Coding Trees in the Deflate format

Christoph-Simon Senjak

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

Oberseminar Informatik LMU 2014

Outline

- Deflate is probably the most widespread compression format
- We use it as case study for low-level verification
- Deflates way to store coding trees (Huffman trees) is a lot more complicated than expected

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 (extended) Read-length encoding.
- Does not require any specific compression algorithm, even though some are recommended.
- Widely used: HTTP, ZIP/RAR, PNG, SSH

Deflate - 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
```

We are interested in the "coding" part.

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)
4. If there is a lexicographically smaller code c of the same length for some code $D(a)$ in the coding, then it is prefixed by some image of the coding D:
   $\forall_{a,c}.c \sqsubseteq D(a) \rightarrow \operatorname{len} c = \operatorname{len} D(a) \rightarrow \exists_b.D(b) \neq [\,] \wedge D(b) \preccurlyeq c$

Point 4 is not explicitly stated in RFC 1951, but the algorithms and examples it gives imply it.

Formalization in Coq

```
Record deflateCoding : Set :=
mkDeflateCoding {
  M : nat ;
  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)
}.
```

Deflate Coding Sequences

- A **Deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies **Kraft's inequality** $\sum_{i,c_i \neq 0} 2^{-a_i} \leq 1$.
- **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.
- 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.

Uniqueness in Coq

- About 400 loc, excluding lemmata about lexicographical orders and prefixes.
- Does not (yet) exploit the symmetric cases.

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 $l : [n] \to \mathbb{N}$ be given that satisfies Kraft's inequality. Let $L := \mathrm{sort}_{\lesssim}[(x, lx) \mid x \in [n]]$. 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 - 2

For the case $S = (n_S, 1 + m_S) :: S'$, $R = (n_R, 1 + m_R)$, let the intermediate coding c be given.
We have

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

Therefore, $\underbrace{[1, \ldots, 1]}_{1+m_S} \notin \operatorname{img} c$, 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

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

Existence - 3

We have to show that c' is a Deflate coding.

- Axioms 2 and 3 are easy.
- For axiom 4, assume $x \neq []$ and $x \sqsubseteq c' n_R$. If $x \sqsubseteq c' r$ for some $r \neq n_R$, the claim follows by axiom 4 for r and c. Otherwise, by totality of $\sqsubseteq$ we have $c' r \sqsubseteq x$. If $x \sqsubseteq d'$, by the minimality of d' follows $x = c' r$. If $d' \sqsubseteq x$, trivially, $d' \prec c' n_R$.
- For axiom 2, it is sufficient to show that no other non-$[]$ code prefixes d. Consider a code $e \prec d$. As all codes are shorter or of equal length than d', $e \prec d'$. But then, either $e \prec cr$, or $cr \sqsubseteq e$. Contradiction.

Therefore, we can proceed with $((q, 1 + m_R) :: S, c', R')$.
The other cases are easy.

Existence in Coq

- About 2000 loc, not including external lemmata.
- Involves a lot of small combinatorical lemmata.
- Takes long time to compile.

Problems

- The distinction between computational and non-computational definitions: `inl`, `or_introl`, `left`. We chose to use computational definitions wherever possible, at least for now, probably sacrificing efficiency.
- Substitutions with the `rewrite` tactic do not affect type parameters (especially for `Fin.t` which we often use). It is possible to circumvent this. However, we chose to make extensive use of `dependent induction`, which uses `JMeq`.
- Hidden parameters can make it hard to understand why substitutions are not possible, but the option `Set Printing All` can be confusing.
- The amount of small combinatorical lemmata needed is annoying.
- Version conflicts with automatic naming and resolution.

Remaining parts

- RLE, calculating optimal code-lengths, putting the pieces together.
- Port it to the stable Coq-Version (We used a git-version of Coq).
- Compile and test it. Maybe shorten proofs.
- Efficiency! (Utilize irrelevance, etc.)

Conclusion

- The algorithm as such can be implemented very efficiently.
- Verification is, however, very complicated.

$\Rightarrow$ Even more useful to have a verified reference implementation.