Principles of Concurrency

Lecture 19
STM Haskell
An extension of Haskell with support for software transactions

Addresses significant deficiencies in prior approaches:

- safety in the presence of interaction with other threads
- handling blocking operations
- composition: explicit support for retry and non-blocking select
Consider a hash-table with thread-safe insert and delete operations
  - Remove an element A from table T1 and insert into table T2
  - Difficult to realize using locks
    - Hard to do using software transactions if the computation should block waiting for A to be visible in T1
    - **Example:** Item get() { atomic (n_items > 0) { ... remove ...}}
    - How do we atomically remove two items?

Consider procedure p1 that waits for data on two pipes and p2 that does the same thing (on different pipes). How can we select between p1 and p2?
  - What is the semantics of: atomic { p1 ‘orElse’ p2 }
Concurrent Haskell

- Extension to Haskell, a lazy purely functional programming language
- Support for explicitly-forked threads, and shared-memory communication
- Model communication effects using monads

Intuition: a value of type $\mathcal{M} \, t$ is a computation which when run may perform $\mathcal{M}$-actions before returning a value of type $t$.

- think of a monad as a design pattern that “amplifies” its underlying type, providing some additional computational or representational expressiveness
- alternatively, it represents a way of building workflows without exposing cross-cutting concerns
- Strong encapsulation of its actions/effects
**IO actions and mutable state**

```haskell
putChar :: Char -> IO ()
getChar :: IO Char

main = do { c <- getChar; putChar c; putChar c }

newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()

forkIO :: IO a -> IO ThreadId

main = do { forkIO (print 'x'); print 'y' }```

---

This page discusses the use of `IO` actions and mutable state in Haskell. The `putChar` and `getChar` functions are shown, along with the `main` function that illustrates their usage. The `newIORef`, `readIORef`, `writeIORef`, and `forkIO` functions are also introduced, providing examples of handling mutable state and concurrent operations in Haskell.
The STM Monad

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

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

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

-- Transactional variables
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
Composable Transactions

Implement a resource manager that holds an integer-valued resource

- `getR r n`: acquire `n` units of resource `r`, blocking if `r` has an insufficient quantity
- `putR r n`: deposit `n` units of resource `r`

```haskell
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
  ; writeTVar r (v+i) }
```

To commit an STM action:

```haskell
atomic :: STM a -> IO a
main = do { ...; atomic (putR r 3); ... }
```
STM Actions

Only STM actions and pure computations can be performed inside a memory transaction. IO operations cannot.

Conversely, no STM actions can be performed outside a transaction: e.g., can’t read or write a TVar unless it’s in the dynamic scope of an atomic.

getR :: Resource -> Int -> STM ()
getR r i = do { v <- readTVar r
  ; if (v < i) then retry
    else writeTVar r (v-i) }

Blocking and retry: retried transaction executes only when a read TVar is updated
Composition

atomic (do { getR r1 3; getR r2 7 })

- blocks if either r1 or r2 have insufficient resource
- no opportunity for deadlock
- caller oblivious to implementation of getR
- making getR an STM, rather than IO, action enables compositionality
Composing Alternatives

```
atomic (getR r1 3 'orElse' getR r2 7)
```

```
nonBlockGetR :: Resource -> Int -> STM Bool
nonBlockGetR r i = do { getR r i ; return True }
    'orElse' return False

blockGetR :: Resource -> Int -> STM ()
blockGetR r i =
do { s <- nonBlockGetR r i;
    if s then return () else retry }
```

Laws

```
M1 'orElse' (M2 'orElse' M3)
    = (M1 'orElse' M2) 'orElse' M3
retry 'orElse' M = M
M 'orElse' retry = M
```
MVars

```haskell
type MVar a = TVar (Maybe a)
newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing

takeMVar :: MVar a -> STM a
takeMVar mv
  = do { v <- readTVar mv
        ; case v of
          Nothing -> retry
          Just val -> do { writeTVar mv Nothing
                           ; return val } }

putMVar :: MVar a -> a -> STM ()
putMVar mv val
  = do { v <- readTVar mv
        ; case v of
          Nothing -> writeTVar mv (Just val)
          Just val -> retry }
```

communication channel with single buffered item
Multicast channels

data MChan a
data Port a
newMChan :: STM (MChan a)
-- Write an item to the channel:
writeMChan :: MChan a -> a -> STM ()
-- Create a new read port:
newPort :: MChan a -> STM (Port a)
-- Read the next buffered item:
readPort :: Port a -> STM a

A buffer is represented as a linked list of items

type Chain a = TVar (Item a)
data Item a = Empty | Full a (Chain a)

An MChan points to the “write” end of a chain; a Port points to the “read” end

type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)

type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)

newMChan = do { c <- newTVar Empty; newTVar c }
newPort mc = do { c <- readTVar mc; newTVar c }

readPort p
  = do { c <- readTVar p
    ; i <- readTVar c
    ; case i of
      Empty    -> retry
      Full v c' -> do { writeTVar p c';
                      return v } }

writeMChan mc v
  = do { c <- readTVar mc
    ; c' <- newTVar Empty
    ; writeTVar c (Full v c')
    ; writeTVar mc c' }
Summary

- STM Haskell provides optimistic software transactions in Haskell
- Supports compositional construction of transactions in the presence of blocking and retry actions
- Monadic structure separates transactional effects (e.g., actions over logs) from pure computation