Parallel and Concurrent Haskell
Part II

Simon Marlow
(Microsoft Research, Cambridge, UK)

Concurrent Haskell

- Recap:
  - concurrent programming is about threads of control
  - concurrency is necessary for dealing with multiple sources of input/output:
    - network connections
    - GUI, user input
    - database connections
  - threads of control let you handle multiple input/output sources in a modular way: the code for each source is written separately

Summary

- In this part of the course we’re going to cover
  - Basic concurrency
    - forkIO
    - MVars
  - Asynchronous exceptions
    - cancellation
    - timeout
  - Software Transactional Memory

Forking threads

forkIO :: IO () -> IO ThreadId

- creates a new thread to run the IO ()
- new thread runs “at the same time” as the current thread and other threads

Interleaving example

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (forever (putChar 'A'))
  forkIO (forever (putChar 'B'))
  threadDelay (10^6)
```

ThreadId

- what can you do with a ThreadId?
  - check status with GHC.Conc.threadStatus (useful for debugging):

```
forkIO :: IO () -> IO ThreadId
forever :: m a -> m a
putChar :: Char -> IO ()
threadDelay :: Int -> IO ()
```

```
> import Control.Concurrent
> do { t <- forkIO (threadDelay (10^6)); GHC.Conc.threadStatus t }
ThreadRunning
> do { t <- forkIO (threadDelay (10^6)); yield; GHC.Conc.threadStatus t }
ThreadRunning BlockedOnMVar
```

- Also:
  - compare for equality
  - kill / send exceptions (later...)

ThreadId
A note about performance

• GHC's threads are lightweight

> ./Main 1000000 +RTS -s
Creating pipeline with 1000000 processes in it.
Pumping a single message through the pipeline.
- create, pump1, pump2, create, pump1, pump2
1000000 1.110 1.110 1.48 0.77 0.77

• 10^6 threads requires 1.5Gb — 1.5k/thread
- most of that is stack space, which grows/shrinks on demand
- cheap threads makes it feasible to use them liberally, e.g. one thread per client in a server

Communication: MVars

• MVar is the basic communication primitive in Haskell.

Communication:

• And conversely: putMVar blocks when the MVar is full.

Example: overlapping I/O

• One common use for concurrency is to overlap multiple I/O operations
- overlapping I/O reduces latencies, and allows better use of resources

Example: overlapping I/O

• overlapping I/O is easy with threads: just do each I/O in a separate thread
- the runtime takes care of making this efficient
- e.g. downloading two web pages

Abstract the common pattern

• Fork a new thread to execute an IO action, and later wait for the result

Abstract the common pattern

Using Async....

Using Async....

Downloading URLs

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 146

<!DOCTYPE html>
<html>
  <head><title>Shovel</title></head>
  <body>
    <h1>Shovel</h1>
    <p>A spade in another form</p>
  </body>
</html>
A driver to download many URLs

```haskell
main = mapM (async.http) sites >>= mapM wait
where
  http url = do
    (page, time) <- timelt $ getURL url
    printf "downloaded: %s (%d bytes, %.2fs)\n" url (B.length page) time
```

downloaded: http://www.google.com (14524 bytes, 0.17s)
downloaded: http://www.bing.com (24740 bytes, 0.18s)
downloaded: http://www.wikipedia.com/wiki/Spade (62586 bytes, 0.60s)
downloaded: http://www.wikipedia.com/wiki/Shovel (68897 bytes, 0.60s)
downloaded: http://www.yahoo.com (153065 bytes, 1.11s)

An MVar is also...

- lock
  - MVar () behaves like a lock: full is unlocked, empty is locked
    - Can be used as a mutex to protect some other shared state, or a
      critical region
- one-place channel
  - Since an MVar holds at most one value, it behaves like an
    asynchronous channel with a buffer size of one
- container for shared state
  - e.g. MVar (Map key value)
    - convert persistent data structure into ephemeral
      - efficient (but there are other choices besides MVar)
- building block
  - MVar can be used to build many different concurrent data
    structures/abstractions...

Unbounded buffered channels

- Interface: `data Chan a = abstract
  newChan :: IO (Chan a)
  writeChan :: Chan a -> a -> IO ()
  readChan :: Chan a -> IO a`
- write and read do not block (indeinitely)
- we are going to implement this with MVars
- one easy solution is just
  ```haskell
  data Chan a = MVar (Sequence a)
  ```
  or perhaps...
  ```haskell
  data Chan a = MVar [a]
  ```
- but in both of these, writers and readers will conflict with each other

Structure of a channel

- Multiple readers:
  - 2nd and subsequent readers block here
- Multiple writers:
  - 2nd and subsequent writers block here
- a concurrent read might block on old_hole until
  writeChan fills it in at the end

Unbounded buffered channels

- Interface: `data Chan a = abstract
  newChan :: IO (Chan a)
  writeChan :: Chan a -> a -> IO ()
  readChan :: Chan a -> IO a`
- write and read do not block (indeinitely)
- we are going to implement this with MVars
- one easy solution is just
  ```haskell
  data Chan a = MVar [a]
  ```
  or perhaps...
  ```haskell
  data Chan a = MVar (Sequence a)
  ```
- but in both of these, writers and readers will conflict with each other

Implementation

- Multiple readers:
  - 2nd and subsequent readers block here
- Multiple writers:
  - 2nd and subsequent writers block here
- a concurrent read might block on old_hole until
  writeChan fills it in at the end
Adding more operations

- Add an operation for pushing a value onto the read end: `unGetChan :: Chan a -> a -> IO ()`
- Doesn’t seem too hard:

```
unGetChan (Chan readVar _) val = do
  new_read_end <- newEmptyMVar
  read_end <- takeMVar readVar
  putMVar new_read_end (Item val read_end)
  putMVar readVar new_read_end
```

But...

- This doesn’t work as we might expect:

```
c <- newChan :: IO (Chan String)
forkIO $ do
  v <- readChan c; print v
ThreadId 217
writeChan c "hello"
forkIO $ do
  v <- readChan c; print v
ThreadId 243
unGetChan c "hello"
```
- we don’t expect `unGetChan` to block
- but it needs to call `takeMVar` on the read end, and the other thread is currently holding that MVar
- No way to fix this...
- Building larger abstractions from MVars can be tricky
- Software Transactional Memory is much easier (later...)

A note about fairness

- Threads blocked on an MVar are processed in FIFO order
- No thread can be blocked indefinitely, provided there is a regular supply of `putMVar`s (fairness)
- Each `putMVar` wakes exactly one thread, and performs the blocked operation atomically (single-wakeup)

MVars and contention

- Fairness can lead to alternation when two threads compete for an MVar
  - thread A: `takeMVar` (succeeds)
  - thread B: `takeMVar` (blocks)
  - thread A: `putMVar` (succeeds, and wakes thread B)
  - thread A: `takeMVar` again (blocks)
  - cannot break the cycle, unless a thread is pre-empted while the MVar is full
- MVar contention can be expensive!

Cancellation/interruption

(asynchronous exceptions)

Motivation

- Often we want to interrupt a thread. e.g.
  - in a web browser, the user presses “stop”
  - in a server application, we set a time-out on each client, close the connection if the client does not respond within the required time
  - if we are computing based on some input data, and the user changes the inputs via the GUI
Isn’t interrupting a thread dangerous?

- Most languages take the polling approach:
  - you have to explicitly check for interruption
  - maybe built-in support in e.g. I/O operations
- Why?
  - because fully-asynchronous interruption is too hard to program
  - with in an imperative language.
  - Most code is modifying state, so asynchronous interruption will
    often leave state inconsistent.
- In Haskell, most computation is pure, so
  - completely safe to interrupt
  - furthermore, pure code cannot poll
- Hence, interruption in Haskell is asynchronous
  - more robust: don’t have to remember to poll
  - but we do have to be careful with I/O code

Interrupting a thread

- Throws the exception \( e \) in the given thread
- So interruption appears as an exception
  - this is good – we need exception handlers to clean up
  - in the event of an error, and the same handlers will
    work for interruption too.
- Code that is already well-behaved with respect to
  - exceptions will be fine with interruption.

Example

- Let’s extend the async API with cancellation
- So far we have:

  ```haskell
  newtype Async a = Async (MVar a)
  async :: IO a -> IO (Async a)
  async io = do
    m <- newEmptyMVar
    forkIO $ do
      r <- io;
      putMVar m r
    return (Async m)

  wait :: Async a -> IO a
  wait (Async m) = readMVar m
  
  cancel :: Async a -> IO ()
  
  bracket (newTempFile "temp")
  (\file -> removeFile file)
  (\file -> ...)
  ```

- we want to add:

  ```haskell
  cancel :: Async a -> IO ()
  ```

- Cancellation is an exception, so wait should
  return the exception that was thrown.
  - This also means that wait will correctly
    handle exceptions caused by errors.

  ```haskell
  newtype Async a = Async (ThreadID (Either SomeException a))
  async :: IO a -> IO (Async a)
  async io = do
    t <- forkIO $ do
      (left, right) <- runIOMonad (io >>= (\c -> do
        c <- putChar
        when (c == 'q') $ mapM_ cancel as
      )
      )
    return (Async t)

  wait :: Async a -> IO a
  wait (Async _ var) = takeMVar var
  ```

Example

```haskell
main = do
  as <- mapM (async.http) sites
  forkIO $ do
    hSetBuffering stdin NoBuffering
    forever $ do
      c <- getChar
      when (c == 'q') $ mapM_ cancel as
  rs <- mapM wait as
  printf "%d/%d finished\n" (length (rights rs)) (length rs)
```
• Points to note:
  • We are using a large/complicated HTTP library underneath, yet it supports interruption automatically
  • Having asynchronous interruption be the default is very powerful
  • However: dealing with truly mutable state an interruption still requires some care...

• Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.
• Haskell provides mask for this purpose:

```
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

• Use it like this:

```
problem m f = do
  a <- takeMVar m
  r <- f a `catch` \\
  \e -> do putMVar m a; throw e
  putMVar m r
```

• mask takes as its argument a function (\[\_\] \[\_\])
• during execution of (\[\_\] \[\_\]) asynchronous exceptions are masked (blocked until the masked portion returns)
• The value passed in as the argument restore is a function (\[IO a \rightarrow IO a\]) that can be used to restore the previous state (unmasked or masked) inside the masked portion.

• Operations that block are declared to be interruptible and may receive asynchronous exceptions, even inside mask.

• Problem:
  - call takeMVar
  - perform an operation (\[\_\] \[\_\] \[\_\]) on the value
  - put the new value back in the MVar
  - if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

```
problem m f = do
  a <- takeMVar m
  do r <- f a
  putMVar m r
  `catch` \e -> do putMVar m a; throw e
```

• So did this solve the problem?

```
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

• async exceptions cannot be raised in the red portions... so we always safely put back the MVar, restoring the invariant
• But: what if takeMVar blocks?
  • We are inside mask, so the thread cannot be interrupted. Bad!!
  • We didn’t really want to mask takeMVar, we only want it to atomically enter the masked state when takeMVar takes the value

```
problem m f = mask $ \restore -> do
  a <- takeMVar m
  do r <- f a
  putMVar m r
  `catch` \e -> do putMVar m a; throw e
```

• hmm, don’t we have another problem now?

```
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

• putMVar is interruptible too!
• interruptible operations only receive asynchronous exceptions if they actually block
  • In this case, we can ensure that putMVar will never block, by requiring that all accesses to this MVar use a take/put pair, not just a put.
  • Alternatively, use the non-blocking version of putMVar, tryPutMVar
Async-exception safety

- IO code that uses state needs to be made safe in the presence of async exceptions
- ensure that invariants on the state are maintained if an async exception is raised.
- We make this easier by providing combinators that cover common cases.
- e.g. the function problem we saw earlier is a useful way to safely modify the contents of an MVar:

```haskell
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
```

Making Chan safe

- We had this:

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)
```

- use modifyMVar_

```haskell
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  modifyMVar_ writeVar $ \old_hole -> do
    putMVar old_hole (Item val new_hole)
  return new_hole
```

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 async-exception safety

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 and float 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 desktop – how should we represent it?
Option 1: a single MVar

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:

```
moveWindow disp w a b = do
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete w wa)
  putMVar mb (Set.insert w wb)
  where
  ma = fromJust (lookup disp a)
  mb = fromJust (lookup disp b)
```

• 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”)

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 must obey the rules (including libraries)
  - error-checking can be done at runtime, but complicated (and potentially expensive)

STM solves this

```
moveWindow disp w a b = atomically $ do
  wa <- readTVar ma
  wb <- readTVar mb
  writeTVar ma (Set.delete w wa)
  writeTVar mb (Set.insert w wb)
  where
  ma = fromJust (lookup disp a)
  mb = fromJust (lookup disp b)
```

• when we need to perform atomic operations on multiple state items at once, STM lets us do this in a robust and painless way
  - just write the obvious sequential code, wrap it in atomically

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

• Basic STM API:

```
data TVar a -- abstract
instance MonadSTM TVar -- amongst other things
atomically :: STM a -> IO a
data STM a -- abstract
  instance MonadSTM STM -- amongst other things
  atomically :: STM a -> IO a
```
Modularity

- STM regains some modularity compared with MVars / locks
- e.g. write an operation to swap two windows

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

- with MVars we would have to write a special-purpose routine to do this...

STM and blocking

- 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

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

  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

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))
```

- with STM we can build on what we already have:

```
moveWindows disp w x b = atomically $ do
  maxMVar disp STM write w a b
  maxMVar dispSTM disp a b
```

- (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.

```
swapWindows :: Display

  | renderThread disp focus = do
  |   render Thread disp focus = do
  |     wins <- atomically $ getWindows disp focus
  |     loop wins
  |     | wins = do
  |     |     render wins
  |     |     focus' <- atomically $ do
  |     |     wins' <- getWindows disp focus
  |     |     if focus == focus' then retry
  |     |     else return wins'
  |     | next <- atomically $ do
  |     |     wins' <- getWindows disp focus
  |     |     loop wins
  |     |   else return wins
```
• 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

Channels in STM

• Earlier we implemented channels with MVars
• Instructive to see what channels look like in STM
• Also we’ll see how STM handles composing transactions that block
• And how STM makes it much easier to handle exceptions (particularly asynchronous exceptions)

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.:

Benefits of STM channels (3)

• Composable blocking. Suppose we want to implement

• we want to write a transaction like

• (this was not possible with MVar, trivial with STM)
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

STM summary

- Composable atomicity
- Composable blocking
- Robustness: easy error handling

Lab

- Download the sample code here:
  \url{http://community.haskell.org/~simonmar/par-tutorial.tar.gz}
- or get it with git:
  \texttt{git clone git@github.com:simonmar/par-tutorial.git}
- code is in par-tutorial/code
- lab exercises are here:
  \url{http://community.haskell.org/~simonmar/lab-exercises.pdf}

- bullet list
  \begin{itemize}
  \item code box
  \item interaction box
  \end{itemize}

Hit 'q' to stop the downloads