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

Agda Implementor's Meeting 2015

Goals

- Originally planned as a case study on the feasibility of dependent types for low-level formats.
- Originally in Agda, but at that time
  - 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 (addition of rational numbers).
  - Compilation is slow and the large proofs sometimes crashed Agda.
  - Tactics in Coq, especially `omega`.
- Now a case study for program extraction.

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.

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 (stored in a sophisticated way $\rightarrow$ not a simple parser).
  - Backreferences (which can be used to realize run length encoding).
- Widely used: HTTP, ZIP/RAR, PNG, SSH, and lots of other software

```
debian@jessie$ apt-cache rdepends zlib1g | wc -l
1965
```

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
```

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

Backreferences - 1

- Backreferences as found in Lempel-Ziv-compression, and Run-length-encoding, merged into one mechanism.
- 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:

```
resolve :: Int -> Int -> Int -> STArray s Int Int -> ST s Int
resolve len dist ptr out  =
 if len > 0
 then do byte <- readArray out $ ptr - dist
         writeArray out ptr byte
         resolve (len - 1) dist (ptr + 1) out
 else do return ptr
```

Backreferences - 2

- An intuitive algorithm would be:

```
resolve :: Int -> Int -> Int -> STArray s Int Int -> ST s Int
resolve len dist ptr out  =
 if len > 0
 then do byte <- readArray out $ ptr - dist
         writeArray out ptr byte
         resolve (len - 1) dist (ptr + 1) out
 else do return ptr
```

- $\Rightarrow$ Works also when $l > d$, results in a repetition of already written bytes $\Rightarrow$ Run-length-encoding
- Examples:
  - ananas_banana_batata $\Rightarrow$ an $\langle 3, 2 \rangle$ s_b $\langle 5, 8 \rangle$ $\langle 3, 7 \rangle$ t $\langle 3, 2 \rangle$
  - aaaaaaaargh! $\Rightarrow$ a $\langle 1, 7 \rangle$ rgh!

Backreferences in Coq - 1

Split into two parts:

- Copy *n*-th last byte to front:

```
Inductive nthLast {A : Set} :
 forall (n : nat) (L : list A) (a : A), Prop :=
| makeNthLast : forall L M a,
 nthLast (ll (a :: M)) (L ++ a :: M) a.
```

- Repeat the copying:

```
Inductive ResolveBackReference {A : Set} :
    forall (len dist : nat) (inp out : list A), Prop :=
| rbrZero : forall dist inp,
    ResolveBackReference 0 dist inp inp
| rbrSucc : forall n dist inp output1 X,
    ResolveBackReference n dist inp output1 ->
    nthLast dist output1 X ->
    ResolveBackReference (S n) dist inp (output1 ++ [X]).
```

Backreferences in Coq - 2

Multiple backreferences: We use a list of possible backreferences as an intermediate structure. To be able to use list operations and theorems, we do not define a new datatype.

```
Function SequenceWithBackRefs A := (list (A + (nat * nat))%type).

Inductive ResolveBackReferences {A : Set} :
  forall (input : SequenceWithBackRefs A) (output : list A), Prop :=
| ResolveNil : ResolveBackReferences (A:=A) nil nil
| ResolveByte : forall a b (c : A),
                  ResolveBackReferences a b ->
                  ResolveBackReferences (a ++ [inl c]) (b ++ [c])
| ResolveRef : forall a (b c : list A) len dist,
                  ResolveBackReferences a b ->
                  ResolveBackReference len dist b c ->
                  ResolveBackReferences (a ++ [inr (len, dist)]) c.
```

Resolving Backreferences in Coq

- For decompression, we prove the theorems

```
Theorem ResolveBackReferencesUnique :
 forall {A : Set} (input : SequenceWithBackRefs A)
                  output1 output2,
 ResolveBackReferences input output1 ->
 ResolveBackReferences input output2 ->
 output1 = output2.

Theorem ResolveBackReferencesDec :
 forall {A : Set} (input : SequenceWithBackRefs A),
  {output | ResolveBackReferences input output} +
  ({output | ResolveBackReferences input output} -> False).
```

- We can extract a function that resolves the backreferences. However, due to the heavy use of snoc- and append-operations, without modification it is slow.

Creating Backreferences in Coq

- Compression is still work in progress. We started with implementing Backreferences.
- Backreferences have bounds in Deflate $\Rightarrow$ predicate `BackRefsBounded`.
- So far, we proved

```
Theorem makeBackReferences :
 forall (minlen maxlen maxdist : nat) (l : LByte),
 {seq : SequenceWithBackRefs Byte |
   BackRefsBounded minlen maxlen maxdist seq /\
   ResolveBackReferences seq l}.
```

- This theorem is **trivial**, as we can just choose to never set backreferences.

  $\Rightarrow$ We try to implement a specific algorithm for finding possible backreferences, similar to the one specified in RFC 1951.

  $\Rightarrow$ Probably we will prove something more specific.

Strong Uniqueness and Decidability

- A parser for a byte stream canonically satisfies:
  1. *Strong Decidability*: $(\lambda_X.X \vee \neg X)(\exists_{a,x,y}.l = x +\!\!+ y \wedge Rax)$.
  ... Follows (in the relevant cases) from Decidability, but proofs can lead to bad runtime behavior.
  2. Left-Uniqueness ("injectivity"): $\forall_{a,b,l}.Ral \to Rbl \to a = b$.
  3. Furthermore: If there is a prefix, then it cannot be extended: $\forall_{a,b,l,l'}.Ral \to Rb(l +\!\!+ l') \to l' = [\,]$.

  $\Rightarrow$ 2. and 3. are equivalent to *Strong Uniqueness*: $\forall_{a,b,l_a,l'_a,l_b,l'_b}.l_a +\!\!+ l'_a = l_b +\!\!+ l'_b \to R\ a\ l_a \to R\ b\ l_b \to a = b \wedge l_a = l_b$.

$\Rightarrow$ We define a parser to be a relation satisfying strong uniqueness and strong decidability.

Prefix Free Codings

- Codings are functions from an alphabet $\{0, \ldots, n-1\}$ into bit sequences:
$$\lceil \cdot \rceil : \{0, \ldots, n-1\} \to \{0,1\}^*$$
- We allow [] in the image, and it shall mean that the preimages do not occur in the encoded sequence of characters from $\{0, \ldots, n-1\}$.
- We say that $\lceil \cdot \rceil$ is *prefix free* iff it is prefix free except for the preimages of []:
$$\forall_{ab}. a \neq b \to \lceil a \rceil \neq [] \to \lceil a \rceil \preccurlyeq \lceil b \rceil \to \bot$$
- Prefix free codings satisfy strong uniqueness (except for the preimages of []) and strong decidability (for example via code trees).

Deflate Codings

$\lceil\cdot\rceil : \{0, \ldots, n-1\} \to \{0,1\}^*$ is a Deflate coding iff

1. $\lceil\cdot\rceil$ is prefix free.
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:

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

$$\exists_b.\, \lceil b\rceil \neq [] \wedge \lceil b\rceil \preccurlyeq l$$

Deflate Sequences

- A *Deflate sequence* is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies *Kraft's inequality* $\sum_{i,c_i \neq 0} 2^{-a_i} \leq 1$.
- Theorem: $\lambda_{c:\{0,\ldots,n-1\}\to\{0,1\}^*}.[\text{len}\, c(x) \mid x \leftarrow [1..n]]$ is an isomorphism between the lists that satisfy Kraft's inequality and 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.

- We proved a "uniqueness" and an "existence" theorem. From the "existence" theorem, we can extract an algorithm.
- Example:
  - Alphabet $\{A, B, C, D\}$ with code lengths $\{A \to 2, B \to 1, C \to 3, D \to 3\}$

  $\Rightarrow$ Coding $\{A \to [10], B \to [0], C \to [110], D \to [111]\}$

Uncompressed Blocks

- Uncompressed data must be byte aligned.

$\Rightarrow$ We cannot "forget" about bytes and just define a grammar on a bit stream.

$\Rightarrow$ Our solution: Define a grammar on a bit stream but count the bits $\Rightarrow$ Padding can be done by skipping bits until the bit count is a multiple of 8.

The Encoding Relation

Currently (but constantly changing):

- 243 LOC definitions.
- 4340 LOC proofs of strong uniqueness and decidability.
- 14092 LOC for the whole extractable decompression program.

For actual decompression:

- Tremendous amounts of memory needed.
- Originally, decompression of even small files took weeks: We use list append often, so we got (at least) quadratic time in some places.
- Now we use `Extract Constant`, and implemented "Catenable Double-Ended Queues" (Chris Okasaki) in Haskell (unverified).

Experience and Suggestions

- Whether our definition is "correct" can only be empirically tested (because it is axiomatic) $\Rightarrow$ Relations as simple as possible, put complexity into proofs, test the extracted program.
- Tried to encapsulate relations with proofs and monadically combine them. But: The definitions became less clear.
- "Computational vs. non-computational" should be easier (autodetection, overload `inl`/`or_introl`)
- Rely on lazy evaluation (extract to Haskell): Easier to make extracted programs efficient as well as proofs readable.
- Intrinsics in Coq would be nice (for extraction we used `Extract Constant`).
- Try using combinations of given structures rather than defining new ones: Intrinsics can be used and theorems and tactics can be used easier.

Conclusion & Further Work

- The standard was probably defined with imperative low-level programming in mind.
- We chose not to care too much about efficiency by now, and focus on correctness. However, the specification can be used to verify an imperative algorithm as well.
- Decompression works. We are currently working on a compression algorithm.
- It is interesting as a case study for program extraction, and having a verified compressor-decompressor-pair is worthwile for its own sake.