Fun With Haskell: Off to the Races

Nathaniel Wesley Filardo

January 18, 2012

Metadata
Questions?

- Any questions from last time?
- Sorry about running over; the remaining slides have been hauled into this deck.

Monad Transformers

- Hey! Sometimes we want more than one!
  - Last week: `Maybe`, `Reader`, `State`, ... Monads
  - This week: `IO`
- How can we get `Readers` that can do IO too?
  - It would be sad if we couldn't!

Monad Transformers

- Hey! Sometimes we want more than one!
  - Last week: `Maybe`, `Reader`, `State`, ... Monads
  - This week: `IO`
- How can we get `Readers` that can do IO too?
  - It would be sad if we couldn't!
- Disclaimer: this is, I think, one of the places where Haskell can use some more work. Quite recently, there is Monatron [1] which brings a lot of this onto better mathematical foundations.
- Relatedly, The Monad Zipper [4] shows a better way to manage stacks.

Monad Transformers

- `Reader` endowed a *pure* computation with additional (Monadic) functionality.
- We want something to *transform IO monadic* computations.

Monad Transformers

- `Reader` endowed a *pure* computation with additional (Monadic) functionality.
- We want something to *transform IO monadic* computations.
- Behold, the `ReaderT` in `Control.Monad.Reader`:

```
newtype ReaderT r m a =
  ReaderT {runReaderT :: r -> m a}
instance Monad m => Monad (ReaderT r m) --...
ask :: Monad m => ReaderT r m r
```

Monad Transformers

```
newtype ReaderT r m a =
  ReaderT {runReaderT :: r -> m a}
```

- `ReaderT` functions have an environment `r`, and produce an action in `m` that computes a value `a`.
- `Reader` is actually defined as a `ReaderT` on the **Identity Monad**:

```
newtype Identity a = Identity
    { runIdentity :: a }
instance Monad Identity where
  return a = Identity a
  m >>= k  = k (runIdentity m)
```

Monad Transformers

- Alright, so `Reader` wasn't interesting.
- How about `ReaderT` on IO? We'd like:

```
main = runReaderT (ask >>= putStrLn) "Test"
```

Monad Transformers

- Alright, so `Reader` wasn't interesting.
- How about `ReaderT` on IO? We'd like:

```
main = runReaderT (ask >>= putStrLn) "Test"
```

- Type failure: `Couldn't match expected type ‘ReaderT String m0 a0’ with actual type ‘IO ()’`
- Oh right: `putStrLn :: String -> IO ()`

Monad Transformers

- Alright, so `Reader` wasn't interesting.
- How about `ReaderT` on IO? We'd like:

```
main = runReaderT (ask >>= putStrLn) "Test"
```

- Type failure: `Couldn't match expected type ‘ReaderT String m0 a0’ with actual type ‘IO ()’`
- Oh right: `putStrLn :: String -> IO ()`
- Need to make `IO ()` into `ReaderT r IO ()`.

Monad Transformers

- Monad transformers also specify how to “lift” actions from the wrapped monad:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ReaderT r) --...
```

Monad Transformers

- Use `lift`:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

- So:

```
main = runReaderT
         (ask >>= lift . putStrLn)
         "Test"
```

- Often, don't have to lift: transformers defined so that, for example, an un-lifted `ask` always applies to the outermost ReaderT, even if there is stuff in the way.

Monad Transformers

In fact, stacks of transformers over `IO` and the need to lift into `IO` are so common that there's a special class and function:

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a

instance MonadIO m => MonadIO (ReaderT r m) --...
```

So:

```
main = runReaderT
         (ask >>= liftIO . putStrLn)
         "Test"
```

Monad Transformers

- Transformer stacks get used in real software:
- "xmonad" is a X11 window manager in Haskell; it defines a core monad:

```
-- | The X monad, 'ReaderT' and 'StateT'
-- transformers over 'IO' encapsulating the
-- window manager configuration and state,
-- respectively.
newtype X a = X (ReaderT XConf
                 (StateT XState IO) a)
```

Monad Transformers

A fistful of standard transformers:

- `ReaderT`
- `StateT`
- `WriterT` (accumulate results monoidally)
- `RWST` (Reader-Writer-State all in one)
- `MaybeT` (partial functions)
- `ErrorT` (pure throw/catch)
- `ContT` (continuations!)

(For the moment, these are provided by the `mtl` package in the Haskell Platform. There is at least one ongoing effort to improve and likely replace it.)

Monad Transformers

A brief word on stack ordering and effects:

- Consider `StateT`, and a transformer for (pure) exceptions, `ErrorT`.
- Two ways of stacking on top of `m`:
  - `StateT s (ErrorT e m)`
  - `ErrorT e (StateT s m)`

Monad Transformers

A brief word on stack ordering and effects:

- Consider `StateT`, and a transformer for (pure) exceptions, `ErrorT`.
- Two ways of stacking on top of `m`:
  - `StateT s (ErrorT e m)`
    - State backtracked when exception thrown.
    - i.e. catch handler runs with state as of the *start* of the wrapped computation.
  - `ErrorT e (StateT s m)`
    - State preserved when exception thrown.
    - catch handler runs with state changed by code up to the point of throw.

Monad Transformers

```
newtype ErrorT e m a = ErrorT
  { runErrorT :: m (Either e a) }
newtype StateT s m a = StateT
  { runStateT :: s -> m (a,s) }
```

- `StateT s (ErrorT e m)` – state backtracking:

```
runStateT :: s -> ErrorT e m (a,s)
-- runErrorT (runStateT act state)
```

- `ErrorT e (StateT s m)` – state preserving:

```
runErrorT :: StateT s m (Either e a)
-- runStateT (runErrorT act) state
```

Intro to Explicit Concurrency

The `Control.Concurrent` module and friends provide

- light-weight coroutine-style threads
- standard heavy-weight OS threads
- asynchronous exceptions
- inter-thread communication primitives
- (Bonus: the implementation abstracts over native event-driving mechanisms but presents straight-line code)

Intro to Explicit Concurrency

- How do we get these things?
- Primitive function:

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

- Takes the `RealWorld` and makes two of them.

Intro to Explicit Concurrency

- How do we get these things?
- Primitive function:

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

- Takes the `RealWorld` and makes two of them.
- Yes, that means that anything we share between them is subject to the laundry-list of race condition woes.

Intro to Explicit Concurrency
Race Conditions

- Race conditions?! Consider:

```
racer ref = forM_ [1..10000] $
  const $ modifyIORef ref (+1)
main = do
r <- newIORef 0
forkIO $ racer r
forkIO $ racer r
readIORef r >>= print
```

- Assuming both forked threads terminate before the `readIORef`, what does the last line print?

Intro to Explicit Concurrency
Race Conditions

- Real program in `Racer.hs`.
- Uses some stuff not yet discussed to ensure that the threads actually finish before printing.
- To actually run,

```
$ ghc --make -threaded -rtsopts Racer.hs
$ ./Racer +RTS -N2
```

- I ran the program a few times and got: 17973, 18724, 19263, 15035, 20000.

Intro to Explicit Concurrency
MVars

- Well that's no good.
- Classical answer: use a *lock* or *atomic* action.
  - In fact: `atomicModifyIORef`.
- An interesting Haskell answer: `MVar` in `Control.Concurrent.Mvar` (or just `Control.Concurrent`).
- Very similar to `IORefs`, but:

Intro to Explicit Concurrency
MVars

- Well that's no good.
- Classical answer: use a *lock* or *atomic* action.
    - In fact: `atomicModifyIORef`.
- An interesting Haskell answer: `MVar` in `Control.Concurrent.Mvar` (or just `Control.Concurrent`).
- Very similar to `IORefs`, but:
    - May be either *full* or *empty*.

Intro to Explicit Concurrency
MVars

- Well that's no good.
- Classical answer: use a *lock* or *atomic* action.
    - In fact: `atomicModifyIORef`.
- An interesting Haskell answer: `MVar` in `Control.Concurrent.Mvar` (or just `Control.Concurrent`).
- Very similar to `IORefs`, but:
    - May be either *full* or *empty*.
    - Taking an empty `MVar` blocks until somebody else puts.
    - Putting a full `MVar` blocks until somebody else takes.

Intro to Explicit Concurrency
MVars

- Well that's no good.
- Classical answer: use a *lock* or *atomic* action.
    - In fact: `atomicModifyIORef`.
- An interesting Haskell answer: `MVar` in `Control.Concurrent.Mvar` (or just `Control.Concurrent`).
- Very similar to `IORefs`, but:
    - May be either *full* or *empty*.
    - Taking an empty `MVar` blocks until somebody else puts.
    - Putting a full `MVar` blocks until somebody else takes.
    - Fair, depth-one producer/consumer queue.

Intro to Explicit Concurrency
MVars

- Core API:

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

- (take and put are fair: FIFO and wake-one)
- Non-blocking variants `tryTakeMVar` and `tryPutMVar`.
- Exception-safe utilities like

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

Intro to Explicit Concurrency
Other Explicitly-Concurrent Tools

Building up from `MVars`, there are

- `Chan`: Unbounded, MPMC channels.
- `QSem`: Semaphores with take-one/release-one.
- `QSemN`: Semaphores with take-many/release-many.
- `SampleVar`: overwritable MVars.
    - Take from empty still blocks.
    - Write to full overwrites.
    - Use for sampling (ah ha), progress indicators, ...

Software Transactional Memory

- There's a lot to be said about concurrency.
- I would rather talk about something newer than the same old stuff.
- You probably either have seen or will see the standard fare, which applies equally well to Haskell.

Software Transactional Memory

Hah, I'm stealing Simon Peyton Jones' excellent slides. [3]

Parallel Strategies

Your instructor steals again, this time using Andres Löh's excellent slides. [2]

Next time

- You tell me?

Bib

Mauro Jaskelioff.
Monatron: an extensible monad transformer library.
In *Implementation and Application of Functional Languages*, 2008.

Andres Löh.
Tutorial: Deterministic parallel programming in haskell, Oct 2011.
Available from: `http://www.well-typed.com/Hal6/Presentation.pdf`.

Simon Peyton Jones.
Haskell and transactional memory, April 2010.
Available from: `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/#tokyo`.

Tom Schrjvers and Bruno C. d. S. Oliveira.
Functional pearl: The monad zipper.
2010.
Available from: http://users.ugent.be/~tschrijv/Research/papers/monad_zipper_draft.pdf.