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

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

$ ghc fork.hs
[1 of 1] Compiling Main            ( fork.hs, fork.o )
Linking fork ...
$ ./fork | tail -c 300
AAAAAAAABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
$
ThreadId

```haskell
forkIO :: IO () -> IO ThreadId
```

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

```haskell
> 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 }
ThreadBlocked BlockedOnMVar
```

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

A note about performance

- GHC’s threads are *lightweight*

```bash
> ./Main 1000000 1 +RTS -s
Creating pipeline with 1000000 processes in it.
Pumping a single message through the pipeline.
Pumping a 1 messages through the pipeline.
```

<table>
<thead>
<tr>
<th>n</th>
<th>create</th>
<th>pump1</th>
<th>pump2</th>
<th>create/n</th>
<th>pump1/n</th>
<th>pump2/n</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>us</td>
<td>us</td>
<td>us</td>
</tr>
<tr>
<td>1000000</td>
<td>5.98</td>
<td>1.77</td>
<td>1.77</td>
<td>5.98</td>
<td>1.77</td>
<td>1.77</td>
</tr>
</tbody>
</table>

- $10^6$ 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.

```haskell
data MVar a -- abstract

newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

- 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

• 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
getURL :: String -> IO String

do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar

  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r

  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r

  r1 <- takeMVar m1
  r2 <- takeMVar m2
  return (r1, r2)
Abstract the common pattern

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

```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

readMVar :: MVar a -> IO a
readMVar m = do
  a <- takeMVar m
  putMVar m a
  return a
```
Using Async....

do
    a1 <- async $ getURL "http://www.wikipedia.org/wiki/Shovel"
    a2 <- async $ getURL "http://www.wikipedia.org/wiki/Spade"
    r1 <- wait m1
    r2 <- wait m2
    return (r1,r2)
A driver to download many URLs

sites = ["http://www.bing.com",
        "http://www.google.com",
        ...
]

main = mapM (async.http) sites >>= mapM wait
where
    http url = do
        (page, time) <- timeit $ 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:

```haskell
data Chan a -- abstract

newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan :: Chan a -> IO a
```

• write does not block (indefinitely)
• 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
Structure of a channel

```haskell
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```
Implementation

newChan :: IO (Chan a)
newChan = do
  hole <- newEmptyMVar
  readVar <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)

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)

readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar
  Item val new <- takeMVar stream
  putMVar readVar new
  return val
Concurrent behaviour

• Multiple readers:
  – \(2^{nd}\) and subsequent readers block here

• Multiple writers:
  – \(2^{nd}\) 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:
  \[\text{unGetChan} :: \text{Chan } a \rightarrow a \rightarrow \text{IO } ()\]

• Doesn’t seem too hard:

```haskell
unGetChan :: Chan a -> a -> IO ()
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
```

Diagram:
- `new_read_end` connected to `val` and `read_end`
But...

- This doesn’t work as we might expect:

```haskell
> c <- newChan :: IO (Chan String)
> forkIO $ do v <- readChan c; print v
ThreadId 217
> writeChan c "hello"
"hello"
> forkIO $ do v <- readChan c; print v
ThreadId 243
> unGetChan c "hello"
... blocks ....
```

- 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 putMVars (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 IO 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.

```haskell
throwTo :: Exception e => ThreadId -> e -> IO ()
```

```haskell
bracket (newTempFile "temp")
  (\file -> removeFile file)
  (\file -> ...)
```

- threads can also `catch` interruption exceptions and do something – e.g. useful for time-out
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
```

- we want to add:

```haskell
cancel :: Async a -> IO ()
```
• **cancel** is going to call `throwTo`, so it needs the `ThreadId`. Hence we need to add `ThreadId` to Async.

```haskell
newtype Async a = Async ThreadId (MVar a)

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

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
```

• but what about `wait`? previously it had type:

```haskell
wait :: Async a -> IO a
```

• but what should it return if the `Async` was cancelled?
• 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
                 (MVar (Either SomeException a))

async :: IO a -> IO (Async a)
async io = do
    m <- newEmptyMVar
    t <- forkIO (do r <- action; putMVar var (Right r)
                        `catch` e -> putMVar var (Left e))
    return (Async t m)

wait :: Async a -> IO (Either SomeException 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)
```

Hit ‘q’ to stop the downloads
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 and interruption still requires some care...
Masking asynchronous exceptions

Problem:
- call takeMVar
- perform an operation \( (f :: a \rightarrow \text{IO} \ a) \) 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

Attempt 1

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

Attempt 2
Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.

Haskell provides mask for this purpose:

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

Use it like this:

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

- mask takes as its argument a function (\restore -> ...)
- during execution of (\restore -> ...), asynchronous exceptions are masked (blocked until the masked portion returns)
- The value passed in as the argument restore is a function (:: IO a -> IO a) that can be used to restore the previous state (unmasked or masked) inside the masked portion.
• So did this solve the problem?

```haskell
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
Solution:

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

How does this help?
- **takeMVar** is now interruptible, so the thread can be interrupted while blocked
- in general, it is now very hard to write code that is uninterruptible for long periods (it has to be in a busy loop)
- Think of mask as *switch to polling mode*
  - interruptible operations poll
  - we know which ops poll, so we can use exception handlers
  - asynchronous exceptions become *synchronous* inside mask
• **hmm, don’t we have another problem now?**

```haskell
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:

\[
\text{modifyMVar} \_ :: \text{MVar } a \rightarrow (a \rightarrow \text{IO } a) \rightarrow \text{IO } ()
\]
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
```

danger!
Software Transactional Memory
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. 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

type Display = MVar (Map Desktop (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)
```
moveWindow :: Display \rightarrow Window \rightarrow Desktop \rightarrow 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
- 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 \texttt{MVars}, make \texttt{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)
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
• STM is composable
• e.g. write an operation to swap two windows

\[
\text{swapWindows} :: \text{Display} \\
\quad \rightarrow \text{Window} \rightarrow \text{Desktop} \\
\quad \rightarrow \text{Window} \rightarrow \text{Desktop} \\
\quad \rightarrow \text{IO} ()
\]

• 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.
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. 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
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:

```haskell
type UserFocus = TVar Desktop
```

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

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

\[
\text{render} :: \text{Set Window} \rightarrow \text{IO ()}
\]

Here is the rendering thread:

\[
\text{renderThread} :: \text{Display} \rightarrow \text{UserFocus} \rightarrow \text{IO ()}
\]

\[
\text{renderThread disp focus} = \text{do}
\quad \text{wins <- atomically $ \text{getWindows disp focus}}
\quad \text{loop wins}
\quad \text{where}
\quad \text{loop wins} = \text{do}
\quad \quad \text{render wins}
\quad \quad \text{next <- atomically $ \text{do}}
\quad \quad \quad \text{wins' <- getWindows disp focus}
\quad \quad \quad \text{if (wins == wins')} \quad \text{then retry}
\quad \quad \quad \text{else return wins'}
\quad \text{loop next}
\]

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
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)
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
• 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
• 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)
Benefits of STM channels (3)

- Composable blocking. Suppose we want to implement

```haskell
readEitherTChan :: TChan a -> TChan b -> STM (Either a b)
```

- we want to write a transaction like

```haskell
readEitherTChan a b = atomically $
  (fmap Left $ readTChan a)
  `orElse`
  (fmap Right $ readTChan b)
```

```haskell
orElse :: STM a -> STM a -> 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 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
• Don’t believe the anti-hype!
• Why would you still use MVar?
  – fairness
  – single-wakeup
  – performance
Lab

- Download the sample code here:
  
  http://community.haskell.org/~simonmar/par-tutorial.tar.gz

- or get it with git:
  
  git clone https://github.com/simonmar/par-tutorial.git

- code is in par-tutorial/code

- lab exercises are here:
  
  http://community.haskell.org/~simonmar/lab-exercises.pdf