Parallel & Concurrent Haskell 4: Software Transactional Memory

Simon Marlow
Software transactional memory

• An alternative to MVar for managing
  – shared state
  – communication

• STM has several advantages:
  – compositional
  – much easier to get right
  – much easier to manage error conditions (including async exceptions)
Example: a window manager

One desktop has *focus*. The user can change the focus.
Example: a window manager

One desktop has focus. The user can change the focus.

The user can move windows from one desktop to another.
Example: a window manager

One desktop has focus. The user can change the focus.

The user can move windows from one desktop to another.

Applications can move their own windows, and pop-up new windows.
How to implement this?

• Suppose we want to structure the window manager in several threads, one for each input/output stream:
  – One thread to listen to the user
  – One thread for each client application
  – One thread to render the display

• The threads share the state of the desktops – how should we represent it?
Option 1: a single MVar

\[
\text{type Display} = \text{MVar} \ (\text{Map Desktop} \ (\text{Set Window}))
\]

- Advantages:
  - simple
- Disadvantages:
  - single point of contention. (not only performance: one misbehaving thread can block everyone else.)
- representing the Display by a process (aka the actor model) suffers from the same problem
- Can we do better?
Option 2: one MVar per Desktop

This avoids the single point of contention, but a new problem emerges. Try to write an operation that moves a window from one Desktop to another:

```haskell
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)
where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)
```

```haskell
type Display = MVar (Map Desktop (Set Window))
type Display = Map Desktop (MVar (Set Window))
```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()

moveWindow disp win a b = do
    wa <- takeMVar ma
    wb <- takeMVar mb
    putMVar ma (Set.delete win wa)
    putMVar mb (Set.insert win wb)

where
    ma = fromJust (Map.lookup a disp)
    mb = fromJust (Map.lookup b disp)

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()

moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)

where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.

• Ok so far, but what if we have two concurrent calls to moveWindow:

Thread 1: moveWindow disp w1 a b
Thread 2: moveWindow disp w2 b a
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()

moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)

where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

• Ok so far, but what if we have two concurrent calls to moveWindow:

  Thread 1: moveWindow disp w1 a b
  Thread 2: moveWindow disp w2 b a

• Thread 1 takes the MVar for Desktop a

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.
moveWindow :: Display -> Window -> Desktop -> Desktop
-> IO ()
moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)
where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

• Ok so far, but what if we have two concurrent calls to moveWindow:
  Thread 1: moveWindow disp w1 a b
  Thread 2: moveWindow disp w2 b a

• Thread 1 takes the MVar for Desktop a
• Thread 2 takes the MVar for Desktop b

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.
moveWindow :: Display -> Window -> Desktop -> Desktop
             -> IO ()
moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)

where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.

- Ok so far, but what if we have two concurrent calls to moveWindow:

  Thread 1: moveWindow disp w1 a b
  Thread 2: moveWindow disp w2 b a

- Thread 1 takes the MVar for Desktop a
- Thread 2 takes the MVar for Desktop b
- Thread 1 tries to take the MVar for Desktop b, and blocks
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()

moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)

where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

• Ok so far, but what if we have two concurrent calls to moveWindow:

  Thread 1: moveWindow disp w1 a b
  Thread 2: moveWindow disp w2 b a

  • Thread 1 takes the MVar for Desktop a
  • Thread 2 takes the MVar for Desktop b
  • Thread 1 tries to take the MVar for Desktop b, and blocks
  • Thread 2 tries to take the MVar for Desktop a, and blocks

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state.
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()

moveWindow disp win a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)

where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)

• Ok so far, but what if we have two concurrent calls to moveWindow:

  Thread 1: moveWindow disp w1 a b
  Thread 2: moveWindow disp w2 b a

  • Thread 1 takes the MVar for Desktop a
  • Thread 2 takes the MVar for Desktop b
  • Thread 1 tries to take the MVar for Desktop b, and blocks
  • Thread 2 tries to take the MVar for Desktop a, and blocks
  • DEADLOCK (“Dining Philosophers”)

Be careful to take both Mvars before putting the results, otherwise another thread could observe an inconsistent intermediate state
How can we solve this?

• Impose a fixed ordering on `MVars`, make `takeMVar` calls in the same order on every thread
  – painful
  – the whole application, including libraries, must obey the rules (anti-modular)
  – error-checking can be done at runtime, but complicated (and potentially expensive)
STM solves this

```
type Display = Map Desktop (TVar (Set Window))

moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = atomically $ do
    wa <- readTVar ma
    wb <- readTVar mb
    writeTVar ma (Set.delete win wa)
    writeTVar mb (Set.insert win wb)
where
    ma = fromJust (Map.lookup a disp)
    mb = fromJust (Map.lookup b disp)
```

• The operations inside `atomically` happen indivisibly to the rest of the program (it is a *transaction*)
• ordering is irrelevant – we could reorder the `readTVar` calls, or interleave read/write/read/write
- Basic STM API:

```haskell
data STM a -- abstract
instance Monad STM -- amongst other things

atomically :: STM a -> IO a

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

- The implementation does not use a global lock: two transactions operating on disjoint sets of TVars can proceed simultaneously
Composability

- STM is composable
- e.g. write an operation to swap two windows

```haskell
swapWindows :: Display
            -> Window -> Desktop
            -> Window -> Desktop
            -> IO ()
```

- with **MVars** we would have to write a special-purpose routine to do this...
• with STM we can build on what we already have:

```haskell
swapWindows :: Display -> Window -> Desktop -> Window -> Desktop -> IO ()
swapWindows disp w a v b = atomically $ do
  moveWindowSTM disp w a b
  moveWindowSTM disp v b a
```

• *(moveWindowSTM is just moveWindow without atomically – this is typically how STM operations are provided)*

• STM allows us to *compose* stateful operations into larger transactions
  • thus allowing more reuse
  • and modularity – we don’t have to know how moveWindowSTM works internally to be able to compose it.
So far we saw how to use STM to build atomic operations on shared state.

But concurrency often needs a way to manage blocking – that is, waiting for some condition to become true – e.g. a channel is non-empty.

Haskell’s STM API has a beautiful way to express blocking too...
retry :: STM a
retry :: STM a

• that's it!
retry :: STM a

- that’s it!
- the semantics of retry is just “try the current transaction again”
retry :: STM a

• that’s it!
• the semantics of retry is just “try the current transaction again”
• e.g. block until a TVar contains a non-zero value:

```haskell
atomically $ do
  x <- readTVar v
  if x == 0 then retry
      else return x
```

retry :: STM a

- that’s it!
- the semantics of retry is just “try the current transaction again”
- e.g. block until a TVar contains a non-zero value:

```haskell
atomically $ do
  x <- readTVar v
  if x == 0 then retry
  else return x
```

- busy-waiting is a possible implementation, but we can do better:
that’s it!
the semantics of retry is just “try the current transaction again”
e.g. block until a TVar contains a non-zero value:

```haskell
atomically $ do
    x <- readTVar v
    if x == 0 then retry
    else return x
```

busy-waiting is a possible implementation, but we can do better:

• obvious optimisation: wait until some state has changed
that’s it!
• the semantics of retry is just “try the current transaction again”
• e.g. block until a TVar contains a non-zero value:

\[\text{atomically } \{ \text{do} \begin{align*} & \ x \leftarrow \text{readTVar } v \\
& \text{if } x == 0 \text{ then retry } \\
& \text{else return } x \end{align*} \}\]

• busy-waiting is a possible implementation, but we can do better:
  • obvious optimisation: wait until some state has changed
  • specifically, wait until any TVars accessed by this transaction so far have changed (this turns out to be easy for the runtime to arrange)
retry :: STM a

• that’s it!
• the semantics of retry is just “try the current transaction again”
• e.g. block until a TVar contains a non-zero value:

```haskell
atomically $ do
    x <- readTVar v
    if x == 0 then retry
    else return x
```

• busy-waiting is a possible implementation, but we can do better:
  • obvious optimisation: wait until some state has changed
  • specifically, wait until any TVars *accessed by this transaction so far* have changed (this turns out to be easy for the runtime to arrange)
  • so retry gives us blocking – the current thread is blocked waiting for the TVars it has read to change
Implementing MVars

• With STM and retry we can implement a composable **MVar**

  ```haskell
  data TMVar a
  takeTMVar :: TMVar a -> STM a
  putTMVar :: TMVar a -> a -> STM ()
  ```

• How should we represent a TMVar?
  – it is either empty or full
  – which suggests **Maybe a**:

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

• Implement takeTMVar:

```haskell
takeTMVar (TMVar t) = do
  m <- readTVar t
  case m of
    Nothing -> retry
    Just a  -> return a
```

If the TMVar is empty, we just retry!

• putTMVar is very similar
• **TMVar** is much more flexible than **MVar**. e.g.

```haskell
takeTwo :: TMVar a -> TMVar b -> STM (a,b)
takeTwo ma mb = do
  a <- takeTMVar ma
  b <- takeTMVar mb
  return (a,b)
```

• takes both **TMVars**, or blocks. We cannot express this using **MVar** without significant extra complication.

```haskell
takeIfNonZero :: TMVar Int -> STM Int
takeIfNonZero m = do
  a <- takeTMVar m
  when (a /= 0) retry
  return a
```

• only takes the value if it is non-zero. Again, this is hard to express with **MVar**
Using blocking in the window manager

• We want a thread responsible for rendering the currently focussed desktop on the display
  – it must re-render when something changes
  – the user can change the focus
  – windows can move around

• there is a TVar containing the current focus:

  type UserFocus = TVar Desktop

• so we can get the set of windows to render:

  getWindows :: Display -> UserFocus -> STM (Set Window)
  getWindows disp focus = do
    desktop <- readTVar focus
    readTVar (fromJust (Map.lookup desktop disp))
Given:  

```
render :: Set Window -> IO ()
```

Here is the rendering thread:

```
renderThread :: Display -> UserFocus -> IO ()
renderThread disp focus = do
  wins <- atomically $ getWindows disp focus
  loop wins
where
  loop wins = do
    render wins
    next <- atomically $ do
      wins' <- getWindows disp focus
      if (wins == wins')
        then retry
        else return wins'
    loop next
```

so we only call render when something has changed.

The runtime ensures that the render thread remains blocked until either

- the focus changes to a different Desktop
- the set of Windows *on the current Desktop* changes
• No need for explicit wakeups
  • the runtime is handling wakeups automatically
• state-modifying code doesn’t need to know who to wake up – more modularity
• no “lost wakeups” – a common type of bug with condition variables
• One of the exercises uses this pattern: fix errors in text as you type (exercise 3.3)
break...
Channels in STM

• Earlier we implemented channels with MVars
• Instructive to see what channels look like in STM
• Also we’ll introduce one final operation for composing blocking operations in STM
• And how STM makes it much easier to handle exceptions (particularly asynchronous exceptions)
data TChan a = TChan (TVar (TVarList a))
            (TVar (TVarList a))

type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)

• Why do we need TNil & TCons?
  • unlike MVars, TVars do not have an empty/full state, so we have to program it
  • We could have used TMVar, but this is a bit more direct
• Otherwise, the structure is exactly the same as the MVar implementation

readTChan :: TChan a -> STM a
readTChan (TChan read _write) = do
  listhead <- readTVar read
  head <- readTVar listhead
  case head of
    TNil -> retry
    TCons a tail -> do
      writeTVar read tail
      return a
Benefits of STM channels (1)

• Correctness is straightforward: do not need to consider interleavings of operations
  – (recall with MVar we had to think carefully about what happened with concurrent read/write, write/write, etc.)
Benefits of STM channels (2)

• more operations are possible, e.g.:

```haskell
unGetTChan :: TChan a -> a -> STM ()
unGetTChan (TChan read _write) a = do
  listhead <- readTVar read
  newhead <- newTVar (TCons a listhead)
  writeTVar read newhead
```

• (this was not possible with MVar, trivial with STM)
• Composable blocking. Suppose we want to implement

\[
\text{readEitherTChan} :: \text{TChan } a \rightarrow \text{TChan } b \rightarrow \text{STM } (\text{Either } a\ b)
\]

• we want to write a transaction like

\[
\text{readEitherTChan } a\ b = \text{atomically$}$
\[
(\text{fmap Left}$ $\text{readTChan } a)
\]

???

(\text{fmap Right}$ $\text{readTChan } b)\]
Benefits of STM channels (3)

• Composable blocking. Suppose we want to implement

\[
\text{readEitherTChan} :: \text{TChan } a \rightarrow \text{TChan } b \rightarrow \text{STM } (\text{Either } a \ b)
\]

• we want to write a transaction like

\[
\text{readEitherTChan } a \ b = \text{atomically } $
\text{(do } x \leftarrow \text{readTChan } a; \text{return } (\text{Left } x))$
\text{???}
\text{(do } x \leftarrow \text{readTChan } b; \text{return } (\text{Right } x))$
\]
orElse :: STM a \rightarrow STM a \rightarrow STM a

- execute the first argument
- if it returns a value:
  - that is the value returned by orElse
- if it retries:
  - discard any effects (writeTVars) it did
  - execute the second argument
- orElse is another way to compose transactions: it runs \textit{either} one or the other
Benefits of STM channels (4)

- Asynchronous exception safety.

If an exception is raised during a transaction, the effects of the transaction are discarded, and the exception is propagated as normal.

- error-handling in STM is trivial: since the effects are discarded, all invariants are restored after an exception is raised.

- Asynchronous exception safety comes for free!

- The simple TChan implementation is already async-exception-safe
What about some examples?

- Let’s extend our geturls program to stop when the first page is returned.
- Remember our little Async API?

```haskell
data Async a = Async ThreadId (MVar (Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- forkIO (do r <- try action; putMVar m r)
  return (Async t m)

wait :: Async a -> IO (Either SomeException a)
wait (Async t var) = readMVar var

cancel :: Async a -> IO ()
cancel (Async t var) = throwTo t ThreadKilled
```
• First we make it use **TMVar** instead of **MVar**
• And add **waitSTM**, a version of **wait** that we can compose

```haskell
data Async a = Async ThreadId (TMVar (Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyTMVarIO
  t <- forkIO (do r <- try action
              atomically (putTMVar var r))
  return (Async t var)

wait :: Async a -> IO (Either SomeException a)
wait a = atomically (waitSTM a)

waitSTM :: Async a -> STM (Either SomeException a)
waitSTM (Async _ var) = readTMVar var

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
```
• Now, we can add a new operation: \texttt{waitAny}

\begin{verbatim}
waitAny :: \texttt{[Async a] \to IO (Async a, Either SomeException a)}
waitAny asyncs =
  atomically $
  \text{foldr1 orElse}$
  \text{map (}$\text{a \to do r \leftarrow waitSTM a; return (a, r)}$$\text{)} asyncs
\end{verbatim}

• This waits until one async is complete, and then returns it.
• Now use it:

\begin{verbatim}
main = do
  as <- mapM (async.http) sites
  waitAny as
  mapM_ cancel as
  rs <- mapM wait as
  printf "\%d/\%d finished\n" (length (rights rs)) (length rs)
\end{verbatim}
• run it:

$ ./geturlsfirst
downloaded: http://www.google.com (14156 bytes, 0.08s)
1/5 finished

• Couldn’t we have done this with **MVar**?
• Yes, but
  • **waitAny** would have to fork another thread per async
  • The STM version is simpler, and composable
STM summary

• Composable atomicity
• Composable blocking
• Robustness: easy error handling
• Don’t believe the anti-hype!
• Why would you still use MVar?
  – fairness
  – single-wakeup
  – performance