Optimization of a verified Deflate implementation

Christoph-Simon Senjak

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

LMU TCS Oberseminar SoSe 2015

Deflate - The Compulsory Slide

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.

What we have

- Mathematical specification of a compression format that is very likely Deflate
  - Tested it with lots of randomized data
  - And with real-life data
  - However, standards are axiomatic
- Implementation of a decompression algorithm through program extraction for testing

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.

A little argument

"If the verified program is not better than compressing and decompressing with the unverified program, it is worthless":

- Assume unverified algorithms c, d. Then the following are verified algorithms:

```
vc c d x = let y = c x
           in if (d y == x) then (true, y)
                            else (false, x)
vd _ d (b, x) = if b then d x else x
```

- But this assumes that c and d are always the same on every platform.
- It does not allow later optimisations or even compatible bugfixes.
- $\Rightarrow$ better to have a formal specification and verify against it.
- Still a good "benchmark".

Problems

- Large trusted codebase:
  - Coq
  - GHC
  - Coq extraction mechanism
- Slow, probably superquadratic.
- Memory-Consuming.

Reasons and possible solutions

Coq does not have intrinsics/builtins:

- Proving stuff about `nat` is easier and more usual that about `N` (binary integers).
- It is possible to extract inductive predicates to other datatypes:

```
Extract Inductive nat ⇒
  "Prelude.Int" [ "0" "(1 Prelude.+)" ]
  "( let r z s n = case n of {
                   0 → z 0 ;
                   q → s (q Prelude.- 1)}
     in r )".
```

- But Coq by default extracts all functions (even +, *) to their recursive definitions (thus, addition takes linear time, multiplication takes quadratic time, etc.).

Overloading

It is possible to overload defined functions:

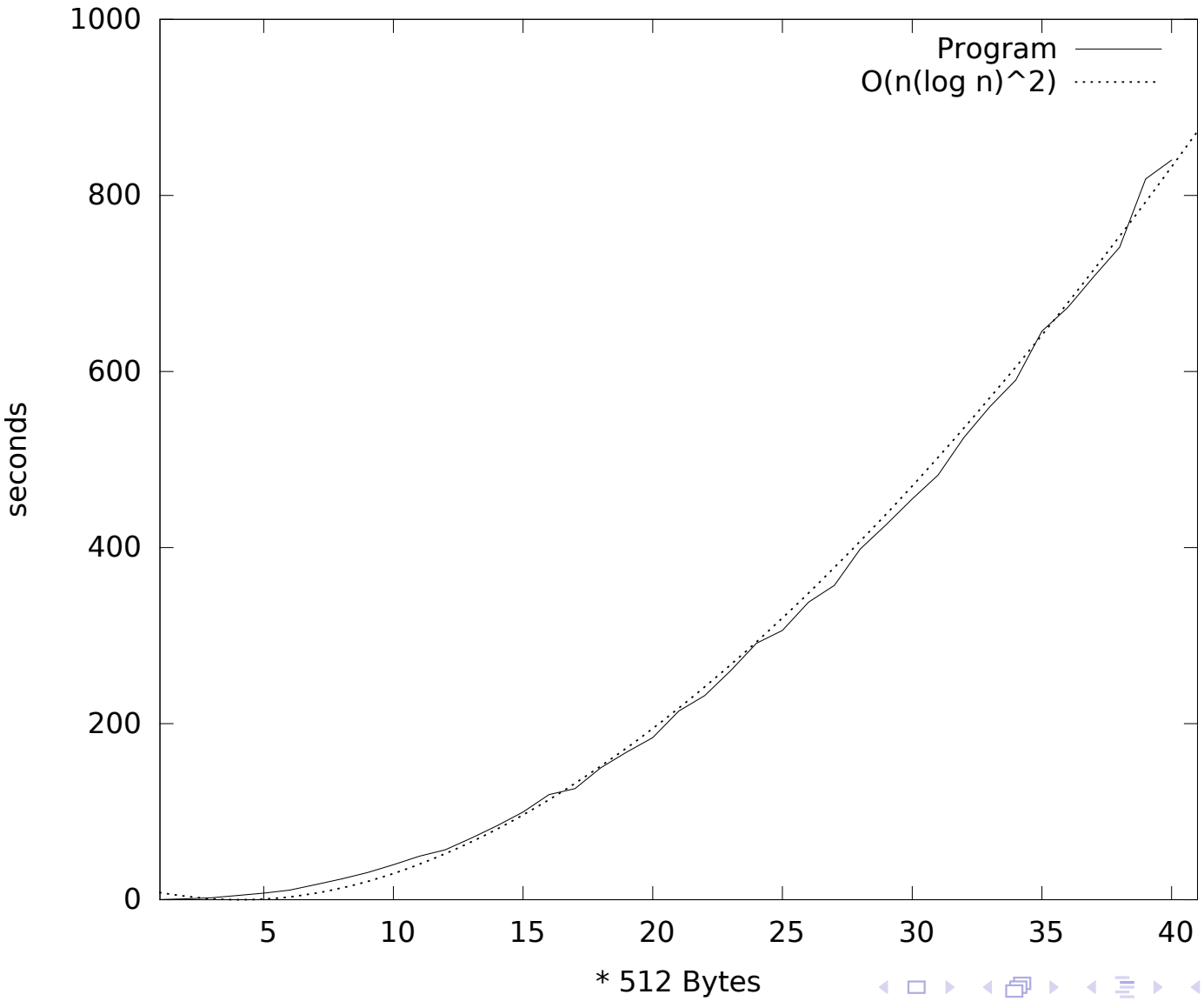
```
Extract Constant plus ⇒ "(Prelude.+)".

Extract Constant mult ⇒ "(Prelude.*)".
```

However, this makes the trusted code base larger, and we cannot overload it differently on other parts of the program.
It would be nicer to have some mechanism for this (maybe univalence will become something like that?)

Catenable Deques - 1

- Benchmarks suggested that there are lots of list `append`, `snoc` and `nth`-calls
- The advantage of lists is that they are a simple well-understood concept with a very small surface for bugs.
- The disadvantage is that the said operations are slow.
- $\Rightarrow$ We used `Extract Constant` and a catenable deque (by Chris Okasaki)

Catenable Deques - 2



Catenable Deques - 3

- The runtime is still much too slow, but at least in a reasonable complexity class.
- However, it uses tremendous amounts of memory:
  - The catenable deque structure we used prevents Haskell from reading the file lazily, terefore the whole file must be loaded
  - The file is loaded bitwise into this structure, but Haskell does not optimize bit lists: A pointer and a box is stored for every bit of the file.
- The main culprit appears to be the resolution of backreferences.

Backreferences - 1

- 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
```

- $\Rightarrow$ Works also when $l > d$, results in a repetition of already written bytes $\Rightarrow$ Run-length-encoding

Backreferences - 2

- Factored out the resolution of backreferences, it is the last thing that is done.
- Tried to use reverse lists (so `snoc`-operations are faster), but still the runtime was bad.
- With the catenable deques, it performed better but consumed too much memory.
- Current implementation uses "ExpList":

```
Inductive ExpList (A : Set) : Set :=
| Enil : ExpList A
| Econs1 : A → ExpList (A * A) → ExpList A
| Econs2 : A → A → ExpList (A * A) → ExpList A.
```

Backreferences with ExpLists

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- The output list is combined with a cache
- The cache contains two `ExpLists`
- If the first `ExpList` fills up (has as much elements as our cache should have), it becomes the second `ExpList`, and the original second `ExpList` is dismissed, and a new empty first `ExpList` is allocated $\Rightarrow$ limits memory overhead.

Performance of Backreferences

- The file `gzip-1.3.3.tar.gz` is 308K large. With backreference resolution the way we do it, it takes 46 minutes to decompress - without resolution, 15 seconds (on alicante - on siena 40 seconds).
- That is, resolution makes the implementation a magnitude slower.
- We are currently trying to use other techniques for optimizing it:
  - DiffArrays - a simple hack that will probably work satisfactory.
  - Ynot - only extracts to OCaml.
  - VST - the specifications are very complicated.