An Implementation of Deflate in Coq

Christoph-Simon Senjak

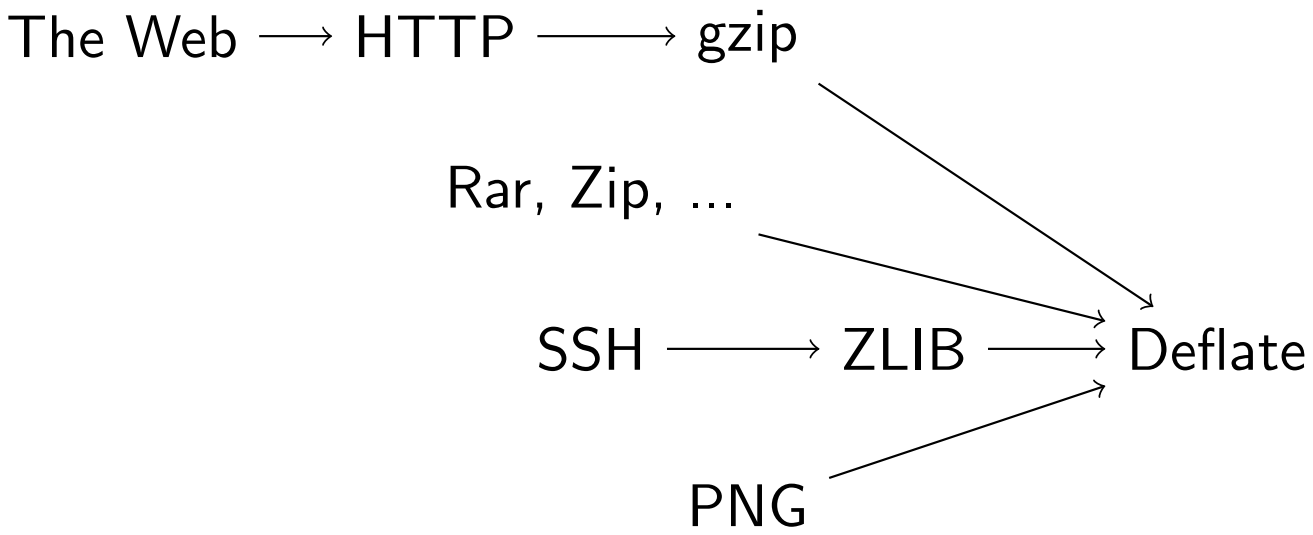
Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

Logik Oberseminar 08.06.2016

Deflate - Basics

- 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 and Run-length encoding.
- Does not require any specific compression algorithm, even though some are recommended.

Why Deflate?



Main implementation is the zlib (zlib.net).
Though the first release was 1995, in 2005 there were still security vulnerabilities.
And there are still a lot of bugfixes with every version.

Deflate - A very brief overview

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
```

Does not require any specific compression algorithm, even though some are recommended $\Rightarrow$ we use encoding relations rather than directly writing a function.

What we have

- Mathematical specification of a compression format that is very likely Deflate
  - Tested it with lots of randomized data
  - And with real-life data
  - However, the specification is axiomatic, and the standard is informally specified.
- Implementation of a compression and a decompression algorithm through program extraction
  - Verified to be inverse to each other.
  - Still slow, but “awaitable”, lots of potential for optimization.

Pros and Cons of Extraction

Pros:

- Sophisticated formats should come with (at least informal) correctness proofs anyway

Cons:

- Proofs must be (mostly) constructive
- Harder to see the computational complexity, especially in presence of tactics - but: Semi-automatic theorem proving and program optimizers get better.

Coq can do both verification of functions as well as program extraction from proofs.
Our experience was that extracting to Haskell and utilizing laziness makes it a lot easier to write proofs from which useful algorithms can be extracted.

A little argument

"If the verified program is not faster than compressing and decompressing with the unverified program, it is worthless":

- Assume unverified algorithms c, d. Then the following are verified algorithms:

$$c'x = \begin{cases} (\top, cx) & \text{for } d(cx) = x \\ (\bot, x) & \text{otherwise} \end{cases}$$

$$d'x = \begin{cases} dy & \text{for } x = (\top, y) \\ y & \text{for } x = (\bot, y) \end{cases}$$

- But this assumes that c and d are always the same on every platform. It is prone to digital obsolescence.
- It does not allow later optimisations or even compatible bugfixes.
- $\Rightarrow$ better to have a formal specification and verify against it.
- Still a good "benchmark". We are still far away from that.

Deflate Codings

1. $\lceil\cdot\rceil^{\{0,\ldots,n-1\}\to\{0,1\}^*}$ is prefix-free, except that there may be codes of length zero (which denote that the character does not occur):

$$\forall_{ab}.a \neq b \to \lceil a\rceil \neq [\,] \to \lceil a\rceil \preccurlyeq \lceil b\rceil \to \bot$$

2. Shorter codes lexicographically precede longer codes:

$$\forall_{ab}.\,\mathrm{len}\lceil a\rceil < \mathrm{len}\lceil b\rceil \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

3. Codes of the same length are ordered lexicographically according to the order of the characters they encode:

$$\forall_{ab}.\,\mathrm{len}\lceil a\rceil = \mathrm{len}\lceil b\rceil \to a \leq b \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

4. For every code, all lexicographically smaller bit sequences of the same length are prefixed by some code:

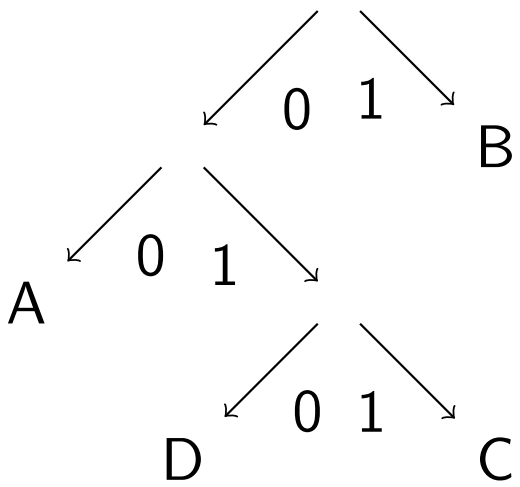
$$\forall_{a\in\{0,\ldots,n-1\},l\in\{0,1\}^*}.l \neq [\,] \to l \sqsubseteq \lceil a\rceil \to$$
$$\mathrm{len}\,l = \mathrm{len}\lceil a\rceil \to \exists_b.\lceil b\rceil \neq [\,] \wedge \lceil b\rceil \preccurlyeq l$$

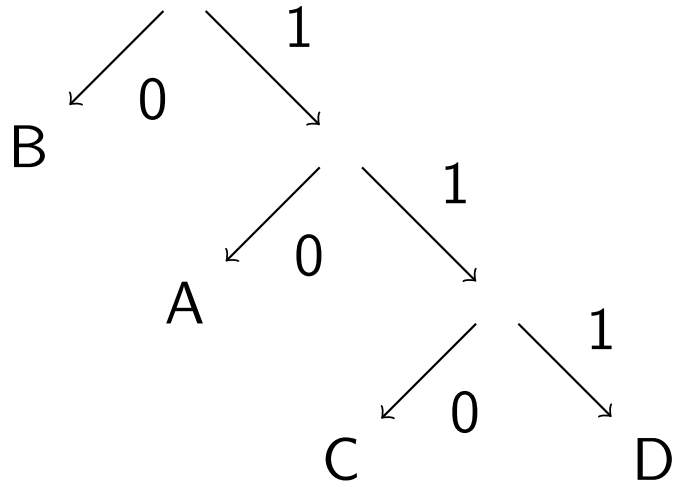
Formalization in Coq

```
Record deflateCoding (M : nat) : Set :=
  mkDeflateCoding {
      C : VecLB M ;
      prefix_free : forall a b, a <> b ->
      ((Vnth C a) <> nil) -> ~ (prefix (Vnth C a) (Vnth C b)) ;
      length_lex : forall a b, ll (Vnth C a) < ll (Vnth C b) ->
      lex (Vnth C a) (Vnth C b) ;
      char_enc : forall a b, ll (Vnth C a) = ll (Vnth C b) ->
      (f_le a b) -> lex (Vnth C a) (Vnth C b) ;
      dense : forall a c, c <> nil -> lex c (Vnth C a) ->
      ll c = ll (Vnth C a) ->
      exists b, (~ (Vnth C b) = nil) /\ (prefix (Vnth C b) c)
  }.
```

Codings - Example

RFC 1951 gives the following example:
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$. It does not satisfy (2.) and (3.), but the equally good $A \to 10$, $B \to 0$, $C \to 110$, $D \to 111$ does. It is uniquely determined by the sequence 2,1,3,3.
The corresponding trees look like this:





Deflate Coding Sequences

- A **Deflate sequence** is a sequence $(a_i)_i$ of code-lengths or 0 that satisfies **Kraft's inequality** $\sum_{i,a_i \neq 0} 2^{-a_i} \leq 1$.
- **Main Theorem:** The type of Deflate sequences is isomorphic to the type of Deflate codings.
- "uniqueness":

```
Lemma uniqueness :
  forall (D E : deflateCoding) (eq : M D = M E),
    (Vmap (ll (A:=bool)) (vec_id eq (C D))) =
      (Vmap (ll (A:=bool)) (C E)) ->
   coding_eq D E.
```

- "existence":

```
Lemma existence : forall n (f : vec nat n),
  kraft_nvec f <= 1 ->
   { D : deflateCoding |
     {eq : M D = n |
       f = (Vmap (ll(A:=bool)) (vec_id(A:=LB) eq (C D)))}}.
```

Outline of the Uniqueness-Proof

- We may do a proof by contradiction (equality of codings is decidable).
- So assume we had two distinct Deflate codings D and E. Then there exist n, m such that $D(n) = \min_{\sqsubseteq}\{D(x) \mid D(x) \neq E(x)\}$ and $E(m) = \min_{\sqsubseteq}\{E(x) \mid D(x) \neq E(x)\}$
- $\operatorname{len} D(n) > \operatorname{len} E(m)$ implies $D(m) \sqsubseteq D(n)$, but $D(n)$ was chosen minimal. Symmetric for $\operatorname{len} D(n) < \operatorname{len} E(m)$. So $\operatorname{len} D(n) = \operatorname{len} E(m)$. Also, $E(m) \neq []$, since otherwise $D(m) = E(m)$ by the length condition. Same for $D(n)$.
- By totality of $\sqsubseteq$, we know that $D(n) \sqsubseteq E(m) \vee E(m) \sqsubseteq D(n)$. Both cases are symmetric. Assume $D(n) \sqsubseteq E(m)$.
- By 4, we know that some b exists, such that $E(b) \prec D(n)$, therefore $E(b) \sqsubseteq E(m)$, and thus either $b = m$ or $E(b) = D(b)$. $b = m$ would imply $D(m) = E(m)$. $E(b) = D(b)$ would imply $D(b) \prec D(n)$ which contradicts 1.

Existence - 1

- Here we need to care more about efficiency.
- We use a recursive algorithm.
- The algorithm will be similar to the algorithm in RFC 1951, but not the same, as the standardized algorithm makes extensive use of stateful operations.
- We basically do a recursion on the list of pairs (char,length) of the given length-function, sorted firstly according to length and secondly according to char.

Existence - 2

Let $(c_1, l_1) \lesssim (c_2, l_2) :\Leftrightarrow (l_1 < l_2) \vee (l_1 = l_2 \wedge c_1 < c_2)$. Let a Deflate sequence $l : \{0, \ldots, n-1\} \to \mathbb{N}$ be given that satisfies Kraft's inequality. Let $L := \mathrm{sort}_{\lesssim}[(x, lx) \mid x \in \{0, \ldots, n-1\}]$. We begin with $S = [\,]$, $c = \lambda_x[\,]$, $R = L$. The following invariants are maintained in the recursion step:

- $\forall_q.(q, \mathrm{len}(cq)) \notin S \to (cq = [\,] \wedge (q, lq) \in R)$
- $(\mathrm{rev}\, S) +\!\!+ R = L$
- $S = [\,] \vee \forall_q.cq \sqsubseteq c(\pi_1(\mathrm{car}\, S))$

Existence - 3

We only look at one of the several cases one has to handle here: The case $S = (n_S, 1 + m_S) :: S'$, $R = (n_R, 1 + m_R) :: R'$, let the intermediate coding c be given.
We have

$$\sum_{\substack{i \in [n] \\ ci \neq []}} 2^{-\operatorname{len}(ci)} < 2^{-m_R - 1} + \sum_{\substack{i \in \{0,\ldots,n-1\} \\ ci \neq []}} 2^{-\operatorname{len}(ci)} \leq \sum_{\substack{i \in \{0,\ldots,n-1\} \\ li \neq 0}} 2^{-li} \leq 1$$

This means that Kraft's inequality is sharp, so $[\underbrace{1, \ldots, 1}_{1+m_S}] \notin \operatorname{img} c$, and we can find a fresh code d' of length $1 + m_S$. Let $d = d' +\!\!+ [\underbrace{0, \ldots, 0}_{m_R - m_S}]$ and set

$$c'x := \begin{cases} d & \text{for } x = n_R \\ cx & \text{otherwise} \end{cases}$$

Verifying the axioms for c' is easy.

Existence - 4

For a better understanding of the algorithm proposed here, we consider the following length function as an example:

$$l : 0 \to 2; 1 \to 1; 2 \to 3; 3 \to 3; 4 \to 0$$

We first have to sort the graph of this function according to the $\lesssim$ ordering.

$$[(4, 0), (1, 1), (0, 2), (2, 3), (3, 3)]$$

Then, the following six steps are necessary to generate the coding.

Existence - 5

	R	S	c(0)	c(1)	c(2)	c(3)	c(4)
0	[(4, 0), (1, 1), (0, 2), (2, 3), (3, 3)]	[]	[]	[]	[]	[]	[]
1	[(1, 1), (0, 2), (2, 3), (3, 3)]	[(4, 0)]	[]	[]	[]	[]	[]
2	[(0, 2), (2, 3), (3, 3)]	[(1, 1), (4, 0)]	[]	[0]	[]	[]	[]
3	[(2, 3), (3, 3)]	[(0, 2), (1, 1), (4, 0)]	[1,0]	[0]	[]	[]	[]
4	[(3, 3)]	[(2, 3), (0, 2), (1, 1), (4, 0)]	[1,0]	[0]	[1,1,0]	[]	[]
5	[]	[(3, 3), (2, 3), (0, 2), (1, 1), (4, 0)]	[1,0]	[0]	[1,1,0]	[1,1,1]	[]

A Theory of Parsers - 1

We work with relations of the form `output -> input -> Prop`. Such a relation must satisfy two conditions to be suitable for parsing:

- **Strong Uniqueness**, meaning that for a given string, there is at most one initial segment that can be parsed:

$$\begin{aligned}\text{StrongUnique}\, R \quad :\Leftrightarrow \forall_{a,b,l_a,l'_a,l_b,l'_b}.\quad & l_a +\!\!+ l'_a = l_b +\!\!+ l'_b \to \\ & R\, a\, l_a \to R\, b\, l_b \to \\ & a = b \wedge l_a = l_b\end{aligned}$$

- **Strong Decidability**, meaning that whether an initial segment exists is decidable.

$$\begin{aligned}\text{StrongDec}\, R :\Leftrightarrow \forall_l.\qquad & (\exists_{a,x,y}. l = x +\!\!+ y \wedge R\, a\, x) \vee \\ & (\neg(\exists_{a,x,y}. l = x +\!\!+ y \wedge R\, a\, x) \wedge \quad E)\end{aligned}$$

  where E is some error type (for us, it is a string giving an error message). This resembles an exception monad.

A Theory of Parsers - 2

- Strong decidability follows from decidability, but the proof would yield an exponential algorithm.
- Parsing relations can be combined in a monadic fashion:

```
Inductive Combine {A BQ BR BS}
 (Q : BQ → list A → Prop)
 (R : BQ → BR → list A → Prop)
 (c : BQ → BR → BS) : BS → list A → Prop :=
| doCombine : forall bq br aq ar,
    Q bq aq → R bq br ar →
      Combine Q R c (c bq br) (aq ++ar).

Notation "A >>[ B ]= C" := (Combine A C B).
Notation "A >>= B" := (Combine A B pi2).
```

Strong decidability and uniqueness can be proved for combined relations.

A Theory of Parsers - 3

Example:

```
Definition UncompressedBlockDirect : SequenceWithBackRefs Byte →
    LB → Prop :=
  (readBitsLSB 16) >>=
  fun len ⇒ (readBitsLSB 16) >>=
  fun nlen ⇒
    (fun swbr lb ⇒ len + nlen = 2 ^ 16 − 1 ∧ nBytesDirect len swbr lb).

Fixpoint nTimes {A B C} (n : nat) (null : C)
                        (comb : A → C → C)
                        (rel : A → list B → Prop)
: C → list B → Prop :=
  match n with
    | 0 ⇒ fun c L ⇒ c = null ∧ L = nil
    | (S n') ⇒ rel >>[ comb ]= fun _ ⇒
                 (nTimes n' null comb rel)
  end.
```

A Theory of Parsers - 4

- Advantages of our theory:
  - It is simple, it needs no sophisticated library.
  - It yields algorithms that can be extracted directly from proofs.
- Disadvantages:
  - The input must be total – otherwise, some relations would not be decidable
  - It does not fully allow for lazy-i/o, and combining exception monads consumes stack space.

A Theory of Parsers - 5

As **further work**, some of the disadvantages of our theory should be solvable by using *Iteratees*:

```
data Iteratee i o
     = Done o
     | Next (Maybe i -> Iteratee i o)

type Enumerator i o =
     Iteratee i o -> Iteratee i o

listEnumerator :: [i] -> Enumerator i o
listEnumerator _ it@(Done _) = it
listEnumerator [] (Next f) = f Nothing
listEnumerator (x : l) (Next f) =
    listEnumerator l ( f ( Just x ))
```

(See `http://okmij.org/ftp/Haskell/Iteratee/describe.pdf`)

A Theory of Parsers - 6

- We can add additional branches, for example to pass more than one character at a time.
- An iterator that returns an iterator is called enumeratee.
- Such constructs can be “piped” into one another, similar to unix pipes and processes.
- The syntax can be made similar to that of lazy I/O, while retaining control over cache and memory.

A Theory of Parsers - 7

A first prototype we designed contains strong guarantees. However, this is just a prototype, we did not elaborate it. It is “further work”.

```
Inductive Iteratee { I O : Set }
      {R : list I -> O -> Prop } ( l1 : list I ) :=
| Done : forall o , R l1 o -> Iteratee l1
| Error : string ->
          (forall l2 o ,  ~R ( l1 ++ l2 ) o )
          -> Iteratee l1
| Proceed : ( forall l2, Iteratee ( l1 ++ l2 ) )
          -> Iteratee l1.
```

Backreferences - 1

- A backreference is a pair $\langle l, d \rangle$ of a length l (number of bytes to be copied) and a distance d (number of recently extracted bytes that have to be skipped).

$$\texttt{ananas\_banana\_batata} \Rightarrow \texttt{ananas\_b}\,\langle 5, 8\rangle\,\langle 3, 7\rangle\,\texttt{tata}$$

- An intuitive algorithm would be:

```
int
resolve (int len, int dist, int pointer, byte[] output) {
  while (len > 0) {
    output[pointer] = output[pointer-dist];
    pointer = pointer + 1;
    len = len - 1;
  }
  return pointer;
}
```

- $\Rightarrow$ Works also when $l > d$, results in a repetition of already written bytes $\Rightarrow$ an $\langle 3, 2\rangle$ s_b $\langle 5, 8\rangle$ $\langle 3, 7\rangle$ t $\langle 3, 2\rangle$ $\Rightarrow$ Run-length-encoding

Backreferences - 2

- The resolution of backreferences has been the major bottleneck from the beginning.
- We tried several optimizations (using reverse lists, or Okasaki's catenable deques)
- Current implementation uses "ExpList":

```
Inductive ExpList (A : Set) : Set :=
| Enil : ExpList A
| Econs1 : A → ExpList (A * A) → ExpList A
| Econs2 : A → A → ExpList (A * A) → ExpList A.
```

Backreferences with ExpLists

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):

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]	[3; 2; 1] → ☠

A purely functional efficient backreference resolver

We have a (not yet verified) efficient algorithm for resulution of backreferences:

- The algorithm reads the input at two positions that are at most 32 KiB apart (which is the maximum distance of backreferences)
- It uses two priority queues where it saves backreferences as source-destination-pairs and destination-value-pairs, it maintains these simultaneously; it reads backreferences in advance instead of saving a back buffer.
- This algorithm is fast, purely functional, but due to its complicated invariants (and the lack of time), we did not yet verify it.

Performance

These are benchmarks from the benchmarks are for the Canterbury Corpus on an `Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz`. We also show the performance without resolving backreferences at all, and with our new, yet unverified resolution algorithm.

File	Original Bytes	Compressed Bytes	Compress Time	Decompress (No Backreferences)	Decompress (ExpList)	Decompress (Queue)
`alice29.txt`	152089	118126	2m51.480s	0m5.047s	5m13.108s	0m5.431s
`asyoulik.txt`	125179	97187	1m56.700s	0m4.207s	4m23.135s	0m4.540s
`cp.html`	24603	15139	0m6.907s	0m0.934s	0m39.143s	0m0.961s
`fields.c`	11150	6325	0m1.735s	0m0.576s	0m4.101s	0m0.600s
`grammar.lsp`	3721	2013	0m0.573s	0m0.417s	0m0.641s	0m0.418s
`kennedy.xls`	1029744	439956	60m59.977s	0m18.212s	44m12.826s	0m20.343s
`lcet10.txt`	426754	327304	27m18.671s	0m13.391s	19m12.085s	0m14.230s
`plrabn12.txt`	481861	389913	40m13.934s	0m16.114s	21m7.902s	0m17.851s
`ptt5`	513216	269382	49m16.218s	0m13.327s	23m22.609s	0m14.012s
`sum`	38240	21703	0m22.869s	0m1.270s	1m58.054s	0m1.329s
`xargs.1`	4227	2656	0m0.847s	0m0.425s	0m1.481s	0m0.419s