A different way to do concurrency

Haskell's STM monad

Elisabeth Stenholm, PhD

Software developer – Frende forsikring

Concurrency, am I right?

Let's look at a simple banking example...

```
class Account
{
  private int _balance = 0;
  private readonly object _lockObject = new object();

  public void Withdraw(int amount)
  {
    lock (_lockObject)
    {
      _balance -= amount;
    }
  }

  public void Deposit(int amount)
  {
    Withdraw(-amount);
```

```
{
  private int _balance = 0;
  private readonly object _lockObject = new object();

  public void Withdraw(int amount)
  {
    lock (_lockObject)
    {
      _balance -= amount;
    }
  }

  public void Deposit(int amount)
  {
    Withdraw(-amount);
```

```
    private int _balance = 0;
    private readonly object _lockObject = new object();

    public void Withdraw(int amount)
    {
      lock (_lockObject)
      {
        _balance -= amount;
      }
    }

    public void Deposit(int amount)
    {
      Withdraw(-amount);
    }
}
```

```
25 static void Transfer(Account fr, Account to, int amount)
26 {
27   fr.Withdraw(amount);
28   to.Deposit(amount);
29 }
```

```
1 class Account
2 {
3   private int _balance = 0;
4   public readonly object LockObject = new object();
5
6   ...
7 }
```

```
25 static void Transfer(Account fr, Account to, int amount)
26 {
27   lock (fr.LockObject)
28   {
29     lock (to.LockObject)
30     {
31       fr.Withdraw(amount);
32       to.Deposit(amount);
33     }
34   }
35 }
```

What do we actually want?

What do we actually want?

Atomic execution

What do we actually want?

Atomic execution that composes,

What do we actually want?

Atomic execution that composes, blocks

What do we actually want?

Atomic execution that composes, blocks and has choice.

This is Haskell's STM monad!

Monad?

Monad?

```
greet :: String -> IO ()
greet name = putStrLn $ "Hi, " ++ name
```

Monad?

```haskell
greet :: IO ()
greet = do
  name <- getLine -- :: IO String
  putStrLn $ "Hi, " ++ name
```

Haskell's STM monad

Software Transactional Memory

Software Transactional Memory

```haskell
data STM a = ...
```

Software Transactional Memory

```
data STM a = ...
data TVar a = ...
```

Software Transactional Memory

```haskell
data STM a = ...
data TVar a = ...
readTVar :: TVar a -> STM a
```

Software Transactional Memory

```haskell
data STM a = ...
data TVar a = ...
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Software Transactional Memory

```haskell
data STM a = ...
data TVar a = ...
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
atomically :: STM a -> IO a
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = withdraw account (- amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = withdraw account (- amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = withdraw account (- amount)

transfer :: TVar Integer -> TVar Integer -> Integer -> STM ()
transfer from to amount = do
  withdraw from amount
  deposit to amount
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = withdraw account (- amount)

transfer :: TVar Integer -> TVar Integer -> Integer -> STM ()
transfer from to amount = atomically $ do
  withdraw from amount
  deposit to amount
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = withdraw account (- amount)

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

We can compose! 

What about blocking?

What about blocking?

```
retry :: STM a
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)
```

```haskell
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  if balance < amount
    then retry
    else writeTVar account (balance - amount)
```

We can block! 

What about choice?

What about choice?

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

```haskell
transfer :: TVar Integer -> TVar Integer -> Integer -> STM ()
transfer from to amount = do
  withdraw from amount
  deposit to amount
```

```haskell
transfer :: TVar Integer -> TVar Integer -> Integer -> STM ()
transfer from to amount = do
  withdraw from amount
  deposit to amount

transferWithBackup :: TVar Integer -> TVar Integer
                   -> TVar Integer -> Integer -> STM ()
transferWithBackup from fromBackup to amount =
  (transfer from to amount)
    `orElse` (transfer fromBackup to amount)
```

We have choice! 

But how does it work?

Pretending!

Pretending!

Memory

Pretending!

from read: 500 | write: 400

Memory

Pretending!

from read: 500 | write: 400

to read: 200 | write: 300

Memory

Pretending!

from read: 500 | write: 400

to read: 200 | write: 300

Memory

Pretending!

from read: 500 | write: 400

to read: 200 | write: 300

Memory

Pretending!

Memory

Pretending!

from read: 500 | write: 400

Memory

Pretending!

from read: 500 | write: 400

to read: 150 | write: 250

Memory

Pretending!

from read: 500 | write: 400

to read: 150 | write: 250

Memory

Pretending!

from read: 500 | write: 400

to read: 150 | write: 250

Memory

No more pretending!

Memory

No more deadlocks 

No more deadlocks 🔒

Can find indefinitely blocked threads 🧵

No more deadlocks 🔒

Can find indefinitely blocked threads 🧵

But can get resource starvation 🍽️

STM ❤️ Haskell

Thanks for listening!