A purely functional, efficient backreference-resolver

Christoph-Simon Senjak

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

PUMA Workshop 2016
We are given a sequence of type \([A + \mathbb{N}^2]\), where \(A\) is some alphabet.

* inl \(c\) denotes the character \(c : A\).

* \(\text{inr}(l, d)\) denotes a **backreference**, which is an instruction to copy a part of the already decompressed data to the front. It has a **length** \(l\), denoting the number of characters to be copied to the front, and a **distance** \(d\), denoting the number of characters to go back the decompression history before copying.

* In practice, \(l\) and \(d\) are limited. In Deflate, \(d\) is limited by 32KiB.
A simple algorithm:

```
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
```

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

Examples:
- `ananas_banana_batata` ⇒ `an ⟨3, 2⟩ s_b ⟨5, 8⟩ ⟨3, 7⟩ t ⟨3, 2⟩`
- `aaaaaaaargh!` ⇒ `a ⟨7, 1⟩ rgh!`
Since $d$ is limited, use a 32KiB ring buffer to save a limited history.

```cpp
struct char_ref { bool is_ref; union { char ch; int backref[2];};};
const int hlen = 32768;

void decomp (std::list<char_ref>& input, std::list<char>& output) {
    char history[hlen];
    int ptr = 0;
    for (auto c : input) {
        if (c.is_ref) {
            for (int l = 0; l < c.backref[0]; ++l) {
                char d = history[(ptr - c.backref[1] + hlen) % hlen];
                output.push_back(history[ptr] = d);
                ptr = (ptr + 1) % hlen;
            }
        } else {
            output.push_back(history[ptr] = c.ch);
            ptr = (ptr + 1) % hlen;
        }
    }
}
```
Use list as buffer.

```haskell
data BR a = Char a | Backref Int Int
res :: [BR a] -> [a] -> [a]
res [] _ = []
res (Char a : input) buf = a : res input (a : buf)
res (Backref 0 d : input) buf = res input buf
res (Backref l d : input) buf =
  let a = buf !! (d - 1)
  in a : res (Backref (l - 1) d : input) (a : buf)
```

⇒ takes linear memory and quadratic time

However, we can get this approach to linear space and time by using a more sophisticated buffer structure. However, it still performs badly.
Our implementation uses two Pairing Heaps.

Pairing Heaps are purely functional priority queues which do not have a decrease-key operation.

find-min has amortized constant time, insert has amortized constant time, delete-min has amortized logarithmic time.

We have a verified implementation of pairing heaps in Coq.
To simplify the problem, we first apply a trivial transformation that replaces any backreference by a sequence of backreferences of length 1. It is then sufficient to save a sequence \([A + \mathbb{N}]\).

```haskell
data BR a = Char a | Backref Int Int
data BR_ a = Char_ a | Backref_ Int

un :: [BR a] -> [BR_ a]
un [] = []
un (Char a : b) = Char_ a : un b
un (Backref 0 a : b) = un b
un (Backref n a : b) = Backref_ a : un (Backref (n - 1) a : b)
```

We can do this lazily, so it will not require extra memory.

This step is not strictly required, but makes the rest of the algorithm much easier.

Example: \(an\langle 3,2\rangle s_b\langle 5,8\rangle \langle 3,7\rangle t\langle 3,2\rangle\) will become

\[
\langle 2\rangle \langle 2\rangle \langle 2\rangle s_b \langle 8\rangle \langle 8\rangle \langle 8\rangle \langle 8\rangle \langle 7\rangle \langle 7\rangle \langle 7\rangle t \langle 2\rangle \langle 2\rangle \langle 2\rangle
\]
Some concepts

- We use **absolute** positions of characters in our lists.
- A backreference has an absolute **source** position, an absolute **destination** position, and a **content**, which is the character it will make the algorithm copy.
- Our algorithm will destructure the list, but additionally track the absolute position where it currently is.
- Our algorithm will work at **two** positions of the list.
To illustrate the idea, we first show a simpler two-step algorithm on lists. It consists of two parts.

In the first part, we iterate over the input list, and collect all backreferences as pairs (destination, source).

\[
\text{collect}_\:\::\:\text{Int} \to [\text{BR}_\:\text{a}] \to [(\text{Int}, \text{Int})]
\]

\[
\text{collect}_\:\_\_ [\_] = [\_]
\]

\[
\text{collect}_\:\_ \:(\text{Char}_\:\_ \:) \:\_\text{r} = \text{collect}_\:\_ \:(\text{n} + 1) \:\text{r}
\]

\[
\text{collect}_\:\_ \:(\text{Backref}_\:\text{b}) \:\_\text{r} = (\text{n}, \text{n} - \text{b}) : \text{collect}_\:\_ \:(\text{n} + 1) \:\text{r}
\]

Example:

\[
[(0, 2), (1, 3), (2, 4), (0, 8), (1, 9), (2, 10), (3, 11), (4, 12), (6, 13), (7, 14), (8, 15), (15, 17), (16, 18), (17, 19)]
\]
We now sort the pairs lexicographically

collect :: [BR_ a] -> [(Int, Int)]
collect = sort . (collect_ 0)

Our example becomes [(0,2), (0,8), (1,3), (1,9), (2,4), (2,10), (3,11),
(4,12), (6,13), (7,14), (8,15), (15,17), (16,18), (17,19)]

Notice that now, the backreferences are sorted in the order their sources occur.
A simpler algorithm with lists - 3

We can resolve the backreferences from such a list with the following algorithm:

Let $m$ be some generic map structure, initially empty. The current absolute position in the input list is saved in a variable $n$, initially 0.

1. If the sorted backreference list is not empty, remove its first element and store it as $(s, d)$. Otherwise, proceed at step 4.
2. If $s \neq n$, proceed at step 4.
3. If $s = n$, there is a backreference to the current position $n$. Peek an element from the input.
   3a. If it is Char, then set $m[d] = c$, and recur at step 1.
   3b. If it is Backref, then set $m[d] = m[n]$, and recur at step 1.
4. Read an element from the input.
   4a. If we are at the end of the input, end.
   4b. If we read a character Char, write $c$ to the output.
   4c. If we read a backreference Backref, write $m[n]$ to the output.
5. Set $n = n + 1$ and recur at step 1.
A simpler algorithm with lists - 4

In Haskell:

```haskell
resolve_ :: [BR a] -> [(Int, Int)] -> Int -> Map Int a -> [a]
resolve_ l r n m =
  let res l r n m =
      case l of
        [] -> []
        (Char_ c : l') -> c : resolve_ l' r (n + 1) m
        (Backref_ _ : l') -> (m ! n) : resolve_ l' r (n + 1) m
  in case r of
    [] -> res l r n m
    ((s, d) : r') ->
      if (s == n) then
        case l of
          (Char_ c : l') -> resolve_ l r' n (insert d c m)
          (Backref_ _ : l') -> resolve_ l r' n (insert d (m!n) m)
      else res l r n m
```

We only ever use the table to look up the current position. We never look at anything smaller than the current position again ⇒ priority queue of destination-character-pairs sorted according to the destination.
A simpler algorithm with lists - 4

In Haskell:

```haskell
case l of
  (Char_ c : l') -> c : resolve_ l' r (n + 1) m
  (Backref_ _ : l') -> (m ! n) : resolve_ l' r (n + 1) m
```

We only ever use the table to look up the current position. We never look at anything smaller than the current position again ⇒ priority queue of destination-character-pairs sorted according to the destination.
A simpler algorithm with lists - 5

```haskell
resolve_ :: Ord a => [BR_ a] -> [(Int, Int)] -> Int -> MinQueue (Int, a) -> [a]
resolve_ l r n m =
  let res l r n m =
      case l of
        [] -> []
        (Char_ c : l') -> c : resolve_ l' r (n + 1) m
        (Backref_ _ : l') ->
          let (_, nm) = findMin m
          in nm : resolve_ l' r (n + 1) (deleteMin m)
    in case r of
      [] -> res l r n m
      ((s, d) : r') ->
        if (s == n)
        then
          case l of
            (Char_ c : l') -> resolve_ l r' n (insert (d, c) m)
            (Backref_ _ : l') ->
              let (_, nm) = findMin m
              in resolve_ l r' n (insert (d, nm) m)
        else res l r n m
```

Christoph-Simon Senjak (LMU München)
Instead of reading the source-destination-pairs into a list and sorting it, we can directly read it into a priority queue:

```haskell
collect_ :: Int -> [BR a] -> [(Int, Int)]
collect_ _ [] = []
collect_ n ((Char _ _) : r) = collect_ (n + 1) r
collect_ n ((Backref b) : r) = (n - b, n) : collect_ (n + 1) r

collect :: [BR a] -> MinQueue (Int, Int)
collect = fromList . (collect_ 0)
```
A simpler algorithm with lists - 7

```
resolve_ l r n m =
  let res l r n m =
    case l of
    [[] -> []
    [Char_ c : l'] -> c : resolve_ l' r (n + 1) m
    [Backref_ _ : l'] ->
      let (_, nm) = findMin m
      in nm : resolve_ l' r (n + 1) (deleteMin m)
    in case minView r of
    Nothing -> res l r n m
    Just ((s, d), r') ->
      if (s == n)
        then
          case l of
            [Char_ c : l'] -> resolve_ l r' n (insert (d, c) m)
            [Backref_ _ : l'] ->
              let (_, nm) = findMin m
              in resolve_ l r' n (insert (d, nm) m)
          else res l r n m
```
A simpler algorithm with lists - 7

```hs
resolve l r n m =
    let res l r n m =
        case l of
            [] -> []
            (Char c : l') -> c : resolve l' r (n + 1) m
            (Backref _ : l') ->
                let (_, nm) = findMin m
                in nm : resolve l' r (n + 1) (deleteMin m)
        in case minView r of
            Nothing -> res l r n m
            Just ((s, d), r') ->
                if (s == n)
                then
                    case l of
                        (Char c : l') -> resolve l r' n (insert (d, c) m)
                        (Backref _ : l') ->
                            let (_, nm) = findMin m
                            in resolve l r' n (insert (d, nm) m)
                else res l r n m
    in res l r n m
```

The distance $d - s$ is limited by $D = 32768$ ⇒ we only have to read 32768 elements in advance.
We write a coroutine that fills a queue with new elements, and a start-function that calls it 32768 times in advance. For strings shorter than 32768 bytes, this is equivalent to our old collect function.

```haskell
proceed_collect :: (Int, [BR_ a], MinQueue (Int, Int)) ->
                 (Int, [BR_ a], MinQueue (Int, Int))
proceed_collect (n, [], m) = (n, [], m)
proceed_collect (n, Char_ _ : r, m) = (n + 1, r, m)
proceed_collect (n, Backref_ b : r, m) =
                                        (n + 1, r, insert (n - b, n) m)

start_collect_ :: Int -> (Int, [BR_ a], MinQueue (Int, Int)) ->
                 (Int, [BR_ a], MinQueue (Int, Int))
start_collect_ 0 x = x
start_collect_ n x = start_collect_ (n - 1) (proceed_collect x)

start_collect :: [BR_ a] -> (Int, [BR_ a], MinQueue (Int, Int))
start_collect x = start_collect_ 32768 (0, x, empty)
The final algorithm - 2

We now need to call this coroutine whenever a new character is produced:

```haskell
resolve_ l r@(rn, rl, rq) n m =
  let
    res l r n m =
      case l of
        [] -> []
        (Char c : l') -> c : resolve_ l' (proceed_collect r) (n + 1) m
        (Backref _ : l') ->
          let (_, nm) = findMin m
          in nm : resolve_ l' (proceed_collect r) (n + 1) (deleteMin m)
    in case minView rq of
      Nothing -> res l r n m
      Just ((s, d), r') ->
        if (s == n)
          then case l of
            (Char c : l') -> resolve_ l (rn, rl, r') n (insert (d, c) m)
            (Backref _ : l') ->
              let (_, nm) = findMin m
              in resolve_ l (rn, rl, r') n (insert (d, nm) m)
          else res l r n m
```

Christoph-Simon Senjak (LMU München)
Final Remarks

Advantages of our algorithm:
- Purely functional
- $O(1)$ space, $O(n)$ time

Disadvantages:
- Intricate
- Hard to formally verify

Alternatives:
- State monads (also hard to verify in Coq)
- Adjustable references (not really pure)
- Linear types (not possible in Coq)