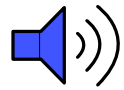
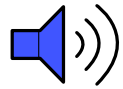
# Lecture 15: Software Transactional Memory



Outline:

- Review of find file function
- Software transactional memory (STM)

# Find files in a directory 

- Traverse a file directory and its sub-directories with a state monad, which decides whether the traversal should be *Done*, *Continue*, or *Skip* a sub-directory.

```
1 data FileAction = Done | Skip | Continue
```

- The result of the traversal is accumulated in State Monad – `FileState s`, where *s* is the state type and contains data that we are interested in.

```
1 type FileState s =  StateT s IO FileAction
```

- The value of `FileState s` is `FileAction`, which tells the `find` function what to do based on the current file and accumulated value (i.e. the state 's').

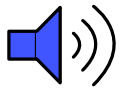
# Find files in a directory

- The *find* function first lists the file names in the top-level directory of the given *path* and then calls a local function `iterate` to visit each file name.

```
find :: forall s.                        -- bind 's' for 'iterate'
        (FileInfo -> FileState s)        -- getState function
     -> FilePath                         -- top-level directory
     -> FileState s                      -- final file state

find getState path = do
  names <- lift $ listDirectory path --lift IO to StateT s IO
  iterate names -- recursively visit each file or directory

  where iterate :: [FilePath] -> FileState s
```

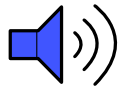
- The *find* function returns a *StateT s IO* monad. So if we want to extract value from an *IO* monad, we have to first 'lift' it using the *lift* function.

```
lift :: (MonadTrans t, Monad m) => m s -> t m s
```

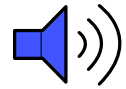
# Find files in a directory 

- *iterate* returns *Continue* action when it reaches the end of the list so that *find* function will not stop just because it has searched a sub-directory.

```
1   iterate :: [FilePath] -> FileState s
2
3   iterate [] = return Continue -- continue if finished
4                                -- searching a directory
5   iterate (name:rest) = do
6     let path' = path </> name      -- full path to file
7     info <- lift $ getInfo path'   -- get file info
8     action <- getState info        -- run file state
9                                    -- to get file action
10    case action of
11      Done -> return Done          -- pass Done to caller
12      Skip -> iterate rest         -- don't recurse
13      _ -> if runFileP searchP info   -- is a directory
14             then do -- recursively visit sub-directory
15                     action' <- find getState path'
16                     case action' of
17                       Done -> return Done -- done
18                       _ -> iterate rest   -- continue
19             else iterate rest    --- visit other files
```

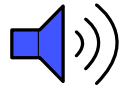
# Software Transactional Memory 

- Errors in concurrent programs are often caused by race conditions, where multiple threads have interleaving access to shared variables.
- To prevent unsafe access to shared variables, concurrent programs use locks; but lock-based programs do not compose, and they may cause deadlock and priority inversion,
- STM helps manage internal concurrency between the threads interacting through memory in a single process.
  - A transaction is executed *atomically* – all or nothing.
  - Only revocable operations (memory access) can be in a *STM*.
  - Blocking is composable, where a transaction can block at *retry* until it commits.
  - Can compose alternative transactions using *orElse*.

# Software Transactional Memory 

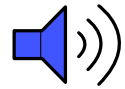
- STM is defined as a monad that wraps computation that involves transactional variable TVar.
- TVars are defined and accessed in STM (except newTVarIO).
- STM is evaluated *atomically* to IO.

```
1  -- STM monad
2  data STM a
3  instance Monad STM
4
5  -- running STM computations
6  atomically :: STM a -> IO a             -- commits STM a
7  retry      :: STM a          -- retry if anything changes
8  orElse     :: STM a -> STM a -> STM a -- choice of two
9
10 -- transactional variables
11 data TVar a
12 newTVar    :: a -> STM (TVar a)
13 readTVar   :: TVar a -> STM a
14 writeTVar  :: TVar a -> a -> STM ()
```

# Software Transactional Memory 

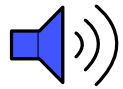
- When a STM is evaluated, the TVars it accessed are logged
  - when it commits, the logged TVars are checked for atomicity violation (whether any TVar is mutated by another thread).
  - If no atomicity violation, then STM commits TVar effects and returns a value.
  - If atomicity violation is detected, all TVar effects are aborted and STM waits for retry (after any TVar is changed).

```
1  -- STM monad
2  data STM a
3  instance Monad STM
4
5  -- running STM computations
6  atomically :: STM a -> IO a
7  retry      :: STM a
8  orElse     :: STM a -> STM a -> STM a
9
10 -- transactional variables
11 data TVar a
12 newTVar    :: a -> STM (TVar a)
13 readTVar   :: TVar a -> STM a
14 writeTVar  :: TVar a -> a -> STM ()
```

# Software Transactional Memory 

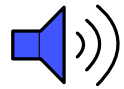
- STM execute read/write operations on TVars atomically.
- When invoked atomically, the STM checks that the logged accesses are valid – no concurrent transaction has committed conflicting updates.

```
1  readTVar  :: TVar a -> STM a
2  writeTVar :: TVar a -> a -> STM ()
3  atomically :: STM a -> IO a
4
5  -- read/write resource
6  type Resource = TVar Int
7
8  putR :: Resource -> Int -> STM ()   -- increment r by i
9  putR r i = do v <- readTVar r       -- read value of r
10               writeTVar r (v + i)   -- write r by v + i
11
12 main = do { ...; atomically (putR r 3); ...}
```

# Software Transactional Memory 

- retry aborts a transaction and restarts from the begining after some logged TVar is changed.

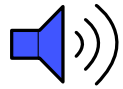
```
1  readTVar  :: TVar a -> STM a
2  writeTVar :: TVar a -> a -> STM ()
3  atomically :: STM a -> IO a
4
5  -- read/write resource
6  type Resource = TVar Int
7
8  getR :: Resource -> Int -> STM ()
9  getR r i = do v <- readTVar r
10               if (v < i) then retry    -- abort and retry
11               else writeTVar r (v-i)
12
13 main = do { ...; atomically (getR r 3); ...}
```

# Software Transactional Memory 

- Sequential composition of STMs is an STM.

```
1 -- read/write resource
2 type Resource = TVar Int
3
4 getR :: Resource -> Int -> STM ()
5 getR r i = do v <- readTVar r
6               if (v < i) then retry    -- abort and retry
7               else writeTVar r (v-i)
8
9 main = atomically $ do { getR r1 3; getR r2 7 }
```

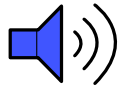
- main blocks if either r1 or r2 lacks resource but there won't a chance of deadlock.

# Software Transactional Memory 

- Composition of alternative STMs is an STM.

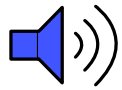
```
1 -- read/write resource
2 type Resource = TVar Int
3
4 getR :: Resource -> Int -> STM ()
5 getR r i = do v <- readTVar r
6               if (v < i) then retry    -- abort and retry
7               else writeTVar r (v-i)
8
9 main = atomically $ do { getR r1 3 `orElse` getR r2 7 }
```

- if *getR r1 3* retries (and aborts), then *getR r2 7* is run; and if the latter also retries, then the entire call retries.

# Software Transactional Memory 

- Use alternative composition to change blocking behavior

```
1  -- read/write resource
2  type Resource = TVar Int
3
4  getR :: Resource -> Int -> STM ()
5  getR r i = do v <- readTVar r
6                if (v < i) then retry    -- abort and retry
7                else writeTVar r (v-i)
8
9  nonblockGetR :: Resource -> Int -> STM Bool
10 nonblockGetR r i = do { getR r i; return True }
11                       `orElse` return False
12
13 blockGetR :: Resource -> Int -> STM ()
14 blockGetR r i = do s <- nonBlockGetR r i
15                    if s then return () else retry
```

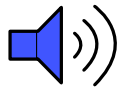
# Software Transactional Memory 

- Composition with *orElse* is associative and has unit *retry*.

```
s1 `orElse` (s2 `orElse s3) = (s1 `orElse` s2) `orElse` s3

retry `orElse` s = s

s `orElse` retry = s
```

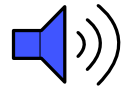
- STM is an instance of MonadPlus

```
instance MonadPlus STM where
  mplus = orElse
  mzero = retry
```

# Software Transactional Memory 

- Uncaught exception aborts the transaction with no effect.

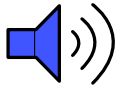
```
1  -- Exceptions
2  throwSTM :: Exception e => e -> STM a
3  catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a
4
5  atomically $ do
6    n   <- readTVar v_n
7    lim <- readTVar v_lim
8    writeTVar v_n (n+1)
9
10   if n > lim then throwSTM (AssertionFailed "fail")
11   else if n == lim then retry
12   else return ()
```

# Encode MVar with STM 

- Operations on MVar can be defined as STM.

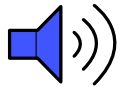
```haskell
type TMVar a = TVar (Maybe a)

newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = newTVar Nothing

takeTMVar :: TMVar a -> STM a
takeTMVar mv = do
  v <- readTVar mv
  case v of Nothing    -> retry -- block until mv has data
            Just value -> do writeTVar mv Nothing
                             return value
```

# Encode MVar with STM



- Operations on MVar can be defined as STM.

```haskell
type TMVar a = TVar (Maybe a)

putTMVar :: TMVar a -> a -> STM ()
putTMVar mv value = do
  v <- readTVar mv
  case v of Nothing    -> writeTVar mv $ Just value
            Just value -> retry

tryPutTMVar :: TMVar a -> a -> STM Bool
tryPutTMVar mv value = do
  do { putTMVar mv value; return True }
  `orElse` return False
```

# Multicast channel 

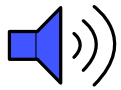
- Represent buffered data by a linked-list, where each node has a data and a TVar to point to the tail.

```
type Chain a = TVar (Item a)  -- item pointer
type Item a = Empty           -- end item of the list
            | Full a (Chain a)-- node item of the list

type MChan a = TVar (Chain a) -- pointer to end item pointer
type Port a = TVar (Chain a)-- pointer to front item pointer

newMChan :: STM (MChan a)

writeMChan :: MChan a -> a -> STM ()

newPort :: MChan a -> STM (Port a)

readPort :: Port a -> STM a
```

- Write data to the end of the list and read data from the front.

```
   --> Full a1 --> Full a2 --> Full a3 --> Empty
   |                                      |
  Port                                  MChan
```

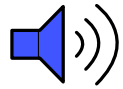
# Multicast channel



- Multiple threads can read from (write into) the same channel.

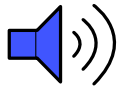
```
1  type Chain a = TVar (Item a)  -- item pointer
2  type Item a = Empty | Full a (Chain a)
3
4  type MChan a = TVar (Chain a) -- pointer to end item pointer
5  type Port a = TVar (Chain a)-- pointer to front item pointer
6
7  newMChan = do { c <- newTVar Empty; newTVar c }
8
9  writeMChain mc v = do
10    c <- readTVar mc        -- end item pointer
11    c' <- newTVar Empty     -- new end item
12    writeTVar c $ Full v c' -- insert a node item
13    writeTVar mc c'         -- update channel
```

# Multicast channel



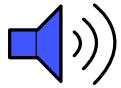
- Multiple threads can read from (write into) the same channel.

```
1  type Chain a = TVar (Item a)  -- item pointer
2  type Item a = Empty | Full a (Chain a)
3
4  type MChan a = TVar (Chain a) -- pointer to end item pointer
5  type Port a = TVar (Chain a)-- pointer to front item pointer
6
7  newPort mc = do { c <- newTVar mc; newTVar c }
8
9  readPort p = do
10    c <- readTVar p          -- front item pointer
11    i <- readTVar c          -- front item
12    case i of Empty -> retry  -- block on empty
13              Full v c' ->
14                do writeTVar p c' -- update port
15                   return v       -- return data
```

# Multicast channel 

- Merging operations on channel

```haskell
type Chain a = TVar (Item a)
type Item a = Empty | Full a (Chain a)
type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)

-- read from channel p1 or channel p2
atomically $ readPort p1 `orElse` readPort p2

-- read from channel p1 or terminate on signal from m1
atomically $ readPort p1 `orElse` takeTMVar m1

-- merge values from a list of STMs
merge :: [STM a] -> STM a
merge = foldr1 orElse
```

# Top-level TVar 

- Top-level TVars created with *newTVarIO* allow explicit sharing of TVars between threads.

```
1 newTVar :: a -> STM (TVar a)  -- TVar used in STM
2
3 newTVarIO :: a -> IO (TVar a) -- TVar used in IO
4
5 readTVar :: TVar a -> STM a   -- read TVar in STM
6
7 readTVarIO :: TVar a -> IO a  -- read TVar in IO
8 -- equal to atomically . readTVar but faster
```