Streaming Compression via Laziness

V. E. McHale

July 18, 2024

Abstract

Lazy data structures offer an elegant interface to streaming compression. However, compression libraries are implemented in C and work by effects (memory writes). Monadic I/O is the blessèd way to handle effects in Haskell, but can induce unwanted strictness in the resulting stream. We step through how to recover streaming via lazy lists for the LZ4 compression library, and show off some of the sophisticated facilities for interfacing C and Haskell. The case of streaming from a C API provides a nontrivial example of juggling effects and laziness.

1 Introduction

Compression libraries compress and decompress in constant memory, taking input in chunks. A nice API in Haskell would be:

```
compress, decompress
  :: [ByteString] -> [ByteString]
```

This would certainly be possible to write in pure Haskell—the resulting stream depends only on the input stream.

Typical C APIs, on the other hand, provide procedures with side effects. Haskell has elegant and capable support for working with effects, via the famous I/O monad [4][5]. However, using *only* the monadic interface to I/O makes the `(:)` constructor of the output list strict—a streaming failure.

2 I/O in Haskell

GHC defines `IO` by passing around a `RealWorld#`[1] token [2], viz.

```
newtype IO a =
  IO (RealWorld# -> (# RealWorld#, a #))
```

This is a monad. It forces sequencing by chaining flatness.

[1] The octothorpe `#` at the end of the identifier indicates it is flat; `(# _, _ #)` is a flat tuple.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(IO m) >>= g = IO (\s ->
    case m s of (# s', a #) ->
        (\(IO x) -> x) (g a) s')
```

The result (of type `IO b`) depends on the state token returned by `m` and the argument of type `a` passed to `g`—associated actions will be performed in order.

However, this definition admits more than just the monadic interface! For instance we can define:

```
unsafeInterleaveIO :: IO a -> IO a
unsafeInterleaveIO (IO f) = IO $ \s ->
    case f s of {(# _, a #) -> (# s, a #)}
```

This is immoral—it discards the `RealWorld#`—but it foreshadows the sort of state token manipulations that are possible.

2.1 ST Monad

The token passed to ensure ordering of effects need not be the `RealWorld#`; the `ST` monad [3] cleverly forbids state from escaping,[2] viz.

```
newtype ST s a =
  ST (State# s -> (State# s, a))
```

There is also a lazier variant,

```
newtype LazyST s a =
  LazyST (State s -> (State s, a))
```

where

```
data State s = State (State# s)
```

In this case state actions are deferred until the result is forced.

3 Modern C Bindings

We will use c2hs [1], a Haskell source preprocessor which reads C header files and generates imports with the corresponding Haskell types. We will also follow Edward Z Yang's advice to baptize `Ptr`s as `ForeignPtr`s so that we do not have to worry about explicitly `free`ing them later on [6]. This is especially welcome in our case, as it avoids the need to interleave calls to `free` while accounting for its side effects.

[2]`runST` has type `forall s. ST s a -> a` and so no part of the state `s` can be returned—the type variable is not in scope on the right side of the arrow!

4 LZ4 C API

Here is a simplified LZ4 C API:

```
typedef size_t LZ4F_errorCode_t;

typedef struct LZ4F_dctx_s LZ4F_dctx;
typedef LZ4F_dctx* LZ4F_decompressionContext_t;

typedef struct {
  ...
} LZ4F_decompressOptions_t;

LZ4F_errorCode_t
    LZ4F_createDecompressionContext
      (LZ4F_dctx** dctxPtr);
LZ4F_errorCode_t
    LZ4F_freeDecompressionContext(LZ4F_dctx* dctx);

size_t LZ4F_decompress(LZ4F_dctx* dctx,
    void* dstBuffer, size_t* dstSizePtr,
    const void* srcBuffer, size_t* srcSizePtr,
    const LZ4F_decompressOptions_t* dOptPtr);
```

To decompress data:

1. Create a decompression context `dctx`.
2. Feed a chunk of input data to `LZ4F_decompress` and read out decompressed data (`srcBuffer` and `srcSizePtr`). The return value, of type `size_t` tells us how many bytes of input were decompressed; we may need to feed it to the decompressor incrementally.
3. Repeat (2) with each remaining chunk of our data.
4. Free the decompression context.

4.1 C2hs

The corresponding c2hs:

```
type LZ4FErrorCode = CSize
{# typedef LZ4F_errorCode_t LZ4FErrorCode #}

data LzDecompressionCtx

{# pointer *LZ4F_dctx as LzDecompressionCtxPtr
    foreign finalizer LZ4F_freeDecompressionContext
```

```
        as ^
    -> LzDecompressionCtx #}

{# fun LZ4F_createDecompressionContext as ^
    { alloca- ‘Ptr LzDecompressionCtx’ peek*
    } -> ‘LZ4FErrorCode’
  #}

data LzDecompressOptions

{# pointer *LZ4F_decompressOptions_t
    as LzDecompressOptionsPtr
  -> LzDecompressOptions #}

{# fun LZ4F_decompress as ^
    { ‘LzDecompressionCtxPtr’
    , castPtr ‘Ptr a’
    , castPtr ‘Ptr CSize’
    , castPtr ‘Ptr b’
    , castPtr ‘Ptr CSize’
    , ‘LzDecompressOptionsPtr’
    } -> ‘CSize’ coerce
  #}
```

This uses several advanced features worth mentioning. First,

```
{# pointer *LZ4F_dctx as LzDecompressionCtxPtr
    foreign finalizer LZ4F_freeDecompressionContext
        as ^
    -> LzDecompressionCtx #}
```

defines `LzDecompressionCtxPtr` on the Haskell side as

```
type LzDecompressionCtxPtr
    = ForeignPtr LzDecompressionCtx
```

C2hs will generate code so that functions taking a `*LZ4F_dctx` as an argument have a wrapper taking a `ForeignPtr` on the Haskell side. It also generates code to import
`LZ4F_freeDecompressionContext` as a finalizer.

```
{# fun LZ4F_createDecompressionContext as ^
    { alloca- ‘Ptr LzDecompressionCtx’ peek*
    } -> ‘LZ4FErrorCode’
  #}
```

also deserves commentary. The C procedure
`LZ4F_createDecompressionContext` uses destination-passing style to be able

to return both an error and a pointer to the newly created compressor. However, Haskell has tuples, a more fluent way of returning multiple values. C2hs will generate a wrapper for us of type:

```
lZ4FCreateCompressionContext ::
    IO (LZ4FErrorCode, Ptr LzCtx)
```

4.2 First Cut

The following uses the C API to decompress chunks into a list of `ByteString`s. It takes place in the `IO` monad, passing around a `RealWorld#` token, because this is what GHC supports for foreign function calls.[3]

```
decompress :: [ByteString]
           -> [ByteString]
decompress bss = unsafePerformIO $ do
    (ctx, buf) <- do
        (_, preCtx) <- lZ4FCreateDecompressionContext
        ctx <- castForeignPtr
            <$> newForeignPtr lZ4FFreeCompressionContext
                    (castPtr preCtx)
        dstBuf <- mallocForeignPtrBytes bufSz
        pure (ctx, dstBuf)
    loop ctx buf bss

  where
    bufSz :: Integral a => a
    bufSz = 32 * 1024

    loop :: LzDecompressionCtxPtr
         -> ForeignPtr a
         -> [BS.ByteString]
         -> IO [BS.ByteString]
    loop _ _ [] = pure []
    loop ctx buf (b:bs') = do
        (nxt, res) <- stepChunk ctx buf b
        case nxt of
            Nothing   -> (res:) <$> loop ctx buf bs'
            Just next -> (res:) <$> loop ctx buf (next:bs')

    stepChunk :: LzDecompressionCtxPtr
              -> ForeignPtr a
              -> BS.ByteString
              -> IO ( Maybe BS.ByteString
```

[3]C2hs also requires `IO`; for instance in wrapping the destination-passing style it enlists `alloca :: Storable a => (Ptr a -> IO b) -> IO b`.

```
                 , BS.ByteString
                 )
stepChunk !ctx !dst b =
  BS.unsafeUseAsCStringLen b $ (buf, sz) ->
    withForeignPtr dst $ \d ->
      alloca $ \dSzPtr ->
        alloca $ \szPtr -> do
          poke dSzPtr (fromIntegral bufSz)
          poke szPtr (fromIntegral sz)
          _ <- lZ4FDecompress
              ctx d dSzPtr buf szPtr nullPtr
          bRead <- fromIntegral <$> peek szPtr
          bWritten <- peek dSzPtr
          outBs <- BS.packCStringLen
              (castPtr d, fromIntegral bWritten)
          let remBs = if bRead == sz
              then Nothing
              else Just (BS.drop bRead b)
          pure (remBs, outBs)
```

We see from profiling information in Figure 1 that streaming fails.

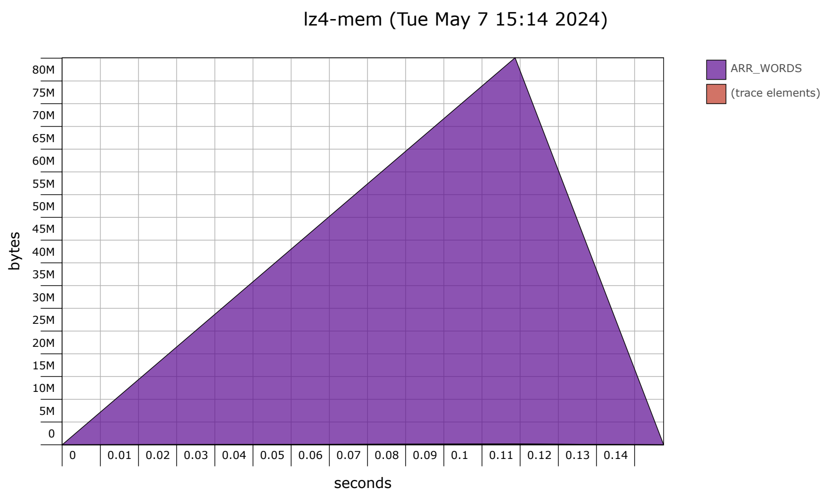

Figure 1: Naïve loop

This makes sense; `(:)` occurs modulo the `IO` monad—evaluating even the head of the resulting list means performing all effects. The decompressor will loop through all steps, i.e. all input will be fed in before the first chunk is read off.

To get streaming behavior we shoehorn the procedure into the `LazyST` monad.

As we shall see, the LazyST monad works well with the ST monad; we can keep islands of strict effect management embedded in the LazyST monad.

4.3 Streaming Compression

```
decompress :: [BS.ByteString]
           -> [BS.ByteString]
decompress bss = runST $ do
    (ctx, buf) <- LazyST.unsafeIOToST $ do
        (_, preCtx) <- lZ4FCreateDecompressionContext
        ctx <- castForeignPtr
            <$> newForeignPtr lZ4FFreeCompressionContext
                    (castPtr preCtx)
        dstBuf <- mallocForeignPtrBytes bufSz
        pure (ctx, dstBuf)
    loop ctx buf bss
  where
    bufSz :: Integral a => a
    bufSz = 32 * 1024

    loop :: LzDecompressionCtxPtr
          -> ForeignPtr a
          -> [BS.ByteString]
          -> LazyST.ST s [BS.ByteString]
    loop _ _ [] = pure []
    loop ctx buf (b:bs') = do
          (nxt, res) <- stepChunk ctx buf b
          case nxt of
              Nothing   -> (res:) <$> loop ctx buf bs'
              Just next -> (res:) <$> loop ctx buf (next:bs')

    stepChunk :: LzDecompressionCtxPtr
              -> ForeignPtr a
              -> BS.ByteString
              -> LazyST.ST s ( Maybe BS.ByteString
                             , BS.ByteString
                             )
    stepChunk !ctx !dst b = LazyST.unsafeIOToST $
      BS.unsafeUseAsCStringLen b $ (buf, sz) ->
        withForeignPtr dst $ \d ->
          alloca $ \dSzPtr ->
            alloca $ \szPtr -> do
              poke dSzPtr (fromIntegral bufSz)
              poke szPtr (fromIntegral sz)
              _ <- lZ4FDecompress
                  ctx d dSzPtr buf szPtr nullPtr
```

```
                bRead <- fromIntegral <$> peek szPtr
                bWritten <- peek dSzPtr
                outBs <- BS.packCStringLen
                    (castPtr d, fromIntegral bWritten)
                let remBs = if bRead == sz
                    then Nothing
                    else Just (BS.drop bRead b)
                pure (remBs, outBs)
```

The unsafeIOToST deserves comment. It is defined as:

```
unsafeIOToST :: IO a -> LazyST s a
unsafeIOToST = strictToLazyST . unsafeIOToST
```

unsafeIOToST converts an IO action to a strict ST action; both use state token-passing and we coerce a RealWorld# into a State# s.

strictToLazyST is more interesting. It embeds strict ST actions in the LazyST monad—sequences of actions can be chained as dependencies, which will be performed all together when the final result is demanded in the LazyST monad.

```
strictToLazyST :: ST s a -> LazyST s a
strictToLazyST (ST m) = LazyST $ \(State s) ->
  case m s of
    (# s', a #) -> (a, State s')
```

This takes a State# s and wraps it in a State; the ST action will only be performed when its result is demanded. However, since the state token is obtained by wrapping the last state token of the strict ST action, evaluating the result in the LazyST monad will perform all effects defined in the strict ST monad.

Consider our own code:

```
LazyST.unsafeIOToST $ do
    ...
    _ <- lZ4FDecompress
        ctx d dSzPtr buf szPtr nullPtr
    bRead <- fromIntegral <$> peek szPtr
    bWritten <- peek dSzPtr
    outBs <- BS.packCStringLen (castPtr d, fromIntegral bWritten)
```

This cannot take place wholly in the LazyST monad; the result of lZ4FDecompress is never demanded but we need to guarantee it is called before outBSs is read. On the other hand, by converting the block to a LazyST action, none of the effects will take place until outBs is demanded. We have placed LazyST.unsafeIOToST precisely at stepChunk above.

Thus, inspecting the first decompressed chunk will entail one call to lZ4FDecompress, not the whole loop.

And in fact, this works: we can see from profiling information (Figure 2) that memory use has gone from megabytes to kilobytes.

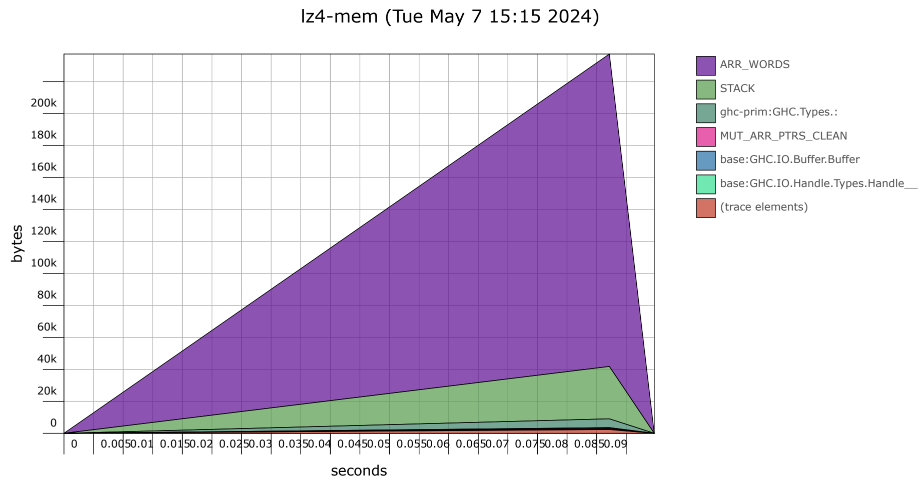

Figure 2: Streaming in the lazy ST monad

5 Acknowledgments

The above is not new. Duncan Coutts used lazy `ByteString`s for compression APIs in 2006, and the use of the `LazyST` monad comes from studying Herbert Valerio Riedel's code.

References

[1] Manuel M. T. Chakravarty. C $\longrightarrow$haskell, or yet another interfacing tool. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, pages 131–148, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[2] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES*, 180:47–96, 2001.

[3] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. *SIGPLAN Not.*, 29(6):24–35, jun 1994.

[4] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[5] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[6] Edward Z Yang. Principles of ffi api design. `http://blog.ezyang.com/2010/06/principles-of-ffi-api-design/`, 2010.