Software Transactional Memory

Tuesday, November 01, 2011
13:33

Account Transfer

→ multi-threaded transfer could lose money in worst case

→ Locking can cause deadlock

  → fix by ordering locks, but this gets bad quickly

Locks are bad

→ Common problems

  → too few locks – breaks atomicity

  → too many locks – serializes code (or deadlocks)

  → wrong locks – lock-data relationship is implicit

  → wrong order – deadlock!

  → error recovery – hard w/ exceptions

  → lost wake ups – hard w/ non-standard control

⇒ Locks are anti-modular

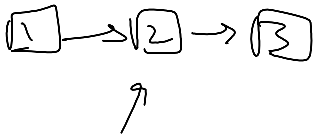
Software Transactional Memory

How does Haskell handle side effects?

main function → monad IO of unit

Functional Languages

→ values are immutable!



can I mutate! must replace!

can't mutate! must replace!

refcell → mutable value container implemented as IO monad

```
newRef :: a → IO (Ref a)
readRef :: Ref a → IO a
writeRef :: Ref a → a → IO ()
fork :: IO a → IO ThreadId
```

```
do { r ← newRef 0
   ; incR r
   ; s ← readRef r
   ; print s
   }
```

Data Race!

---

atomically :: STM a → IO a

has

→ Atomicity

→ isolation

---

transactions need to be re-tried!

⇒ transactions shouldn't have side effects

hPutStr = printLine

---

retry is pretty cool

orElse computation ⟺ same as retry computation!?

Optimistic Execution

→ no locks needed

→ Record write set in log