurrent.MVar
import qualified Data.Map as M

data CMap k v = CMap (MVar (M.Map k v))

insert :: Ord k => k -> v -> CMap k v -> IO ()
lookup :: Ord k => k -> CMap k v -> IO (Maybe v)
```

Option #2

- A pure map in a box, but this time using STM
  - Safe concurrent lookup
  - Insertion updates the entire tree (*all* other threads must retry)

```haskell
import Control.Concurrent.STM
import qualified Data.Map as M

data CMap k v = CMap (TVar (M.Map k v))

insert :: Ord k => k -> v -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
```

Option #3

- A pure map with mutable values
  - Allows values to be read and adjusted (mutated) concurrently
  - Fixed key set

```haskell
import Control.Concurrent.STM
import qualified Data.Map as M

data CMap k v = CMap (M.Map k (TVar v))

adjust :: Ord k => (v -> v) -> k -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
```

Option #4

- Implement the data structure ourselves
  - Goal: Fully concurrent insertion and lookup
  - Updates to disjoint parts of the tree do not conflict with each other

```
data CMap k v = CMap (TVar (Node k v))
data Node k v
  = Bin k (TVar v) (CMap k v) (CMap k v)
  | Tip

insert :: Ord k => k -> v -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
adjust :: Ord k => (v -> v) -> k -> CMap k v -> STM ()
```

Option #4

- Lookup a value in the map
  - Standard recursive traversal
  - Try to implement `insert`!
    - Important! Minimise the number of `writeTVar`!

```haskell
data CMap k v = CMap (TVar (Node k v))
data Node k v
  = Bin k (TVar v) (CMap k v) (CMap k v)
  | Tip
```

```haskell
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
lookup key (CMap ref) = readTVar ref >>= go
  where
    go Tip             = return Nothing
    go (Bin k v l r) =
      case compare key k of
        LT -> lookup key l
        GT -> lookup key r
        EQ -> Just <$> readTVar v
```

Summary

What can we not do with STM?

- STM offers *composable* blocking and atomicity
  - Concurrent programming without locks!
- But, there are also things that it can *not* do compared to using locks
  - Fairness: all blocked threads are woken up when a `TVar` changes
  - Threads can not communicate *that* they are blocking

Performance considerations

- `atomically` works by accumulating a log of `writeTVar` and `readTVar` operations; this has consequences:
  - Discarding the effects of the transaction is easy: delete the log
  - Each `readTVar` must traverse the log to see if it was written by an earlier `writeTVar`: $O(n)$
  - A transaction that called `retry` is woken up whenever one of the `TVar` in its read set changes: $O(n)$
  - A long running transaction can re-execute indefinitely because it is repeatedly aborted by shorter transactions: starvation
- Most abstractions have a runtime cost…

Photo by Jamie Street

Extra slides

- Parallel and Concurrent Programming in Haskell
  Chapter 10: Software Transactional Memory
- STM library
  https://hackage.haskell.org/package/stm