A Case Study On Practical Usability Of Dependently Typed Languages - Deflate

Christoph-Simon Senjak

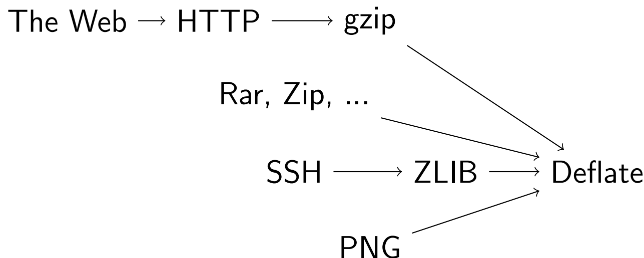
Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

PUMA Workshop 3. Oktober 2013

Foreword

- This talk gives an intermediate overview of work in progress.
- It is limited to one special data format, but aims to find problems and solutions through its implementation.
- The ultimate goal is to provide techniques for dealing with low-level structures in a verifiable way.

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 - 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.

Deflate - Overview

```
Deflate ::= ('0' Block)* '1' Block (0|1)*
Block    ::= '00' UncompressedBlock |
             '01' DynamicallyCompressedBlock |
             '10' StaticallyCompressedBlock
UncompressedBlock ::= length ~length bytes
StaticallyCompressedBlock ::=
  ( "code != 256" )* "code for 256"
DynamicallyCompressedBlock ::=
  header codes ( "code != 256" )* "code for 256"
```

Compressed Blocks may also contain backreferences to a 32KiB backbuffer.

Pitfalls

- The format is specified to operate on bytes. Bits have to be correctly extracted from them $\Rightarrow$ confusing rules about positions of lsb and msb.
- In some cases, byte-boundaries are relevant $\Rightarrow$ it is not possible to abstract away from the bytes and operate only on the resulting bitstream.
- Almost no implementation gives a pure Deflate stream, even though it claims so (for example java.util.DeflaterOutputStream) $\Rightarrow$ extracting them from gzip streams seems to be the easiest way.

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)

$\Rightarrow$ Can be implemented as a dependent tree structure in Agda.

- **Theorem:** For every prefix-free coding, there is an equally good or better deflate coding.

Code Example for Deflate Codings

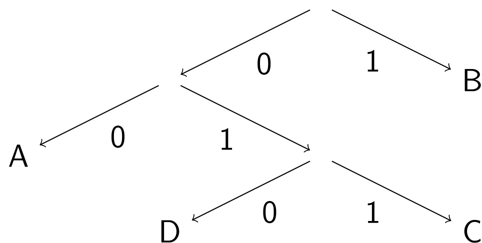
```
data DeflateTree {alpha : ℕ} : (range : Subset alpha)
                         → (shortHeight : ℕ) → (shortBranchChar : Fin alpha)
                         → (longHeight : ℕ) → (longBranchChar : Fin alpha)
                         → (notNonFork : Bool) → Set where
  leaf : (b : Fin alpha) → DeflateTree ⦃ 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
```

Deflate Coding Sequences

- A **deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0.
- It must satisfy $\sum_i (2^{-a_i} \cdot \operatorname{sgn} a_i) \leq 1$.
- **Main Theorem:** The projection of deflate codings on these sequences of code lengths is bijective.

Codings - Example

RFC 1951 gives the following example:
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$. The tree looks like



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.

Codings in Agda

- Trees can be realized as dependent trees.
- Deflate sequences can be realized as dependent lists.
- Theorems should be proved - which is currently the hardest part.

Run-Length Encoding

Data streams often contain duplicates of strings that have already been sent. Several algorithms exist to find these duplicates and eliminate them, using a backreference.

```
#include <stdio.h> \n#include <unistd.h>
```

```
#include <stdio.h> \n ○ unistd.h>
```

10

Run-Length Encodings in Deflate

Deflate allows backreferences up to 32 KiB - a sufficiently large backbuffer must be maintained. This is a challenge in a purely functional environment. Possible solutions:

- Efficient purely functional data structures (Okasaki, etc.)
- DiffArrays
- Monads (ST, IO)
- Uniqueness Types

Results so far

- Proof-of-Concept-Implementation of "gunzip" in pure Haskell, with recursive slowdown and with runST. Takes $<$ 2 minutes for about 6 MB: slow, but a start. Main problem is the 32K backbuffer. Can be found at https://github.com/dasuxullebt/inflate.hs
- Informal "pen-and-paper"-proofs for the theorems we need.
- Expierience while formalizing them: Work "top-down": Make excessive use of Agda's postulate command and prove the complicated stuff in the end.

Goals

- Provide a formal specification of the Deflate format in Agda.
- Create an implementation of “inflate” (decompression of a Deflate-stream) which is both efficient and verified.
- Test it against many gzipped files, to make sure the specification is compliant with the standard.

Plans

- Currently working on the formalization in Agda.
- Optimizing the Haskell-Version (feel free to look at it and send me comments), do experiments with linear types and diffarrays.
- Porting and verifying the current implementation from Haskell to Agda.