Using DiffArrays to optimize Decompression

Christoph-Simon Senjak

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

ABM 2017
We formalized the Deflate compression standard in the Coq proof assistant.

We tried to utilize program extraction to create a fully verified implementation of a compression and decompression algorithm.

However, making it efficient in the absence of destructive operations is hard.

DiffArrays to the rescue!
Deflate - A very brief overview

An informal illustration of the format:

\[
\text{Deflate} ::= ('0' \text{ Block})* '1' \text{ Block} (0|1)* \\
\text{Block} ::= '00' \text{ UncompressedBlock} | \\
\quad '01' \text{ DynamicallyCompBl} | \\
\quad '10' \text{ StaticallyCompBl} \\
\text{UncompressedBlock} ::= \text{length} \sim \text{length} \text{ bytes} \\
\text{StaticallyCompBl} ::= \text{CompBl(standard coding)} \\
\text{DynamicallyCompBl} ::= \text{header coding CompBl(coding)} \\
\text{CompBl}(c) ::= [\sim 256] \times 256
\]

Specified in RFC 1951. Allows for Lempel-Ziv-compression (backreferences) and Huffman coding. Widely used (GZip, Zip, HTTP, SSH, etc).
Program Extraction

- Informal Proof
  - formalization (usually manual; some projects like Naproche try to narrow this step)
- Formal Proof
  - reconstruction
- Classical Proof Term
  - translation
- Constructive Proof Term
  - proof-to-extracted-term in Minlog
  - pruning
- Realizing Term
  - ‘Extract’ in Coq
- Program in stock programming language

- Usually manually for Agda, Idris
- ‘Qed’, ‘Defined’ in Coq
- ‘cdp’ in Minlog
- ‘Extract’ in Coq
- Compilation
Pros and cons of program extraction:

▶ Sophisticated formats should come with (at least informal) correctness proofs anyway. However, reality looks different.
▶ Proofs must be (mostly) constructive.

Alternatives in Coq:

▶ Writing the functions dependently typed.
▶ Verifying functions a posteriori.

We sometimes mix these styles when appropriate.
Backreferences

- 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).

- Example:

  ananas_banana_batata
Backreferences

- 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).

- Example:

```
2
ananas_banana_batata
| | |
+-+
```
Backreferences

- 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).

- Example:

  ```
  22
  ananas_banana_batata
  |   |
  ---
  ```
Backreferences

- 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).

- Example:

```
222
ananas_banana_batata
  |   |
  +-+
```
Backreferences

- 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).

- Example:

\[
\begin{array}{c}
222 & 8 \\
ananas\_banana\_batata \\
\mid & \mid \\
\hline
\end{array}
\]
Backreferences

- 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).

- Example:

  222  88
  ananas_banana_batata
  |     |     |
  +-----+-----+
  ^^^^^^
Backreferences

- 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).

- Example:

\[
\begin{array}{c}
222 & 888 \\
ananas\_banana\_batata \\
| & | \\
+-------+
\end{array}
\]
Backreferences

- 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).

- Example:

```
   222    8888
  ananas_banana_batata
   |       |
  +--------+
```
Backreferences

- 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).

- Example:

```
  222   88888
ananas_banana_batata
|       |
+--------+
```
Backreferences

- 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).

- Example:

```
  222   888887
ananas_banana_batata
  |   |   |
+-----+
```
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).

Example:

```
   222     8888877
ananas_banana_batata
     \   / \   /
      +-------+
```
Backreferences

- 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).

- Example:

```
222   88888777
ananas_banana_batata
|     |     |
+-----+-----+
```
Backreferences

- 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).

- Example:

```
222 88888777 2
ananas_banana_batata
  |   |
  +++
```
Backreferences

- 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).

- Example:

```
222   88888777   22
ananas_banana_batata
   |   |   |
   +-+-
```
Backreferences

- 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).

- Example:

```
222  88888777  222
ananas_banana_batata
  ||
+++
```
Backreferences

- 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).
- Example:
  
  
  $\begin{array}{cccc}
  3 & 5 & 3 & 3 \\
  222 & 88888777 & 222 \\
  \text{ananas\_banana\_batata}
  \end{array}$
Backreferences

- 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).

- Example:

\[
\begin{array}{cccc}
3 & 5 & 3 & 3 \\
222 & 88888777 & 222 \\
ananas\_banana\_batata \\
\Rightarrow \text{an} \langle 3, 2 \rangle \ s\_b \langle 5, 8 \rangle \langle 3, 7 \rangle \ t \langle 3, 2 \rangle
\end{array}
\]
Backreferences

- 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).

- Example:

\[
\begin{array}{cccc}
3 & 5 & 3 & 3 \\
222 & 88888777 & 222 \\
\text{ananas_banana_batata}
\end{array}
\]

\( \Rightarrow \) an \( \langle 3, 2 \rangle \) s b \( \langle 5, 8 \rangle \) \( \langle 3, 7 \rangle \) t \( \langle 3, 2 \rangle \)

- This is the slowest part of the decompression algorithm.

- We have
  - a purely functional and fast resolver, which is not verified
  - an implementation that utilizes a list structure with recursive slowdown, which is slow
  - an implementation using diffarrays, which is fast
Resolution in General

- Use some map structure as buffer to save the last $n$ decompressed bytes.
- If a backreference points to it, copy it to the front.
- For example, a naïve way of doing it, using a list as buffer (the second argument), is:

```haskell
resolve :: [Either a (Int, Int)] -> [a] -> [a]
resolve [] _ = []
resolve ((Left b) : r) x = b : resolve r (b : x)
resolve (Right (0, _ ) : r) x = resolve r x
resolve (Right (l , d ) : r) x =
    let b = x !! (d - 1) -- get (d-1)th element from x
    in b : resolve (Right (l-1, d ) : r) (b : x)
```
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom.
- Example of a queue of doom with 3 elements (instead of 32 KiB):

```
start [] []
```
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.

- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom

- Example of a queue of doom with 3 elements (instead of 32 KiB):
  ```
  push 1 [1] []
  ```
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom.
- Example of a queue of doom with 3 elements (instead of 32 KiB): push 2 [2;1] []
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):

```
push 3  [3;2;1] []
```
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom.
- Example of a queue of doom with 3 elements (instead of 32 KiB):
  ```
  push 4
  [4] [3;2;1] [] → 🕷️
  ```
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom.
- Example of a queue of doom with 3 elements (instead of 32 KiB): push 5 [5;4] [3;2;1]
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.

- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom.

- Example of a queue of doom with 3 elements (instead of 32 KiB): push 6 [6;5;4] [3;2;1]
Queue of Doom

- In an imperative setting, we would use a ring buffer. However, in the functional setting, even with DiffArrays, this makes things more complicated.
- We only need to save a 32KiB history, as backreferences are limited. We therefore can put two objects of 32KiB size in a Queue of Doom
- Example of a queue of doom with 3 elements (instead of 32 KiB): push 7 [7] [6; 5; 4] [3; 2; 1] → 🕷️
Using Recursive Slowdown

▶ “ExpList”:

\[
\text{Inductive ExpList } (A : \text{Set}) : \text{Set} :=
\]
\[
| \quad \text{Enil} : \text{ExpList } A \\
| \quad \text{Econs1} : A \rightarrow \text{ExpList } (A \ast A) \rightarrow \text{ExpList } A \\
| \quad \text{Econs2} : A \rightarrow A \rightarrow \text{ExpList } (A \ast A) \rightarrow \text{ExpList }
\]

▶ The advantage of ExpLists is that \text{nth} and \text{cons} consume logarithmic time.
DiffArrays

- The advantage of imperative arrays with destructive updates is their $O(1)$ modification and read.
- It is possible to embed destructive operations into linear type systems. However, Coq is not linear, and neither are Haskell (which uses Monads instead) and OCaml (which is impure).
- However, we may still write the code in a linear fashion, so it is theoretically possible for the compiler to optimize the code.
- DiffArrays are fast exactly when they are used in a linear fashion, and slow otherwise.
DiffArrays

Ref 1

Array: 1 2 3
DiffArrays

Ref 1 → (0, 1) → Ref 2

Array: 4 2 3
DiffArrays

Ref 1 $\rightarrow$ (0, 1) $\rightarrow$ Ref 2 $\rightarrow$ (2, 3) $\rightarrow$ Ref 3

Array: 4 2 5
DiffArrays

Ref 1 → (0, 1) → Ref 2 → (2, 3) → Ref 3
Ref 5 ← (1, 2) ← Ref 4 ← (0, 4)

Array: 7 8 5
The old versions can be reconstructed in $O(t)$, where $t$ is the number of versions.

Read- and Write-Access to the newest reference takes $O(1)$, as for imperative arrays.
DiffStacks

On top of the diffarrays, we put a stack structure in the usual way. Our formalization in Coq then has axioms like

\[
\text{Axiom DSPush : } \forall (A : \text{Set}) \ (a : A) \ (ds : \text{DiffStack } A), \ \text{DiffStack } A.
\]

\[
\text{Axiom DSNth : } \forall (A : \text{Set}) \ (n : \text{nat}) \ (ds : \text{DiffStack } A) \ (\text{default : } A), \ A \ast \text{DiffStack } A.
\]

\[
\text{Axiom DSNthFakeLinear : } \forall \{A : \text{Set}\} \ n \ ds \ d, \ \text{snd (@DSNth } A \ n \ ds \ d) = ds.
\]

The last axiom says that the DiffStack after reading one element from it is the same as before.
The Canterbury Corpus is a standardized set of files for benchmarking compression and decompression algorithms. Decompressing the files compressed with gzip results in the following table:

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>ExpList</th>
<th>Time (s)</th>
<th>DiffArray</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.27</td>
<td></td>
<td>0.11</td>
<td></td>
<td>grammar.lsp</td>
</tr>
<tr>
<td>4.22</td>
<td></td>
<td>0.25</td>
<td></td>
<td>fields.c</td>
</tr>
<tr>
<td>86.73</td>
<td></td>
<td>0.56</td>
<td></td>
<td>cp.html</td>
</tr>
<tr>
<td>0.42</td>
<td></td>
<td>0.14</td>
<td></td>
<td>xargs.1</td>
</tr>
<tr>
<td>177.54</td>
<td></td>
<td>0.85</td>
<td></td>
<td>sum</td>
</tr>
<tr>
<td>628.5</td>
<td></td>
<td>3.43</td>
<td></td>
<td>asyoulik.txt</td>
</tr>
<tr>
<td>727.25</td>
<td></td>
<td>4.05</td>
<td></td>
<td>alice29.txt</td>
</tr>
<tr>
<td>1958.08</td>
<td></td>
<td>10.72</td>
<td></td>
<td>lcet10.txt</td>
</tr>
<tr>
<td>2177.95</td>
<td></td>
<td>12.78</td>
<td></td>
<td>plrabn12.txt</td>
</tr>
<tr>
<td>2241.7</td>
<td></td>
<td>7.5</td>
<td></td>
<td>ptt5</td>
</tr>
<tr>
<td>3540.27</td>
<td></td>
<td>17.42</td>
<td></td>
<td>kennedy.xls</td>
</tr>
</tbody>
</table>
DiffArrays helped to boost efficiency without really losing any formal guarantees, by only slightly increasing the trusted codebase.

This seems like a good way to embed some linear type constructs into purely functional code without linear type system.

To lift this to a formal level, there has been done some work in “Adjustable References” by Vafeiadis, V.