1 Atomic Bank Transfer with Locks

We want to atomically transfer funds between accounts. We can synchronize the withdraw method, but this is not enough: we can still be interrupted between `from.withdraw(amount)` and `to.deposit(amount);` at this time, `amount` is in neither account.

One way to solve this is by adding explicit locks:

```
from.lock();
to.lock();
...
from.unlock();
to.unlock();
```

However, this is prone to deadlock. We can avoid this by imposing an ordering on the locks when getting them, and this does work, but it is very complicated.

Common problems with locks:

- **Taking too many locks:** breaks atomicity
- **Taking too few locks:** inhibits concurrency
- **Taking the wrong locks:** the lock-data connection is implicit
- **Taking locks in the wrong order:** hard to avoid deadlocks
- **Error recovery:** hard with exceptions
- **Lost wake-ups:** hard with non-standard control

Locks are anti-modular!

2 Review: Haskell

2.1 Side Effects in Haskell

```
newRef   :: a -> IO (Ref a)
readRef  :: Ref a -> IO a
writeRef :: Ref a -> a -> IO ()

do { r <- newRef 0
   ; incR r
   ; s <- readRef r
   ; print s }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v + 1) }
```

`IO` is a monad, and `Ref` is a mutable object. `Ref`'s side effects are embedded in `IO` (see monad notes).

2.2 Concurrency in Haskell

```
fork::IO a -> IO ThreadId

do { r <- newRef 0
*     ; fork (incR r)
*     ; incR r
      ; s <- readRef r
      ; print s }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v + 1) }
```

There is a data race between the two lines marked with '*'.

3 Atomic Bank Transfer with Haskell STM

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

- **Atomicity:** effects of the embedded action become visible all at once
- **Isolation:** completely unaffected by other threads; as if the embedded action gets a snapshot of the world

We also introduce transaction variables, which are essentially `Ref`s in the `STM` monad instead of in `IO`:

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

We could then implement `transfer` in this way:

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount
  = do { bal <- readTVar acc
       ; writeTVar acc (bal - amount) }

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

transfer :: Account -> Account -> Int -> STM ()
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount
  = atomically (do { deposit  to   amount
                   ; withdraw from amount })
```

4 More about Haskell STM

Transactions might be aborted or re-tried, so they must not have side-effects (outside of `STM` monad/inside of `IO` monad), such as shown below:

```
atomic $ if x > y then launchMissiles
```

However, the type system prevents this kind of error:

```
bad :: Account -> IO ()
bad acc = do { hPutStr stdout "Withdrawing..."
             ; withdraw acc 10 }
```

The `hPutStr` conflicts with the `withdraw` since they have effects in different monads `IO` and `STM`, respectively; it is therefore a type error to compose them. We can fix this by converting the `STM` monad to an `IO` monad:

```
good :: Account -> IO ()
good acc = do { hPutStr stdout "Withdrawing..."
              ; atomically (withdraw acc 10) }
```

However, it is important to note that `hPutStr` and `withdraw` are not executed atomically here: we can get interrupted after printing but before actually withdrawing.

How do we debug when we can't print everywhere? We can use `UnsafePerformIO :: IO a -> a` to unsafely pass typechecking to print or launch missiles.

We use `retry :: STM a` to abort and retry the current STM. This is useful if we e.g. need to wait for an event to occur before we can proceed with the transaction. We can then add a check on the current balance to `withdraw` to prevent overdraft:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; if amount > 0 && amount > bal
         then retry
         else writeTVar acc (bal - amount) }
```

We introduce a new function `check` to generalize this:

```
check :: Bool -> STM ()
check True  = return ()
check False = retry
```

`limitedWithdrawn` can be rewritten to use `check`:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; check (amount <= 0 || amount <= bal)
       ; writeTVar acc (bal - amount) }
```

We introduce a new function `orElse :: STM a -> STM a -> STM a`, which tries two choices biased towards the first. Here is an example, `limitedWithdraw2`, which attempts to withdraw from one of two accounts:

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

The `STM` monad has built-in support for exceptions:

```
throw :: Exception -> STM a
catch :: STM a -> (Exception -> STM a) -> STM a
```

We can also check invariants using `always :: STM Bool -> STM ()`. Here is an example that prevents an account's balance from ever becoming negative:

```
checkBal   :: TVar Int -> STM Bool
checkBal v = do cts <- readTVar v
                return (v >= 0)

newAccount :: STM (TVar Int)
newAccount = do v <- newTVar 0
                always $ checkBal v
                return v
```

Any transactions that modify the account will always check the invariant. The invariant can be violated *during* a transaction, so long as it is satisfied before and after.

5 Optimistic Execution

Here is one model of optimistic execution:

1. No locks needed to execute an action
2. Record `writeTVar` in log
3. Record `readTVar` if not written by this action
4. On completion of action, validate the log; i.e., check that each `readTVar` matches the actual value
5. If validation succeeds, commit
6. Otherwise, abort and retry

More sophisticated implementations are also possible.

How well does this perform? As the number of threads increases, STM's time spent per operation grows much more slowly using optimistic execution than using coarse- or fine-grained locking.

However, there is a problem with optimistic execution: what if a transaction fails over and over? To address this, we should perform early checks for conflicts so that we can abort early.

6 Java STM

An ordinary `Object` is wrapped within a `TMObject` to become a transactional object; thus, `TMObject` is the Java analogue of `TVar`. `TMObject`'s methods require the wrapped `Object` to be cloneable.

`TMObjects` have explicit read/write modes. It is possible to explicitly release read access to a `TMObject` using its `release()` method, but this is unsafe: other transactions are now able to modify that `TMObject`, yet the current transaction has not yet committed. It is up to the programmer to ensure safety in this situation. This unsafeness is reminiscent of the lock idiom. Can we check safeness of release during log validation?

Unlike `TVars`, any access to a `TMObject` can throw a `Denied` exception. This allows us to abort earlier, but seems to be exposing low-level mechanisms.

Each `TMObject` contains `Locator` objects; these represent its state. Each `Locator` object contains the old and new object values, as well as the status of the current transaction. Commits and aborts are performed by atomically swapping locators using pointer swapping.

Java STM supports plugging in any implementation of a "contention manager", which is in charge of transaction aborting strategy; i.e., whether to keep waiting or give up and abort a transaction.