An efficient implementation of Deflate in Coq

Christoph-Simon Senjak

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

PUMA & RiSE Workshop 2015

The Deflate format

- Deflate is probably the most widespread compression format (used in several other standards (HTTP, ZIP, RAR, PNG)).
- Specified in RFC 1951. (RFC 1952 specifies the GZIP file format, and RFC 1950 specifies the ZLIB file format. Both are often confused with Deflate).
- Can make use of Huffman codings, Run-length encoding and Lempel-Ziv-compression, but does not require any specific compression algorithm

The Compulsory Slide

An informal illustration of the format:

```
Deflate                ::=  ('0' Block)* '1' Block (0|1)*
Block                  ::=  '00' UncompressedBlock |
                            '01' DynamicallyCompBl |
                            '10' StaticallyCompBl
UncompressedBlock      ::=  length ~length bytes
StaticallyCompBl       ::=  CompBl(standard coding)
DynamicallyCompBl      ::=  header coding CompBl(coding)
CompBl(c)              ::=  [^256]* 256
```

What we already have

- Mathematical specification.
- Injectivity and existence proofs for this specification.
- Compression and decompression algorithm in Coq:
  - Compression and decompression are inverse due to uniqueness $\Rightarrow$ verified data integrity (in absence of hardware errors).
  - Still slow, but "minutes"-slow rather than "weeks"-slow (or "centuries").
  - Both use lazy-IO (extraction to Haskell rather than OCaml) $\Rightarrow$ still a large trusted codebase.
- Benchmarks show that resolution of backreferences is the current bottleneck.

Backreferences - Example

Original text: `ananas_banana_batata$` (where $ is the end-marker).

$[\overrightarrow{l}, \overleftarrow{d}]$ shall mean "go back d bytes in the history, and copy l bytes forward".

→ `ananas_b`$[\overrightarrow{5}, \overleftarrow{8}][\overrightarrow{3}, \overleftarrow{7}]$`tata`

Simple algorithm:

```
void resolve (int len, int dst, char* p) {
  while (len--> 0) { *p = *(p-dst); p++; }}
```

also works when $l > d$, results in repetition of already written bytes (run length encoding).

⇒ `an`$[\overrightarrow{3}, \overleftarrow{2}]$`s_b`$[\overrightarrow{5}, \overleftarrow{8}][\overrightarrow{3}, \overleftarrow{7}]$`t`$[\overrightarrow{3}, \overleftarrow{2}]$

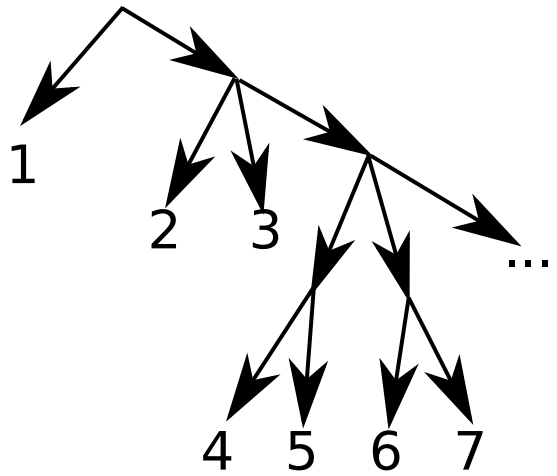
Backreferences - 1

- Simple with stateful operations (e.g. ringbuffer).
- However, harder to do purely functional.

naïve solution (on reverse list, `!!` means n-th):

```
r 0 d x = x
r l d x = r (l - 1) d ((x !! (d - 1)) : x)
```

more sophisticated: Use recursive slowdown, so n-th element can be accessed in $O(\log n)$ ("explist").



Backreferences - 2

This is still memory-hungry. But the range of backreferences is bounded. We use a "forgetful" list, consisting of two explists.

start	[]	[]
push 1	[1]	[]
push 2	[2; 1]	[]
push 3	[3; 2; 1]	[] → ☠
push 4	[4]	[3; 2; 1]
push 5	[5; 4]	[3; 2; 1]
push 6	[6; 5; 4]	[3; 2; 1] → ☠
push 7	[7]	[6; 5; 4]

Backreferences - 3

- `kennedy.xls` from the Canterbury Corpus (1029744 Bytes) takes 18s with our non-optimized extracted algorithm without resolution of backreferences (on a 3.4 GHz machine). It takes 44min with our current algorithm – and weeks with the naïve algorithm.

$\Rightarrow$ Still too slow, but the right direction.

- We are currently working on a much faster, but still purely functional algorithm that reads backreferences in advance. Our unverified implementation takes 20s for `kennedy.xls`, that is, only 2s for actual resolution

A faster purely functional algorithm - 1

- Idea: Read backreferences in advance. Count the bytes, work with **absolute** positions. Save source and destination position in a priority queue, sorted by the **absolute** source position. When reaching the absolute source position, save the byte and the destination into another priority queue, sorted by the **absolute** destination position.

$\Rightarrow$ The queues' sizes are limited by the maximal distance 32KiB.

- To make it easier, we first transform every backreference into a sequence of backreferences of length 1: $[\overrightarrow{3}, \overleftarrow{d}] = [\overrightarrow{1}, \overleftarrow{d}][\overrightarrow{1}, \overleftarrow{d}][\overrightarrow{1}, \overleftarrow{d}]$. It produces no overhead because of lazyness. We just have to save the $\overleftarrow{d}$ part then.

A faster purely functional algorithm - 2

For clarity, we show an algorithm that is equivalent to ours for very large/unlimited distances. We will use the example an $\overleftarrow{2}\ \overleftarrow{2}\ \overleftarrow{4}$ s.

```
input = map Left "an" ++ map Right [2,2,2] ++
        [Left 's']
```

Collect all backreferences, calculate their **absolute** source and destination, and sort the list according to the source:

```
collect x = (sortBy (\ (a, _) (b, _)->
                        compare a b)) (c 0 x)
    where
      c n [] = []
      c n (Left _ : r) = c (n+1) r
      c n (Right d : r) = (n-d, n) : c (n+1) r
```

Result: `collect input = [(0,2),(1,3),(2,4)]`

A faster purely functional algorithm - 3

Using this list [(0,2),(1,3),(2,4)], we know when to save bytes in advance, and can use a priority queue to efficiently utilize this information:

- At position 0, we have character `a`. Furthermore, we have (0,2) at the head of our previous list. $\Rightarrow$ we push (2,a) on our priority queue. At position 1, same for `n`. The queue becomes [(2,a),(3,n)].
- At position 2, we have a backreference. The head of our queue contains (2, a), that is, the character at this position must be `a`. The remaining list of backreferences is [(2, 4)], so we have to push (4,a) to our queue, which becomes [(3,n),(4,a)].
- From [(3,n),(4,a)], the characters immediately follow.

A faster purely functional algorithm - 4

Since there is a maximal distance, our former `collect` function can be "weaved" into the algorithm, using an own priority queue that makes sure that the minimal element is always the element that the next backreference from the input corresponds to (this is now done by sorting the list).
However, this algorithm is not yet entirely verified.

Conclusion and Future Work

- There should be a tactic collection for sequences: Manually proving and applying lemmata such as `a ++ l = b ++ l -> a = b` is tedious.
- Smaller trusted codebase (native-coq, VST).
- Iteratees instead of lazy IO (?)

The Final Slide