COMS W4995
hzip - Parallel gzip in Haskell

Annie F Song

December 22, 2021

1 Introduction

gzip is a popular compression algorithm and format. Developed by Mark Adler and Jean-loup Gailly and first released in 1992, it stands as one of the most popular compression schemes in the world.

2 Background

At the core of gzip is the DEFLATE algorithm. Specified in RFC 1951, DEFLATE algorithm is a combination of the LZ77 compression algorithm and Huffman Encoding.

DEFLATE starts by figuratively separating the file into blocks. Each block can be compressed using one of three compression strategies. Block type 0 means that the block will remain uncompressed, a scheme which is useful for files that are already compressed. Block type 1 uses the LZ77 algorithm to reduce redundancy and a static Huffman encoding to encode the result. Block type 2 also uses the LZ77 algorithm, but uses a dynamically generated Huffman encoding to encode the result. The dynamically generated Huffman encoding will also be encoded into the block. These compressed blocks will then be written out to the compressed file, resulting in a smaller file.

3 Parallel Haskell Implementation

I was not able to find any Haskell implementation of the gzip compression algorithm online. I was able to find some decompression algorithms,[1] but any implementation of the compression scheme invoked the zlib C library directly.[2]

In my project, I've implemented a simplified version of gzip. My implementation supports block type 0 and block type 1. I was half way through implementing block type 2 (with dynamic Huffman tree generation portion done) when I ran out of time.

My implementation also supports parallelization. Similar to the parallel implementation of gzip (also known as pigz), developed by Mark Adler,[3] I parallelized the compression of each block.

[1] https://hackage.haskell.org/package/pure-zlib

[2] https://hackage.haskell.org/package/zlib-0.6.2.3/docs/Codec-Compression-GZip.html

[3] https://zlib.net/pigz/

4 Methods

The following tests were run on my personal desktop machine, running Ubuntu-20.04.2 with Linux kernel 5.11.0-43-generic. My CPU is Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, with 6 cores and 12 processors.

My tests were run on various corpora such as the Canterbury Corpus and the Large Corpus.[4]

5 Results

Each of the following tables represents a different corpus. Each file was compressed twice, one using sequential implementation (with no parallelization flags enabled) and one using parallel implementation (with parallelization flags enabled).

Table 1: The Artificial Corpus

File	Size (B)	Compressed Size (B)	Compression Ratio	Seq Time (s)	Par Time (s)
a.txt	1	21	2100.000 %	0.266	0.278
aaa.txt	100000	1420	1.420 %	3.041	0.895
alphabet.txt	100000	1869	1.869 %	0.465	0.360
random.txt	100000	99706	99.706 %	5.820	2.326

Table 2: The Calgary Corpus

File	Size (B)	Compressed Size (B)	Compression Ratio	Seq Time (s)	Par Time (s)
bib	111261	65511	58.880 %	2.439	1.477
book1	768771	520342	67.685 %	17.130	6.176
book2	610856	353255	57.830 %	11.713	5.197
geo	102400	88764	86.684 %	3.656	2.341
news	377109	228641	60.630 %	8.405	3.798
obj1	21504	11996	55.785 %	0.829	0.810
obj2	246814	112378	45.531 %	4.404	2.527
paper1	53161	29862	56.173 %	1.215	1.253
paper2	82199	49512	60.234 %	1.847	1.203
paper3	46526	28982	62.292 %	1.163	0.811
paper4	13286	7991	60.146 %	0.538	0.510
paper5	11954	6877	57.529 %	0.486	0.427
paper6	38105	21089	55.344 %	0.951	0.715
pic	513216	72232	14.074 %	7.415	2.754
progc	39611	20235	51.084 %	0.953	0.737
progl	71646	26771	37.366 %	1.121	0.999
progp	49379	18681	37.832 %	0.861	0.599
trans	93695	41138	43.906 %	1.572	1.267

[4] https://corpus.canterbury.ac.nz/descriptions/

Table 3: The Canterbury Corpus

File	Size (B)	Compressed Size (B)	Compression Ratio	Seq Time (s)	Par Time (s)
alice29.txt	152089	90967	59.812 %	3.146	2.276
asyoulik.txt	125179	79235	63.297 %	2.787	1.534
cp.html	24603	12352	50.205 %	0.738	0.570
fields.c	11150	4294	38.511 %	0.375	0.531
grammar.lsp	3721	1489	40.016 %	0.321	0.441
kennedy.xls	1029744	293751	28.527 %	16.122	5.415
lcet10.txt	426754	251592	58.955 %	8.173	4.599
plrabn12.txt	481861	323802	67.198 %	10.394	4.737
ptt5	513216	72232	14.074 %	7.431	3.211
sum	38240	18324	47.918 %	1.102	0.931
xargs.1	4227	2163	51.171 %	0.327	0.434

Table 4: The Large Corpus

File	Size (B)	Compressed Size (B)	Compression Ratio	Seq Time (s)	Par Time (s)
E.coli	4638690	2079253	44.824 %	59.749	26.503
bible.txt	4047392	2069616	51.135 %	64.402	29.776
world192.txt	2473400	1511588	61.114 %	54.467	27.370

Table 5: The Miscellaneous Corpus

File	Size (B)	Compressed Size (B)	Compression Ratio	Seq Time (s)	Par Time (s)
pi.txt	1000000	779216	77.922 %	21.599	11.314

6 Analysis

Comparing the sequential and parallel results, we see that parallelization did improve performance. This is expected given that 12 CPUs shared the work. The speed-up was most noticeable when compressing large files, such as the bible, which cut down the time by more than half.

In some scenarios, however, parallelization did not prove to be worthy. This was specially the case for smaller files. This was probably due to the fact that there was not much work to be shared, and extra overhead of context-switching and thread creation slowed down the parallel implementation.

The results also show when compression should not be used. When we started with a small file, such as a.txt, we ended up with a bigger file because we had to add headers and footers following the DEFLATE algorithm. This bloated the size from 1 byte to 21 bytes. This, however, should be considered as an exception, given that we usually compress large files.

It should be noted that when although the runtime may differ, the compression ratio remained the same. This is because compression ratio does not depend on how many threads are doing the work. Compression ratio is related to the Huffman encoding and LZ77 compression algorithm, both of which remained the same for the two runs.

Also attached in the appendix is ThreadScope analysis generated during the sample run while compressing bible.txt. It it clear that all CPUs are busy. Of the 989 sparks generated, only 1 fizzled and the rest were converted.

7 Future Direction

I've thought of some ways to improve the project in the future. First, I can finish block type 2 implementation. The next thing I can improve is the performance of the LZ77 algorithm. Currently it uses a naive approach without any sort of maps, which slows down the runtime by quite a lot. CRC32 calculation is done by leveraging the C library, so it is also another area where I can improve this project to make `hzip` pure Haskell.

8 Conclusion

In the report I presented `hzip`, a parallel Haskell implementation of the renowned `gzip` program.

A Appendix: ThreadScope

Figure 1: ThreadScope

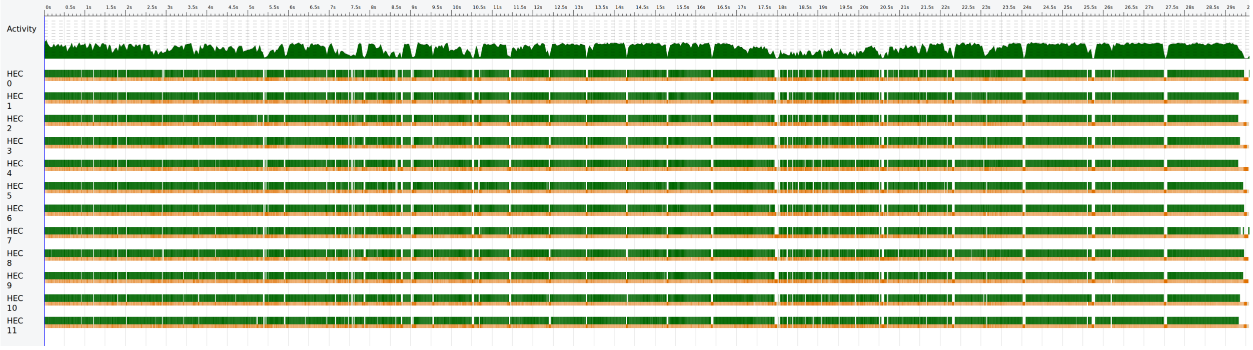
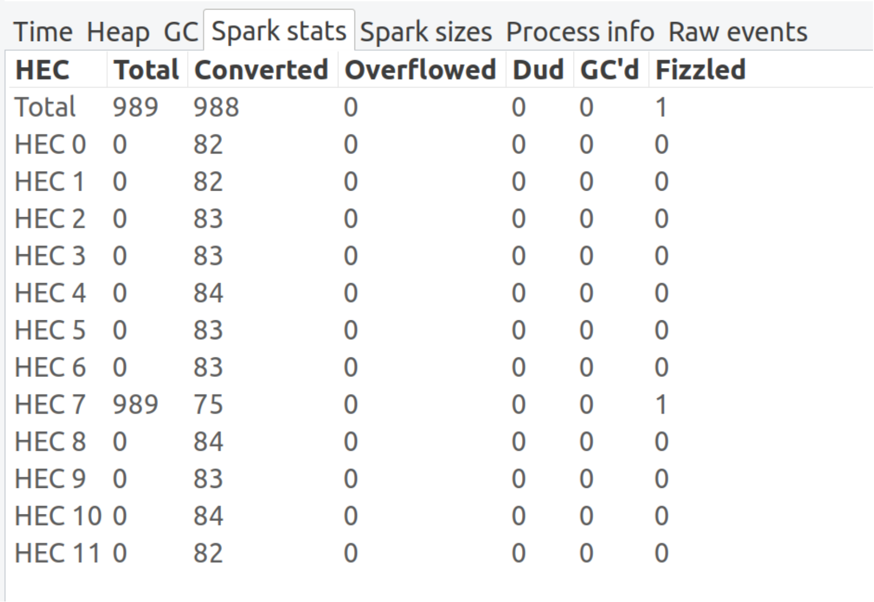


Figure 2: Sparks



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	989	988	0	0	0	1
HEC 0	0	82	0	0	0	0
HEC 1	0	82	0	0	0	0
HEC 2	0	83	0	0	0	0
HEC 3	0	83	0	0	0	0
HEC 4	0	84	0	0	0	0
HEC 5	0	83	0	0	0	0
HEC 6	0	83	0	0	0	0
HEC 7	989	75	0	0	0	1
HEC 8	0	84	0	0	0	0
HEC 9	0	83	0	0	0	0
HEC 10	0	84	0	0	0	0
HEC 11	0	82	0	0	0	0

B Appendix: Code

Main.hs

```haskell
module Main where

import Lib ( writeOut, parCompress )
import System.Environment (getArgs)
import System.IO.Error
  ( catchIOError
  , ioeGetFileName
  , isDoesNotExistError
  , isPermissionError
  , isUserError
  )

main :: IO ()
main = mainLogic `catchIOError` handler

mainLogic :: IO ()
mainLogic = do
  [filename] <- getArgs
  compressed <- parCompress filename
  writeOut (filename ++ ".gz") compressed
  return ()

handler :: IOError -> IO ()
handler e
  | isDoesNotExistError e = putStrLn $ fn ++ ": No such file or directory"
  | isPermissionError e = putStrLn $ fn ++ ": Permission denied"
  | isUserError e = putStrLn "Usage: ./hzip <filename>"
  | otherwise = ioError e
  where
    Just fn = ioeGetFileName e
```

Lib.hs

```haskell
module Lib
  ( zlibCompress,
    parCompress,
    seqCompress,
    writeOut,
    getHeader,
    getFooter,
  )
where

import qualified Block as B
import qualified Codec.Compression.GZip as GZip
import qualified Data.ByteString.Lazy as LBS
import qualified Data.Functor((<&>))

-- write the output to a given filename
writeOut :: String -> LBS.ByteString -> IO ()
writeOut = LBS.writeFile

-- use zlib for compression
zlibCompress :: String -> IO LBS.ByteString
zlibCompress fname = LBS.readFile fname Data.Functor.<&> GZip.compress

-- sequential implementation
seqCompress :: String -> IO LBS.ByteString
seqCompress fname =
  LBS.readFile fname >>= (\f -> return (LBS.concat [getHeader, B.doSeqCompress f]))

parCompress :: String -> IO LBS.ByteString
parCompress fname =
  LBS.readFile fname >>= (\f -> return (LBS.concat [getHeader, B.doParCompress f]))

emptyCompress :: IO LBS.ByteString
emptyCompress = return $ LBS.concat [getHeader, x, getFooter]
  where
    x = LBS.pack [0x01, 0x00, 0x00, 0xff, 0xff]

-- add the 10-byte header for .gz files
getHeader :: LBS.ByteString
getHeader = LBS.pack [0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03]

-- add the 8-byte footer for .gz files
getFooter :: LBS.ByteString
getFooter = LBS.pack [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
```

LZ77.hs

```haskell
module LZ77 where

import BitHelper
import qualified Data.ByteString.Lazy as LBS
import qualified Data.List as L
import qualified Data.Word as W

-- The result is either a (length, distance) pair
-- or 8-bit characters representing the string value
type Result = (Int, Int, W.Word8)
type MResult = Maybe Result

-- minimum matching length as described in RFC 1951.
minMatchLength :: Int
minMatchLength = 3

windowLength :: Int
windowLength = 10000

getFileContent :: String -> IO LBS.ByteString
getFileContent  = LBS.readFile

{-
do lz77 compression
-}
lz77Compress :: LBS.ByteString -> [MResult]
lz77Compress content = doLZ77 [] (LBS.unpack content)

doLZ77 :: [W.Word8] -> [W.Word8] -> [MResult]
doLZ77 buffer str
  | null str = [Nothing]
  | otherwise = Just res : doLZ77 newBuffer newStr
  where
    res@(l, d, c) = search buffer str
    matchLen = if l == 0 then 1 else l
    (matched, newStr) = splitAt matchLen str
    tempBuffer = buffer ++ matched
    newBuffer = drop (max 0 (length tempBuffer - windowLength)) tempBuffer

search :: [W.Word8] -> [W.Word8] -> (Int, Int, W.Word8)
search buffer str
  | null str = error "this shouldn't happen"
  | null buffer = (0, 0, fromIntegral $ head str)
  | otherwise = (len, dist, fromIntegral nextChar)
  where
    searchStr = take 258 str
    (len, dist) = findBuf buffer searchStr
    nextChar = head str

{-
Given a buffer and a needle, return the (length,distance) pair
returns the longest input that begins in the buffer

Length has to be a minimum of 3 characters following DEFLATE convention
-}
findBuf :: [W.Word8] -> [W.Word8] -> (Int, Int)
findBuf buffer str
  | null buffer || null str = (0, 0)
  | otherwise = comPair (len, dist) temp
  where
    mLen = prefixMatch buffer str
    len = if mLen >= minMatchLength then mLen else 0
    dist = if len > 0 then length buffer else 0
    temp = findBuf (drop 1 buffer) str

prefixMatch :: [W.Word8] -> [W.Word8] -> Int
prefixMatch [] _ = 0
prefixMatch _ [] = 0
prefixMatch (x:xs) (y:ys)
  | x == y = 1 + prefixMatch xs ys
  | otherwise = 0

{-
Compares two (len,dist) pairs and returns the more optimal pair.
-}
comPair :: (Int, Int) -> (Int, Int) -> (Int, Int)
comPair (llen, ldist) (rlen, rdist)
  | llen > rlen = (llen, ldist)
  | llen == rlen = if ldist < rdist then (llen, ldist) else (rlen, rdist)
  | otherwise = (rlen, rdist)
```

Block.hs

```haskell
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
module Block
  ( doSeqCompress,
    doParCompress,
    combineCrc,
    combine,
    splitUp,
  )
where

import BitHelper (word16ToLBS, word32ToLBS, wordsToBits, bitsToLBS)
import qualified Control.Parallel.Strategies as S
import qualified Data.Bits as Bits
import qualified Data.ByteString.Lazy as B
import qualified Data.List as L
import qualified Data.Word as W
import Deflate (deflate)
import LZ77 (lz77Compress)
import MyCRC32 (CRC32 (crc32), crc32Combine)
import qualified GHC.Generics as S

-- Three types of data blocks
-- Uncompressed, using static tree, using dynamic tree
data BlockType
  = Uncompressed
  | Static
  | Dynamic
  deriving (Eq)

data InputBlock = InputBlock
  { iType :: BlockType,
    iLast :: Bool,
    iData :: B.ByteString
  }

data OutputBlock = OutputBlock
  { oData :: [Bool],
    oCrc :: W.Word32,
    oLen :: W.Word32
  } deriving (S.NFData, S.Generic)

splitUp :: B.ByteString -> [InputBlock]
splitUp s
  | B.length t == 0 = [InputBlock Static True h]
  | otherwise = InputBlock Static False h : splitUp t
  where
```

```
    (h, t) = B.splitAt 4096 s

combine :: [OutputBlock] -> B.ByteString
combine blocks = B.concat [bitsToLBS contents, fcrc, flen]
  where
    contents = concatMap oData blocks
    (len, pairs) =
      L.mapAccumL (\a x -> (a + oLen x, (oCrc x, oLen x))) 0 blocks
    (tcrc, tlen) = foldl1 combinePair pairs
    flen = word32ToLBS tlen
    fcrc = word32ToLBS tcrc

combinePair ::
  (W.Word32, W.Word32) -> (W.Word32, W.Word32) -> (W.Word32, W.Word32)
combinePair p1 p2 = (newCrc, newLen)
  where
    (crc1, len1) = p1
    (crc2, len2) = p2
    newCrc = combineCrc crc1 p2
    newLen = combineLen len1 len2

-- TODO: Do this math in haskell
-- CRC32(AB) = CRC32(A0) ^ CRC32(0B) = CRC32(A0) ^ CRC32(B)
-- https://stackoverflow.com/questions/23122312/crc-calculation-of-a-mostly-static-data-stream
combineCrc :: W.Word32 -> (W.Word32, W.Word32) -> W.Word32
combineCrc crc1 pair = crc32Combine crc1 crc2 (fromIntegral len2)
  where
    (crc2, len2) = pair

combineLen :: W.Word32 -> W.Word32 -> W.Word32
combineLen len1 len2 = len1 + len2

doParCompress :: B.ByteString -> B.ByteString
doParCompress s = combine blocks
  where
    chunks = splitUp s
    blocks = map doCompress' chunks `S.using` S.parList S.rdeepseq

doSeqCompress :: B.ByteString -> B.ByteString
doSeqCompress s = combine $ map doCompress' $ splitUp s

doCompress' :: InputBlock -> OutputBlock
doCompress' i
  | t == Uncompressed = getUncompressed i
  | t == Static = getStatic i
  | t == Dynamic = getDynamic i
  | otherwise = error "invalid block type"
  where
```

```
    t = iType i

getUncompressed :: InputBlock -> OutputBlock
getUncompressed iblock = OutputBlock hc crc (fromIntegral l)
  where
    content = iData iblock
    crc = S.runEval $ S.rseq $ crc32 content
    header = [iLast iblock, False, False] ++ replicate 5 False
    l = (fromIntegral $ B.length content) :: W.Word16
    cl = Bits.complement l
    hc = reverse header ++ comp
    comp = wordsToBits (B.unpack $ B.concat [word16ToLBS l, word16ToLBS cl, content])


getStatic :: InputBlock -> OutputBlock
getStatic iblock = OutputBlock (bits ++ output) crc (fromIntegral l)
  where
    content = iData iblock
    crc = crc32 content
    isFinal = iLast iblock
    l = (fromIntegral $ B.length content) :: W.Word16
    bits = isFinal : [True, False]
    output = deflate content

getDynamic :: InputBlock -> OutputBlock
getDynamic _ = error "unimplemented"
```

BitHelper.hs

```haskell
module BitHelper where

import Data.Bits (FiniteBits)
import qualified Data.ByteString.Builder as Builder
import qualified Data.ByteString.Lazy as B
import qualified Data.List.Split as LS
import qualified Data.Word as W
import Table (distTable, litLenTable, llHuffmanCode)

word32ToLBS :: W.Word32 -> B.ByteString
word32ToLBS w = Builder.toLazyByteString $ Builder.word32LE w

word16ToLBS :: W.Word16 -> B.ByteString
word16ToLBS w = Builder.toLazyByteString $ Builder.word16LE w

bitsToLBS :: [Bool] -> B.ByteString
bitsToLBS bits = B.pack $ map bitsToWord8 (LS.chunksOf 8 bits)

{-
Given a list of bits (at most 8 bits),
convert to word8
-}
bitsToWord8 :: [Bool] -> W.Word8
bitsToWord8 bits
  | n > 8 = error "too long"
  | otherwise = foldl (\a f -> 2 * a + if f then 1 else 0) 0 (reverse padded)
  where
    n = length bits
    padded = replicate (8 - n) False ++ bits

wordsToBits :: [W.Word8] -> [Bool]
wordsToBits = concatMap (\f -> wordToBits (f, 8))

{-
Given the code and expected length in bits, convert into bits
of expected length
-}
wordToBits :: (Integral a) => (a, W.Word8) -> [Bool]
wordToBits (code, len) = replicate pLen False ++ bits
  where
    bits = wordToBits' code
    pLen = fromIntegral len - length bits

wordToBits' :: (Integral a) => a -> [Bool]
wordToBits' 0 = []
wordToBits' x = wordToBits' (x `div` 2) ++ [x `rem` 2 == 1]
```

```
{-
Given a ll code, we need to convert it into bits
if less than 256, convert to literal
if more than 256, convert to length + extra bits
-}
litToBits :: W.Word16 -> [Bool]
litToBits = wordToBits . llHuffmanCode

lenToBits :: W.Word16 -> [Bool]
lenToBits l = litToBits code ++ reverse (wordToBits (l - base, extra))
  where
    (code, extra, base) = litLenTable l

distToBits :: W.Word16 -> [Bool]
distToBits d = wordToBits (code, 5) ++ reverse (wordToBits (d - base, extra))
  where
    (code, extra, base) = distTable d
```

HuffmanTree.hs

```haskell
{-# LANGUAGE TupleSections #-}

module HuffmanTree where

import BitHelper (wordToBits)
import Data.ByteString.Lazy (ByteString, unpack)
import Data.Function (on)
import Data.List (insertBy, sort, sortBy)
import Data.Map (Map, fromList, fromListWith, toList)
import Data.Maybe (fromJust, isNothing)
import Data.Tuple (swap)
import Data.Word (Word8)

data HuffTree = HuffLeaf Word8 Int | HuffNode HuffTree HuffTree Int deriving (Show)

type EncDict = Map Word8 [Bool]

weight :: HuffTree -> Int
weight (HuffLeaf _ w) = w
weight (HuffNode _ _ w) = w

toFreqList :: ByteString -> [(Word8, Int)]
toFreqList bs = toList $ fromListWith (+) $ map (,1) $ unpack bs

mergeTrees :: HuffTree -> HuffTree -> HuffTree
mergeTrees f s = HuffNode f s (weight f + weight s)

construct :: [(Word8, Int)] -> HuffTree
construct ts = construct' $ map (uncurry HuffLeaf) (sortBy (compare `on` snd) ts)

construct' :: [HuffTree] -> HuffTree
construct' [] = error "empty huffman tree"
construct' [t] = t
construct' (f : s : xs) = construct' $ insertBy (compare `on` weight) (mergeTrees f s) xs

-- Given a huffman tree, build bl_count in RFC 1951
buildBitLen :: HuffTree -> [(Int, [Word8])]
buildBitLen tree = toList $ fromListWith (++) $ buildBitLen' tree 0

buildBitLen' :: HuffTree -> Int -> [(Int, [Word8])]
buildBitLen' (HuffNode a b _) blen = buildBitLen' a (blen + 1) ++ buildBitLen' b (blen + 1)
buildBitLen' (HuffLeaf c _) blen
  | blen <= 15 = [(blen, [c])]
  | otherwise = error "unsupported yet"

buildEncTree :: [(Int, [Word8])] -> EncDict
buildEncTree map = fromList $ buildEncTree' 1 0 map

{-
blen -> number of bits
start -> start point for these bits
prev -> number of elements in the previous bit width
map -> bl_count (bit len -> bits)
-}
buildEncTree' :: Int -> Int -> [(Int, [Word8])] -> [(Word8, [Bool])]
buildEncTree' 16 _ _ = []
buildEncTree' blen start map = curr ++ buildEncTree' (blen + 1) newStart map
  where
    entries = lookup blen map
    curr = maybe [] (\f -> genCodeForBitLen f blen start) entries
    count = maybe 0 length entries
    newStart = (start + count) * 2

genCodeForBitLen :: [Word8] -> Int -> Int -> [(Word8, [Bool])]
genCodeForBitLen words bitlen start = zip swords bits
  where
    swords = sort words
    nums = [start .. (start + length swords)]
    bits = map (wordToBits . (,fromIntegral bitlen)) nums
```

Deflate.hs

```haskell
module Deflate
  ( deflate,
  )
where

import BitHelper (lenToBits, litToBits, distToBits)
import qualified Data.ByteString.Lazy as B
import LZ77 (MResult, lz77Compress)

deflate :: B.ByteString -> [Bool]
deflate input = translate $ lz77Compress input

dummyCompress :: B.ByteString -> [MResult]
dummyCompress input = map (\f -> Just (0,0,f)) (B.unpack input) ++ [Nothing]

translate :: [MResult] -> [Bool]
translate [] = error "empty results, shouldn't happen"
translate [Nothing] = replicate 7 False
translate (Just (l, d, c) : rs)
  | l == 0 = litToBits (fromIntegral c) ++ translate rs
  | otherwise = lenToBits (fromIntegral l) ++ distToBits (fromIntegral d) ++ translate rs
translate _ = error "this should not be possible"
```

Table.hs

```
module Table
  ( litLenTable,
    llHuffmanCode,
    distTable,
  )
where

import qualified Data.Word as W

{-
Block type 1 has static trees that are defined in RFC 1951.
This file contains those trees.
-}

{-
Value 0-256 can be converted to bits as is
Value 257-285 must be looked up according to the table
Given a length, return (code, num of extra bits, base)
-}
litLenTable :: W.Word16 -> (W.Word16, W.Word8, W.Word16)
litLenTable ll
  | ll < 3 = error "too small"
  | ll == 3 = (257, 0, 3)
  | ll == 4 = (258, 0, 4)
  | ll == 5 = (259, 0, 5)
  | ll == 6 = (260, 0, 6)
  | ll == 7 = (261, 0, 7)
  | ll == 8 = (262, 0, 8)
  | ll == 9 = (263, 0, 9)
  | ll == 10 = (264, 0, 10)
  | ll <= 12 = (265, 1, 11)
  | ll <= 14 = (266, 1, 13)
  | ll <= 16 = (267, 1, 15)
  | ll <= 18 = (268, 1, 17)
  | ll <= 22 = (269, 2, 19)
  | ll <= 26 = (270, 2, 23)
  | ll <= 30 = (271, 2, 27)
  | ll <= 34 = (272, 2, 31)
  | ll <= 42 = (273, 3, 35)
  | ll <= 50 = (274, 3, 43)
  | ll <= 58 = (275, 3, 51)
  | ll <= 66 = (276, 3, 59)
  | ll <= 82 = (277, 4, 67)
  | ll <= 98 = (278, 4, 83)
  | ll <= 114 = (279, 4, 99)
  | ll <= 130 = (280, 4, 115)
  | ll <= 162 = (281, 5, 131)
  | ll <= 194 = (282, 5, 163)
  | ll <= 226 = (283, 5, 195)
  | ll <= 257 = (284, 5, 227)
  | ll == 258 = (285, 0, 258)
  | otherwise = error "invalid length"

{-
Given a lit val, return the corresponding huffman tree val
and also bit length
-}
llHuffmanCode :: W.Word16 -> (W.Word16, W.Word8)
llHuffmanCode val
  | val <= 143 = (48 + val, 8)
  | val <= 255 = (400 + (val - 144), 9)
  | val <= 279 = (val - 256, 7)
  | val <= 287 = (192 + (val - 280), 8)
  | otherwise = error "invalid val"

{-
(code, num extra bits, base)
-}
distTable :: W.Word16 -> (W.Word16, W.Word8, W.Word16)
distTable d
  | d == 0 = error "invalid zero dist"
  | d == 1 = (0, 0, 1)
  | d == 2 = (1, 0, 2)
  | d == 3 = (2, 0, 3)
  | d == 4 = (3, 0, 4)
  | d <= 6 = (4, 1, 5)
  | d <= 8 = (5, 1, 7)
  | d <= 12 = (6, 2, 9)
  | d <= 16 = (7, 2, 13)
  | d <= 24 = (8, 3, 17)
  | d <= 32 = (9, 3, 25)
  | d <= 48 = (10, 4, 33)
  | d <= 64 = (11, 4, 49)
  | d <= 96 = (12, 5, 65)
  | d <= 128 = (13, 5, 97)
  | d <= 192 = (14, 6, 129)
  | d <= 256 = (15, 6, 193)
  | d <= 384 = (16, 7, 257)
  | d <= 512 = (17, 7, 385)
  | d <= 768 = (18, 8, 513)
  | d <= 1024 = (19, 8, 769)
  | d <= 1536 = (20, 9, 1025)
  | d <= 2048 = (21, 9, 1537)
  | d <= 3072 = (22, 10, 2049)
  | d <= 4096 = (23, 10, 3073)
  | d <= 6144 = (24, 11, 4097)
  | d <= 8192 = (25, 11, 6145)
  | d <= 12288 = (26, 12, 8193)
  | d <= 16384 = (27, 12, 12289)
  | d <= 24576 = (28, 13, 16385)
  | d <= 32768 = (29, 13, 24577)
  | otherwise = error "invalid dist"
```

MyCRC32.hsc

```
{-# LANGUAGE ForeignFunctionInterface, FlexibleInstances #-}
------------------------------------------------------------
-- |
-- Copyright    :   (c) 2008 Eugene Kirpichov
-- License      :   BSD-style
--
-- Maintainer   :   ekirpichov@gmail.com
-- Stability    :   experimental
-- Portability  :   portable (H98 + FFI)
--
-- CRC32 wrapper
------------------------------------------------------------

module MyCRC32 (
    CRC32, crc32, crc32Update, crc32Combine
) where

import Data.ByteString.Unsafe (unsafeUseAsCStringLen)
import Foreign

import qualified Data.ByteString as S
import qualified Data.ByteString.Lazy as L
import qualified Data.ByteString.Lazy.Internal as LI
import qualified System.IO.Unsafe as U

#include "zlib.h"

-- | The class of values for which CRC32 may be computed
class CRC32 a where
    -- | Compute CRC32 checksum
    crc32 :: a -> Word32
    crc32 = crc32Update 0

    -- | Given the CRC32 checksum of a string, compute CRC32 of its
    -- concatenation with another string (t.i., incrementally update
    -- the CRC32 hash value)
    crc32Update :: Word32 -> a -> Word32

instance CRC32 S.ByteString where
    crc32Update = crc32_s_update

instance CRC32 L.ByteString where
    crc32Update = crc32_l_update

instance CRC32 [Word8] where
    crc32Update n = (crc32Update n) . L.pack
```

```
crc32_s_update :: Word32 -> S.ByteString -> Word32
crc32_s_update seed str
    | S.null str = seed
    | otherwise =
        U.unsafePerformIO $
        unsafeUseAsCStringLen str $
        \(buf, len) -> fmap fromIntegral $
            crc32_c (fromIntegral seed) (castPtr buf) (fromIntegral len)

crc32_l_update :: Word32 -> L.ByteString -> Word32
crc32_l_update = LI.foldlChunks crc32_s_update

crc32Combine :: Word32 -> Word32 -> Word64 -> Word32
crc32Combine crc1 crc2 len2 = fromIntegral $ U.unsafePerformIO $ combine
    where
        combine = crc32_combine_c (fromIntegral crc1) (fromIntegral crc2) (fromIntegral len2)

foreign import ccall unsafe "zlib.h crc32"
    crc32_c :: #{type uLong}
            -> Ptr #{type Bytef}
            -> #{type uInt}
            -> IO #{type uLong}

foreign import ccall unsafe "zlib.h crc32_combine"
    crc32_combine_c :: #{type uLong}
                    -> #{type uLong}
                    -> #{type z_off_t}
                    -> IO #{type uLong}
```