Haskell for all

Sunday, February 2, 2014

Streaming logging

I see many beginners use `WriterT [a]` in their programs to log outputs like this:

```
import Control.Monad.Trans.Class (lift)
import Control.Monad.Trans.Writer
import Prelude hiding (log)

log :: Monad m => String -> WriterT [String] m ()
log string = tell [string]

example :: WriterT [String] IO ()
example = do
    log "Printing 1 ..."
    lift $ print 1
    log "Printing 2 ..."
    lift $ print 2

main = do
    strings <- execWriterT example
    mapM_ putStrLn strings
```

This is not the best approach, because you cannot retrieve any logged values until the computation is complete:

```
>>> main
1
2
Printing 1 ...
Printing 2 ...
```

We cannot appropriate this for long-running programs like servers where we wish to inspect logged output while the program is still running. Worse, this approach will waste memory storing all logged values until the very end.

The simplest way to solve this is just to modify our computation to take the desired logging function as a parameter:

```
parametrize :: (String -> IO ()) -> IO ()
parametrize log = do
    log "Printing 1 ..."
    print 1
    log "Printing 2 ..."
    print 2

main = parametrize putStrLn
```

Now we log output immediately without wasting any memory:

```
>>> main
Printing 1 ...
1
Printing 2 ...
2
```

However, this approach is still a little brittle. For example, suppose we wish to log these lines to a file. As a basic denial-of-service precaution we might wish to cap the number of logged lines (or put the log file on a separate partition, but humor me). Limiting the logged output would necessitate the use of an `IORef` to coordinate between logging callbacks:

```
import Control.Monad (when)
import Data.IORef

main = do
    count <- newIORef 0

    let putStrLn' maxLines string = do
            n <- readIORef count
            when (n < maxLines) $ do
                putStrLn string
                writeIORef count (n + 1)

    parametrize (putStrLn' 1)
```

Now we have tightly integrated state into our log function and increased our dependence on `IO`. I prefer to limit unnecessary `IO` and also avoid callback hell, so I will introduce a third solution:

```
import Pipes
import Prelude hiding (log)

log :: Monad m => String -> Producer String m ()
log = yield

piped :: Producer String IO ()
piped = do
    log "Printing 1 ..."
    lift $ print 1
    log "Printing 2 ..."
    lift $ print 2

main = runEffect $ for piped (lift . putStrLn)
```

The `piped` code is syntactically identical to our original `example`, but this time we stream values immediately instead of deferring all results to a large list at the end:

```
>>> main
Printing 1 ...
1
Printing 2 ...
2
```

In fact, the `for` combinator from `Pipes` exactly recapitulates the behavior of our `parametrized` function. `(for p f)` replaces every `yield` in `p` with `f`, and `log` is just a synonym for `yield`, so we can freely substitute `log` commands using `for`. It's as if we had directly parametrized our `piped` function on the logging action:

```
for piped (lift . putStrLn)
```

```
-- Replace each `log`/`yield` with `(lift . putStrLn)`
= do (lift . putStrLn) "Printing 1 ..."
     lift $ print 1
     (lift . putStrLn) "Printing 2 ..."
     lift $ print 2

-- Simplify a little bit
= do lift $ putStrLn $ "Printing 1 ..."
     lift $ print 1
     lift $ putStrLn $ "Printing 2 ..."
     lift $ print 2

-- `lift` is a monad morphism, so we can factor it out
= lift $ do putStrLn $ "Printing 1 ..."
            print 1
            putStrLn $ "Printing 2 ..."
            print 2
```

... and all that `runEffect` does is remove the `lift`:

```
runEffect (for piped yield)

= runEffect $ lift $ do
     putStrLn $ "Printing 1 ..."
     print 1
     putStrLn $ "Printing 2 ..."
     print 2

-- runEffect (lift m) = m
= do putStrLn $ "Printing 1 ..."
     print 1
     putStrLn $ "Printing 2 ..."
     print 2
```

However, unlike the `parametrized` example, `piped` is more flexible. We can manipulate `yields` in many more ways than just the `for` combinator. For example, we can use the `take` `Pipe` from `Pipes.Prelude` to easily limit the number of logged outputs:

```
import qualified Pipes.Prelude as Pipes

limit :: Monad m => Int -> Pipe String String m r
limit n = do
    Pipes.take n  -- Forward the first `n` outputs
    Pipes.drain   -- Ignore the remaining log statements

main = runEffect $ for (piped >-> limit 1) (lift . putStrLn)
```

... or for people who prefer `(>->)` over `for`, you can write the entire thing as one long pipeline:

```
main = runEffect $ piped >-> limit 1 >-> Pipes.stdoutLn
```

This will now only output the first logged value:

```
>>> main
Printing 1 ...
1
2
```

We get all of this with a strict separation of concerns. All three stages in our pipeline are separable and reusable in the same spirit as Unix pipes.

So the next time you need to log a stream of values, consider using a

`Producer` to stream values immediately instead of building up a large list in memory. `Producers` preserve a great deal of flexibility with very few dependencies and low syntactic overhead. You can learn more about `pipes` by reading the tutorial.

Posted by Gabriel Gonzalez at 5:17 AM

6 comments:

Kloplop321 February 5, 2014 at 7:35 AM

You might want to tidy up your code. You have "Printing 2" followed by print 1

Reply

Replies

Gabriel Gonzalez February 5, 2014 at 9:13 PM

Fixed

Reply

Elliot March 28, 2014 at 2:44 PM

Can't you use Writer like this and get your logging to stream during computation?

`mapM_ putStrLn . execWriter $ forever (tell ["hello"])`

Reply

Replies

Gabriel Gonzalez April 9, 2014 at 5:42 PM

That will collect all results until the end of the computation and then print them. When I say streaming I mean that the results will immediately print the moment you log them.

Elliot July 9, 2014 at 1:59 PM

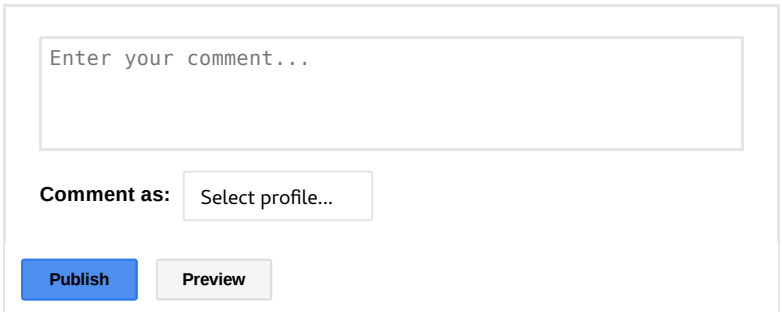
I don't understand since that example has no "end of the computation" and yet it repeatedly prints "hello" to the screen.

Gabriel Gonzalez July 28, 2014 at 9:16 PM

That only works for a pure `Writer` monad. For example, if you use `WriterT [String] IO` then it will not work.

Reply

Add comment



Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Blog Archive