First-Class Concurrency in Haskell

Keegan McAllister

Boston Area Haskell Users' Group

June 8, 2010

Haskell is functional

```haskell
sumSq :: (Num a) => [a] -> a
sumSq = sum . map (^2)
```

```
GHCi> sumSq [1..5]
55
```

Functions as data, arguments, results

Closure over free variables

Evaluate, **apply** functions, and **pattern-match**

Haskell is imperative

```haskell
main :: IO ()
main = do
  putStrLn "Enter a number:"
  x <- readLn :: IO Integer
  putStrLn ("Its square is " ++ show (x^2))
```

Looks like any other imperative language

Execute a sequence of steps by **performing actions**

The paradigm myth

Conflicting "paradigms"?

- "My problem is stateful; no point using Haskell"
- "In Haskell I should feel bad when I see `IO`"
- "Imperative code has to be repetitive"

Classic FP tricks are useful in both worlds

- Good IP requires good FP, for flexibility
- Good FP requires good IP, for practicality

First-class actions

```haskell
reverse :: String -> String
putStr  :: String -> IO ()
```

`putStr` is an "impure function"? Not really...

- `putStr` returns an "IO action"
- Action is inert and opaque, but *describes* effects
- Glue together actions using `do`, `(>>=)`, etc.
- Only one action executes: `main`

Execution $\neq$ evaluation

(GHC) Haskell is concurrent

Meaning: several actions appear to execute at once

- Explicitly create threads
- Explicitly communicate between them
- Nondeterministic due to thread scheduling

Not necessary just to use multiple cores

- see: `par`, Strategies, DPH
- but not relevant to this talk

Spawning threads

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

Takes "recipe for doing stuff"
Returns "recipe for spawning a thread to do stuff"

- In GHC, threads are lightweight
- Run on a few OS threads ($\approx$ # cores)
- All threads die when `main` thread dies

forkOS

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

Misconceptions: "I need `forkOS` to…

- …run in my own OS thread"
- …use multiple cores"
- …make calls to C without blocking all threads"

Only matters when FFI libs care about OS threads

Delaying a thread

```haskell
threadDelay :: Int -> IO ()
```

Takes *n*, returns "recipe for wasting *n* microseconds"

There's a minor API flaw…

Two things at once

```haskell
out :: String -> IO ()
out msg = forkIO (putStr msg)

main :: IO ()
main = do
  out "foo"
  out "bar"
  threadDelay (5 * 10^6)
```

Our threads print concurrently

- Output could be `foobar` or `barfoo` or `baforo`

Idea: logger thread

Scenario: multiple threads generating log events

- Need one complete message per line
- Idea: make a single thread do writes
- Bonus: no IO latency in workers

How do we send log messages to the logger thread?

Chan

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

Chan a is “channels carrying values of type a”

- Unbounded queues
- readChan blocks on empty queue

Logger: conventional design

```haskell
data Logger = MkLogger (Chan String)

startLogger :: IO Logger
startLogger = do
  chan <- newChan
  forkIO (forever
    (readChan chan >>= putStrLn))
  return (MkLogger chan)

writeMessage :: Logger -> String -> IO ()
writeMessage (MkLogger chan) msg =
  writeChan chan msg
```

Logger: conventional flaws

```haskell
example :: IO ()
example = do
  lg <- startLogger
  writeMessage lg "Hello, world!"
```

- User must learn a new type and new methods
- Chan is exposed and can be used in unexpected ways

Logger: first-class concurrency

```haskell
startLogger :: IO (String -> IO ())
startLogger = do
  chan <- newChan
  forkIO (forever
    (readChan chan >>= putStrLn))
  return (writeChan chan)

example :: IO ()
example = do
  lg <- startLogger
  lg "Hello, world!"
```

Don't expose the `Chan`, only a writer function

Logger: first-class benefits

```haskell
example :: IO ()
example = do
  lg <- startLogger
  lg "Hello, world!"
```

Simpler interface and implementation

- Same interface as `putStrLn`

Use of concurrency is hidden within `startLogger`

- User cannot tamper with our `Chan`
- Fine-grained abstraction

Idea: thread results

What if a thread produces a result?

- Query a website or a database
- Long-running stateful computation

How can the thread calling `forkIO` get the result?

MVar

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

At each moment, an `MVar t` is empty, or holds one `t`

- `takeMVar` blocks when empty
- `putMVar` blocks when non-empty

More `MVar`

Read but don't remove:

```
readMVar    :: MVar a -> IO a
```

Non-blocking:

```
tryTakeMVar :: MVar a      -> IO (Maybe a)
tryPutMVar  :: MVar a -> a -> IO Bool
```

Exception-safe modify:

```
modifyMVar  :: MVar a
            -> (a -> IO (a,b))
            -> IO b
```

Thread results

```haskell
spawn :: IO a -> IO (IO a)
spawn body = do
  v <- newEmptyMVar
  forkIO (body >>= putMVar v)
  return (readMVar v)

example :: IO ()
example = do
  let get = simpleHTTP . getRequest
  thr <- spawn (get "http://haskell.org")
  -- do other things concurrently
  result <- thr
  print result
```

Again, hidden communication

Exceptions

What if an action goes wrong and can't finish?

```
throwIO :: (Exception e)
        => e      -- what to throw
        -> IO a   -- recipe for throwing

catch   :: (Exception e)
        => IO a             -- what to try
        -> (e -> IO a)      -- how to handle exns
        -> IO a             -- recipe for trying
```

Ordinary functions, not syntax

First-class exception handling

```
try :: (Exception e)
    => IO a
    -> IO (Either e a)
try action =
  (Right <$> action) `catch` (return . Left)

example :: IO ()
example = do
  result <- try (readFile "foo.txt")
  print (result :: Either IOError String)
```

Define your own control flow!

Bracketed actions

```haskell
bracket ::        IO a    -- acquire resource
        -> (a -> IO b)   -- release resource
        -> (a -> IO c)   -- perform work
        ->        IO c
bracket acquire release act = do
  resource <- acquire
  go resource `catch` cleanup resource where
    go resource = do
      result <- act resource
      release resource
      return result
    cleanup resource exception = do
      release resource
      throwIO (exception :: SomeException)
```

Using bracket

```
withFile :: FilePath
         -> (Handle -> IO r)
         -> IO r
withFile name =
  bracket (openFile name WriteMode) hClose

example :: IO ()
example = withFile "foo.txt" $ \h -> do
   hPutStr h "Value is: "
   hPrint  h 3
```

File is closed even if write fails

Exception-safe spawn

```haskell
type Result a = Either SomeException a

spawnTry :: IO a -> IO (IO (Result a))
spawnTry body = do
  v <- newEmptyMVar
  forkIO (try body >>= putMVar v)
  return (readMVar v)

spawn :: IO a -> IO (IO a)
spawn body = do
  r <- spawnTry body
  return (r >>= either throwIO return)
```

`spawnTry` is our old `spawn` with `try` included

New `spawn` re-throws when result is demanded

An alternative

```
spawn :: IO a -> IO (IO a)
spawn body = do
  me <- myThreadId
  v  <- newEmptyMVar
  let bounce :: SomeException -> IO ()
      bounce = throwTo me
  forkIO ((body >>= putMVar v)
          `catch` bounce)
  return (readMVar v)
```

`throwTo` causes an exception in another thread

- which could be anywhere in its code

Hard to handle properly

Idea: action timeouts

We might give up on an action after some time limit.

```haskell
timeout
  :: Int            -- time limit, in microsec
  -> IO a           -- what to try
  -> IO (Maybe a)   -- yield result or give up
```

Idea: spawn a thread to send interrupting exception

- Adapted from `System.Timeout.timeout`

Unique values

Can't interfere with other exceptions

- including nested `timeout`

```
module Data.Unique where

data Unique = ... -- abstract

instance Eq  Unique where ...
instance Ord Unique where ...

newUnique :: IO Unique
```

A new exception type

```haskell
data Timeout = Timeout Unique
  deriving (Eq, Typeable)

instance Show Timeout where
  show _ = "<timeout>"

instance Exception Timeout
  -- no body needed
```

Timeout

```
timeout :: Int -> IO a -> IO (Maybe a)
timeout usec act = do
  me  <- myThreadId
  exn <- Timeout <$> newUnique

  let watchdog = threadDelay usec
                 >> throwTo me exn
      chooseExn e = guard (e == exn)
      giveUp () = return Nothing

  handleJust chooseExn giveUp $
    bracket (forkIO watchdog) killThread $
      (\_ -> Just <$> act)
```

Idea: Threads as reference cells

A thread can act like a reference cell.

We will implement

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

using `forkIO` and `Chan`.

IORef representation

```
data IORef a = IORef ((a -> IO a) -> IO ())

newIORef :: a -> IO (IORef a)
newIORef v = do
  ch <- newChan
  let f x = readChan ch >>= ($ x) >>= f
  forkIO (f v)
  return . IORef $ writeChan ch
```

Represent IORef by a function which takes “updaters”

- State is stored in the arg to f
- Chan is again hidden

IORef core operation

```haskell
data IORef a = IORef ((a -> IO a) -> IO ())

modify :: IORef a
       -> (a -> (a,b))
       -> IO b
modify (IORef update) f = do
  ch <- newChan
  update $ \v -> do
    let (v', b) = f v
    writeChan ch b
    return v'
  readChan ch
```

Create another Chan to get the result out

`IORef` derived API

The rest follows from `modify`:

```
readIORef   :: IORef a              -> IO a
writeIORef  :: IORef a -> a         -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()

readIORef   r   = modify r $ \v -> (v,   v )
writeIORef  r v = modify r $ \_ -> (v,   ())
modifyIORef r f = modify r $ \v -> (f v, ())
```

Idea: `Chan` from `MVar`

We can implement `Chan` using `MVar`

- and get the blocking behavior for free

A tour of `Control.Concurrent.Chan` source

Chan representation

Messages stored in a MVar-linked list

Read and write positions also stored in MVars

```
data Chan a
  = Chan (MVar (Stream a))  -- read  end
         (MVar (Stream a))  -- write end

type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
```

Empty Chan

Read and write end hold the same empty MVar

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

writeChan

Create a new hole; store it at the write end

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  modifyMVar_ writeVar $ \oldHole -> do
    putMVar oldHole (ChItem val newHole)
    return newHole
```

readChan

Take from the read end; blocks if empty

```
readChan :: Chan a -> IO a
readChan (Chan readVar _) =
  modifyMVar readVar $ \oldRead -> do
    (ChItem val newRead) <- readMVar oldRead
    return (newRead, val)
```

Duplicating channels

```
dupChan :: Chan a -> IO (Chan a)
dupChan (Chan _ writeVar) = do
  hole        <- readMVar writeVar
  newReadVar <- newMVar  hole
  return (Chan newReadVar writeVar)
```

- New Chan starts empty
- Gets a copy of each item written to the old Chan

The π calculus

λ calculus formalizes functional programming.
π calculus formalizes concurrent programming:

- Fork
- Create, read, write channels
- Loop forever

That's all!

- Only thing to send on a channel is a channel

π calculus syntax

```
type Name = String

data Pi
  = Pi :|: Pi               -- parallel execution
  | Inp Name Name Pi        -- read and bind var
  | Out Name Name Pi        -- write     from var
  | New Name      Pi        -- new chan in   var
  | Rep           Pi        -- loop forever
  | Nil                     -- do nothing
  | Embed (IO ()) Pi        -- for observation
```

Names are bound by `New` and by `Inp` (second arg).

Channel semantics

```
Inp x y   ≈   y <- readChan x
Out x y   ≈   writeChan x y
```

$$\boxed{\texttt{Rep (Inp x y (Inp y z Nil))}}$$

has a reader for `x` even while waiting on `y`

- `Rep x   ≈   x :|: Rep x`

π calculus interpreter

Need a Chan of Chan of Chan of...

```
data MuChan = MuChan (Chan MuChan)

type Env = Map Name MuChan

run :: Env -> Pi -> IO ()

run env (Rep p)       = forever (run env p)
run env Nil           = return ()
run env (Embed x a) = x >> run env a
```

π calculus interpreter (2)

```
run :: Env -> Pi -> IO ()

run env (a :|: b) = do
  let f x = forkIO (run env x)
  f a >> f b >> return ()

run env (New bindAs p) = do
  c <- MuChan <$> newChan
  run (insert bindAs c env) p
```

π calculus interpreter (3)

```
run :: Env -> Pi -> IO ()

run env (Inp from bindAs p) = do
  let MuChan c = env ! from
  recv <- readChan c
  forkIO $ run (insert bindAs recv env) p
  return ()

run env (Out dest from p) = do
  let MuChan c = env ! dest
  writeChan c (env ! from)
  run env p
```

λ to π

Claim: π calculus is Turing-complete

Let's compile from a simple λ calculus:

```
data Lam
  = Lam :@: Lam         -- application
  | Var Name            -- variables
  | Abs Name Lam        -- lambda abstraction
  | Eff (IO ()) Lam     -- effects
```

Evaluation has side-effects (boo! hiss!)

λ examples

```
[m,n,f,x] = map (Var . pure) "mnfx"

-- \f x -> f (f (f (... x)))
e_church k = Abs "f" . Abs "x" .
  foldr (:@:) x $ replicate k f

-- \m n f -> n (m f)
e_mult = Abs "m" . Abs "n" . Abs "f" $
  n :@: (m :@: f)

-- \n -> n (\x -> trace "S" x) (trace "0" id)
e_shownum = Abs "n" $ n
  :@: (Abs "x" (Eff (putChar 'S') x))
  :@: (Eff (putChar '0') e_id)
```

Fresh names

```haskell
type M a = State [Name] a

fresh :: M Name
fresh = State (\(x:xs) -> (x,xs))

withFresh :: (Name -> r) -> M r
withFresh f = f <$> fresh
```

Encoding pairs

A pair is a channel with two elements enqueued

```
inp2, out2 :: Name -> (Name, Name)
            -> M (Pi -> Pi)

inp2 from (bind1,bind2)
  = withFresh $ \pair k ->
      Inp from pair  $
      Inp pair bind1 $
      Inp pair bind2 $ k

out2 dest (from1,from2)
  = withFresh $ \pair k ->
      New pair       $
      Out pair from1 $
      Out pair from2 $
      Out dest pair  $ k
```

Continuation channels

A term sends its value to a channel

```
compile :: Name -> Lam -> M Pi

compile k (Var x) = return $ Out k x Nil

compile k (Eff eff a)
  = Embed eff <$> compile k a
```

Encoding functions

A function is a channel accepting "request" pairs:

- (argument, where to send result)

```
compile :: Name -> Lam -> M Pi

compile k (Abs x b) = do
  f   <- fresh
  ret <- fresh
  New f      <$>
    Out k f <$>
    Rep     <$>
    (inp2 f (x,ret) <*> compile ret b)
```

Application

```
compile :: Name -> Lam -> M Pi

compile k (x :@: y) = do
  [xk, yk, xv, yv] <- replicateM 4 fresh
  xp <- compile xk x
  yp <- compile yk y
  rp <- Inp xk xv <$>
        Inp yk yv <$>
        (out2 xv (yv, k) <*> pure Nil)
  return $
    New xk $
    New yk $
    (xp :|: yp :|: rp)
```

Testing

```
runCompile :: Lam -> Pi
runCompile b = evalState act names where
  names = map (('_':).show)
              ([1..] :: [Integer])
  act = do
    k <- fresh
    New k <$> compile k b

e = e_shownum :@:
  (e_mult :@: e_church 2 :@: e_church 3)
```

```
GHCi> run Map.empty (runCompile e)
GHCi> OSSSSSS
```

Machine code

```
New "_1" (New "_2" (New "_3" ((New "_6" (Out
"_2" "_6" (Rep (Inp "_6" "_8" (Inp "_8" "n"
(Inp "_8" "_7" (New "_9" (New "_10" ((New
"_13" (New "_14" ((Out "_13" "n" Nil :|: New
"_17" (Out "_14" "_17" (Rep (Inp "_17" "_19"
(Inp "_19" "x" (Inp "_19" "_18" (Embed <<IO
action>> (Out "_18" "x" Nil)))))))) :|: Inp
"_13" "_15" (Inp "_14" "_16" (New "_20" (Out
"_20" "_16" (Out "_20" "_9" (Out "_15" "_20"
Nil))))))) :|: Embed <<IO action>> (New
"_21" (Out "_10" "_21" (Rep (Inp "_21" "_23"
(Inp "_23" "x" (Inp "_23" "_22" (Out "_22"
"x" Nil))))))))) :|: Inp "_9" "_11" (Inp
"_10" "_12" (New "_24" (Out "_24" "_12" ...
```

Questions?

Slides online at `http://t0rch.org`