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

Christoph-Simon Senjak

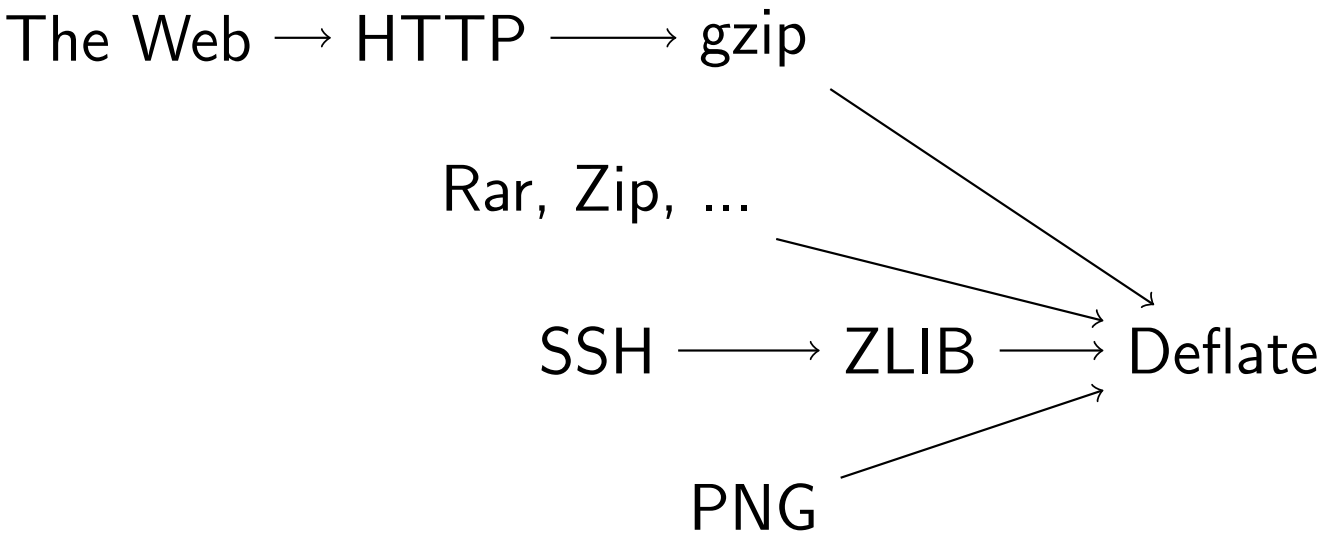
Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

Oberseminarvortrag 19. Juli 2013

Foreword

- This talk gives an intermediate overview of work in progress.
- It mainly contains ideas and goals.
- 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.

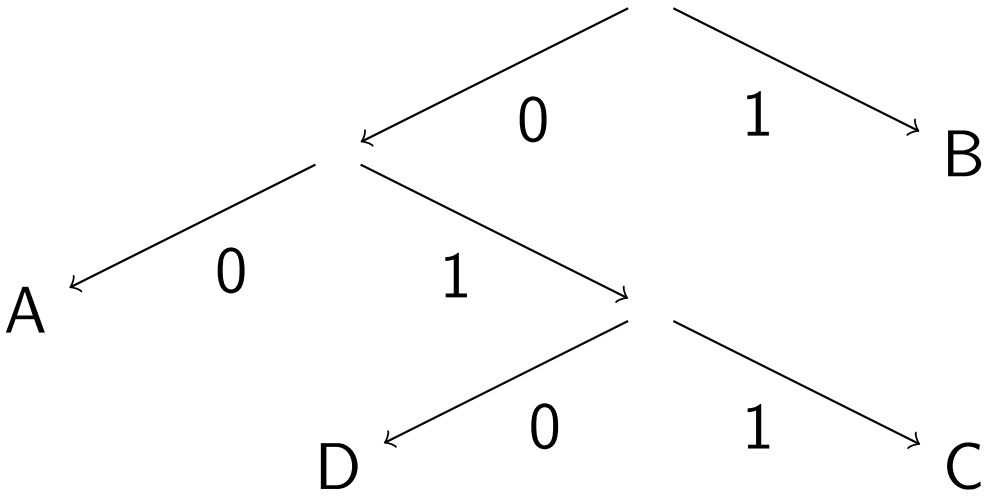
Codings - Definitions

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. For Deflate, we additionally require that the shorter codes lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. Furthermore, 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)

- For every prefix-free coding one can find an equally good prefix-free coding satisfying (2.) and (3.)
- Such a coding is uniquely defined by the sequence of code lengths.

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

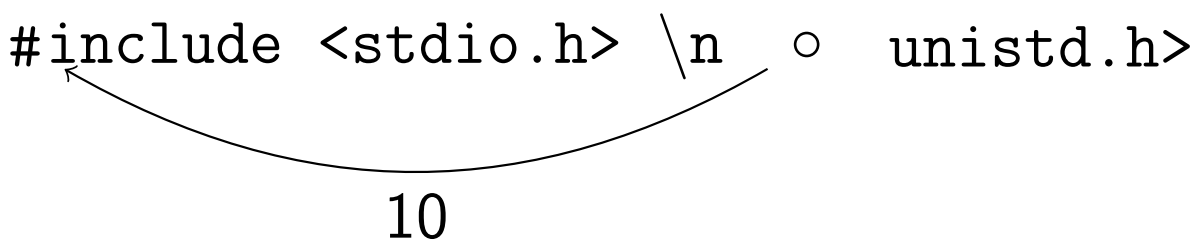


It does not satisfy (2.) and (3.), but the equally good $A \to 10$, $B \to 0$, $C \to 110$, $D \to 111$ does.
It is uniquely determined by the sequence 2,1,3,3.

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>
```



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.

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.

An Intermediate Result

- We made an implementation of “gunzip” in pure Haskell, to make sure that we understood the standard correctly, and see the problems that occur.
- The implementation takes about 2 minutes for about 6 MB, which is slow, but quite good regarding its purely functional source code.
- The main problem with efficiency is the 32K backbuffer for RLE. Implementation as a List is too slow. Current implementation with recursive slowdown is better, but still too slow.

Plans

- Currently working on the formalization in Agda.
- Porting and verifying the current implementation from Haskell to Agda.
- Defining a low-level language with semantics formalized in Agda, in which the equivalence of an efficient implementation of Deflate can be shown.