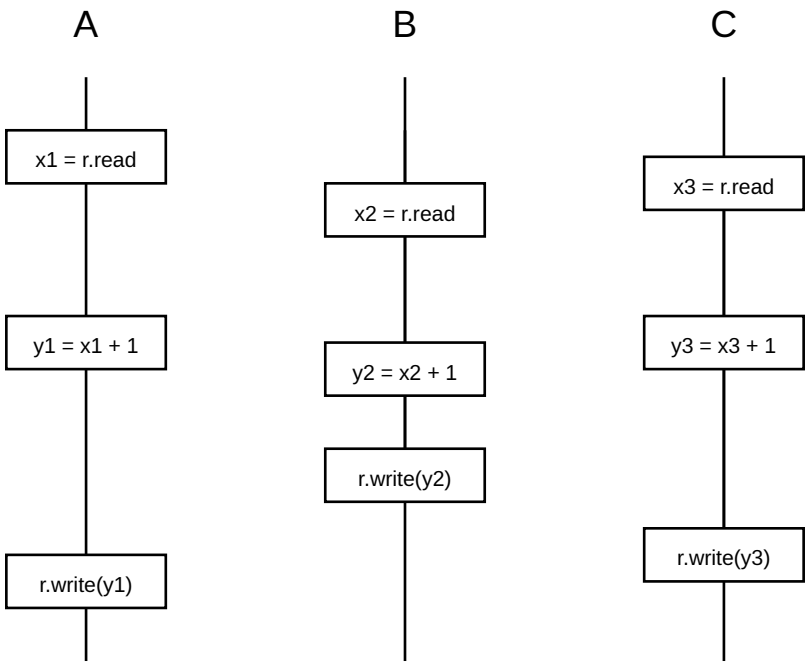
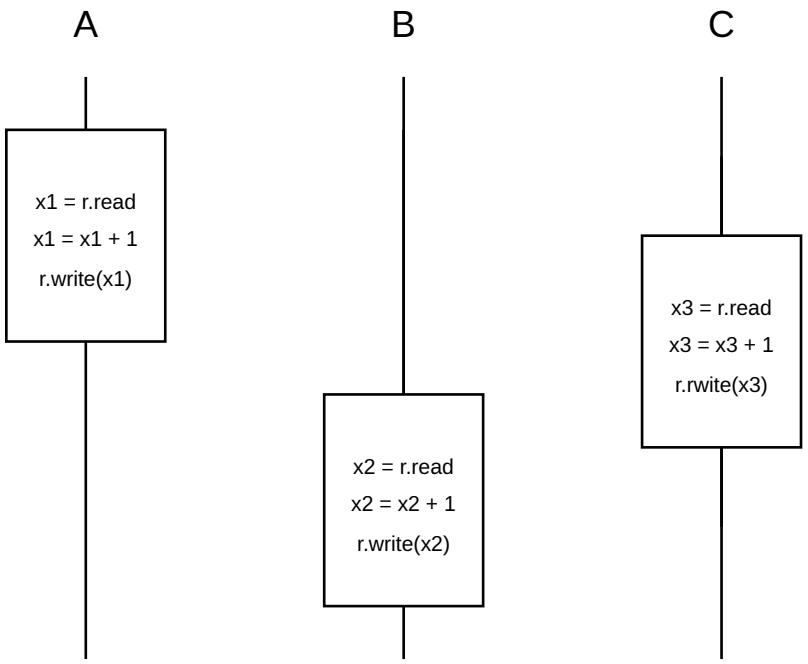
Transactional Memory

Transactions





Alternative to manual locking.

Still non-deterministic, but interleaves only between transactions.

Serializability

> We can broaden the class of allowable executions to include executions that have the same effect as serial ones. Such executions are called serializable. More precisely, an execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions.

Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. Concurrency control and recovery in database systems.

Opacity

> Opacity is a safety property that captures the intuitive requirements that
>
> (1) all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime,
>
> (2) no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and
>
> (3) every transaction always observes a consistent state of the system.

Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory.

Haskell with the Glasgow Haskell Compiler

https://www.haskell.org/

https://www.haskell.org/ghc/

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Glasgow Haskell Compiler by Simon Peyton-Jones, Simon Marlow and others. First appeared 1992.

demo/Basics.hs

Channels in Haskell

https://hackage.haskell.org/package/base/docs/Control-Concurrent-Chan.html

```
newChan :: IO (Chan a)

writeChan :: Chan a -> a -> IO ()

readChan :: Chan a -> IO a
```

demo/Pipeline.hs

MVars in Haskell

https://hackage.haskell.org/package/base/docs/Control-Concurrent-MVar.html

```
newEmptyMVar :: IO (MVar a)

newMVar :: a -> IO (MVar a)

takeMVar :: MVar a -> IO a

putMVar :: MVar a -> a -> IO ()
```

demo/Dining.hs

Select in Haskell

```haskell
race :: MVar a -> MVar b -> IO (Either a b)
race mvar1 mvar2 = do
    result <- newEmptyMVar
    forkIO do
        value1 <- takeMVar mvar1
        tryPutMVar result (Left value1)
        return ()
    forkIO do
        value2 <- takeMVar mvar2
        tryPutMVar result (Right value2)
        return ()
    takeMVar result
```

demo/Timeout.hs

Software Transactional Memory (STM)

> A transaction is a finite sequence of local and shared memory machine instructions:
>
> Read_transactional - reads the value of a shared location into a local register.
>
> Write_transactional – stores the contents of a local register into a shared location.
>
> [...]
>
> Any transaction may either fail, or complete successfully, in which case its changes are visible atomically to other processes.

Nir Shavit and Dan Touitou. 1995. Software transactional memory.

Software Transactional Memory in Haskell

> Concurrent programming is notoriously tricky. Current lock-based abstractions are difficult to use and make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions.

```haskell
-- Transactional variables
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

-- Running STM computations
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- The STM monad itself
data STM a
instance Monad STM
-- Monads support "do" notation and sequencing
```

demo/ReadWrite.hs

demo/Swap.hs

Transactional Data Structures

There are two valid approaches:

1. Immutable data structure in a transactional variable.
2. Mutable data structure using transactional variables.

There is one invalid approach:

3. Mutable data structure behind a transactional variable.

> The API's caller remains responsible for ensuring scalability by making it unlikely that concurrent operations will need to modify overlapping sets of locations. However, this is a performance problem rather than a correctness or liveness one, and in our experience, even straightforward data structures, developed directly from sequential code, offer performance which competes with and often surpasses state-of-the-art lock-based designs.

Keir Fraser and Tim Harris. 2007. Concurrent programming without locks.

Transactional Data Structures in Haskell

1. Immutable data structure in a transactional variable.

```
type Queue a = ([a], [a])
type TQueue a = TVar (Queue a)
```

2. Mutable data structure using transactional variables.

```
data Node a = Nil | Node a (TVar (Node a))
type TQueue a = (TVar (Node a), TVar (Node a))
```

3. Mutable data structure behind a transactional variable.

   Highly discouraged and unlikely in Haskell.

demo/BankersQueue.hs

demo/MutableQueue.hs

Aborting Transactions

Sometimes, we want to abort a transaction based on a user-defined condition.

```
retry :: STM a
```

> Conceptually, retry aborts the transaction with no effect, and restarts it at the beginning. However, there is no point in actually re-executing the transaction until at least one of the TVars read during the attempted transaction is written by another thread.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions.

demo/DiningTransaction.hs

Selecting Transactions

Sometimes, we want to select the first transaction that succeeds.

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

> The transaction s1 ‘orElse‘ s2 first runs s1; if it retries, then s1 is abandoned with no effect, and s2 is run. If s2 retries as well, the entire call retries — but it waits on the variables read by either of the two nested transactions.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions.

demo/TimeoutTransaction.hs

Progress Conditions

> A method is *wait-free* if it guarantees that every call finishes its execution in a finite number of steps.

> A method is *lock-free* if it guarantees that infinitely often some method call finishes in a finite number of steps.

> A method is *obstruction-free* if, from any point after which it executes in isolation, it finishes in a finite number of steps

Maurice Herlihy and Nir Shavit. 2012. The Art of Multiprocessor Programming, Revised Reprint (1st. ed.).

Liveness Properties of Haskell STM

> The STM implementation guarantees that one transaction can force another to abort only when the first one commits. As a result, the STM implementation is lock-free in the sense that it guarantees at any time that some running transaction can successfully commit. [...] Starvation is possible.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions.

demo/Livelock.hs

Data Versioning

In implementations, there are two main approaches to data versioning:

1. Direct Update (eager): keep an undo log.
   Upon commit do nothing, upon abort undo writes.
2. Deferred Update (lazy): keep a write buffer.
   Upon commit perform writes, upon abort do nothing.

In both we keep some auxiliary data structure for each transaction and execute extra code for operations on variables.

Tradeoff: properties, performance, difficulty.

Conflict Detection

In implementations, there are two main approaches to conflict detection:

1. Pessimistic: detect conflicts early, when writing to a variable.
2. Optimistic: detect conflicts late, when attempting to commit.

In both we keep set of variables read and a set of variables written for each transaction.

Tradeoff: properties, performance, difficulty.

Atomic Operations

Processors offer a variety of atomic read-write-modify operations:

Test-And-Set (TAS)

Fetch-And-Add (FAA)

Comapre-And-Swap (CAS)

Load-Linked/Store-Conditional (LL/SC)

These are required for the implementation of locks and transactions.

Beware of the ABA problem with Compare-And-Swap.

Beware of livelocks with Load-Linked/Store-Conditional.

demo/Atomics.hs

Summary and Outlook

Transactional memory data structures and algorithms are composable.

Next week: Actors