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

Agda Implementor's Meeting 2014

Outline

- Deflate is probably the most widespread compression format
- We use it as case study for low-level verification
- Deflates way to store encoding 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) Run-length encoding.
- Does not require any specific compression algorithm, even though some are recommended.
- Widely used: HTTP, ZIP/RAR, PNG, SSH

```
$ apt-cache rdepends zlib1g | wc -l
1418
```

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.

Deflate Trees in Agda

- Deflate codings can be viewed as coding trees. A first formalization in Agda:

```
data DeflateTree {alpha : ℕ} : (range : Subset alpha)
                          → (shortHeight : ℕ) → (shortBranchChar : Fin alpha)
                          → (longHeight : ℕ) → (longBranchChar : Fin alpha)
                          → (notNonFork : Bool) → Set where
  leaf : (b : Fin alpha) → DeflateTree ⁅ b ⁆ 0 b 0 b true
  forkEq : {ld rd md : ℕ} → {f g : Subset alpha} →
           {lc mc1 mc2 rc : Fin alpha} → {tl : Bool}
         → DeflateTree f ld lc md mc1 true → DeflateTree g md mc2 rd rc tl
         → (disjoint : Empty (f ∩ g))
         → (toℕ mc1) < (toℕ mc2)
         → DeflateTree (f ∪ g) ld lc rd rc tl
  forkNeq : {ld md1 md2 rd : ℕ} → {f g : Subset alpha} → {lc mc1 mc2 rc : Fin alpha}
          → {tl : Bool} → DeflateTree f ld lc md1 mc1 true
          → DeflateTree g md2 mc2 rd rc tl
          → (disjoint : Empty (f ∩ g))
          → md1 < md2 → DeflateTree (f ∪ g) ld lc rd rc tl
  nonFork : {b : Bool} → {ld rd : ℕ} → {f : Subset alpha} → {lc rc : Fin alpha}
          → DeflateTree f ld lc rd rc b → DeflateTree f (suc ld) lc (suc rd) rc false
```

⇒ Complicated. Equivalence to the standard is not obvious.

Problems with Agda

- A lot of small combinatorical facts have to be proved.
- Often it is not easy to modularize the proofs.

$\Rightarrow$ Very large proof terms.

- Lack of usable libraries, but writing libraries is not the goal of this project.
- Especially no complete library for rational numbers (or is there one now?).
- Compilation is slow and the large proofs sometimes crashed Agda.

$\Rightarrow$ Switched to coq ;_;

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)
}.
```

$\Rightarrow$ much closer to the standard.

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 lists that satisfy Kraft's inequality is isomorphic to the type of Deflate codings.

$\Rightarrow$ Saving a sequence of lengths is not expensive, and that is the way it is done in Deflate, and is sufficient, as this theorem shows.

- As everything is decidable, we can split the proof into a "uniqueness" and an "existence" part.
- We made a detailled informal proof and now formalize it in Coq.
- The "uniqueness" part is finished:

```
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.
```

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.
- Advantage (?) of Coq vs Agda: It is much easier to “hack” proofs.
- Things I missed in Coq:
  - Putting placeholders into terms to fill them out later. In Coq, a similar functionality is given by the `refine` tactic, but it is not the same.
  - It is harder to see the algorithms behind proofs in Coq code. It is possible to view the proof terms, though.

Existence

- Here we need to care more about efficiency.
- We use a recursive algorithm with a lot of parameters that depend on each other. Even harder to modularize.
- The Algorithm will be similar to the algorithm in RFC 1951, but not the same, as this 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.

Problems

- The distinction between computational and non-computational definitions: `inl`, `or_introl`, `left`. We chose to not use non-computational definitions wherever possible, at least for now, probably sacrificing efficiency.
- Substitutions with the `rewrite` tactic do not go into 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.

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.