An Implementation of Deflate in Coq

Christoph-Simon Senjak

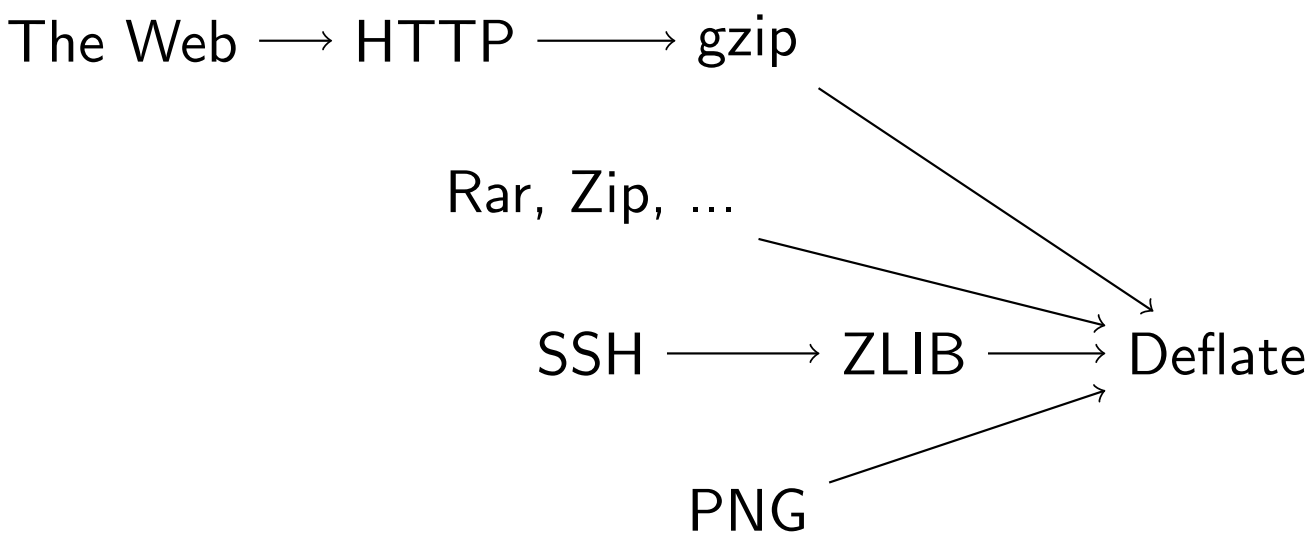
Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

Formal Methods 2016 Limassol, Cyprus

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. There are several different implementations.

Why Deflate?



Main implementation is the zlib (zlib.net).
Though the first release was 1995, in 2005 there were still security vulnerabilities.

Deflate - A very brief 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.

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
  - $\Rightarrow$ the best we can do, since the formal specification is axiomatic, and the standard is informally specified.
- Implementation of a compression and a decompression algorithm through program extraction
  - Verified to be inverse to each other.
  - Still slow, but "awaitable", lots of potential for optimization.

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.
Our experience was that extracting to Haskell and utilizing laziness makes it a lot easier to write proofs from which useful algorithms can be extracted.

Verified from Unverified Compressors?

The following is a theoretical way of creating a verified compression-decompression-pair from an unverified such pair:

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

$$c'x = \begin{cases} (\top, cx) & \text{for } d(cx) = x \\ (\bot, x) & \text{otherwise} \end{cases}$$

$$d'x = \begin{cases} dy & \text{for } x = (\top, y) \\ y & \text{for } x = (\bot, y) \end{cases}$$

- But this assumes that c and d are always the same on every platform. It is prone to digital obsolescence.
- There is no understandable formal specification of the algorithm $\Rightarrow$ 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". We are still far away from that, but we think it is possible.

Deflate Codings

The RFC says:
The Huffman codes used for each alphabet in the "deflate" format have two additional rules:

- All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent;
- Shorter codes lexicographically precede longer codes.

However:

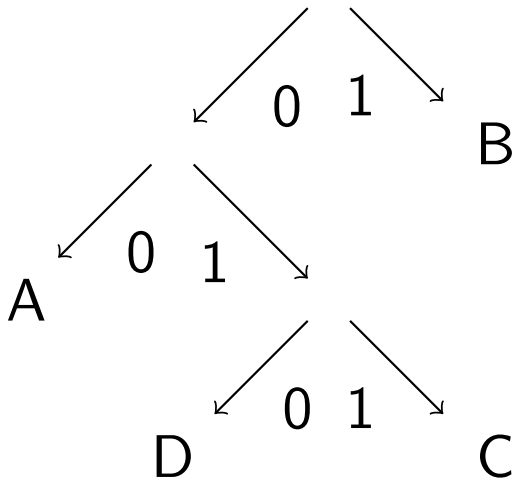
- Being a Huffman-Code is an optimality property regarding a given string, not a criterion on a code itself.
- We worked out an axiomatization that is equivalent for Huffman codings, but does not require any optimality, but adding a "density" condition.

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.

Prefix-Free Codings

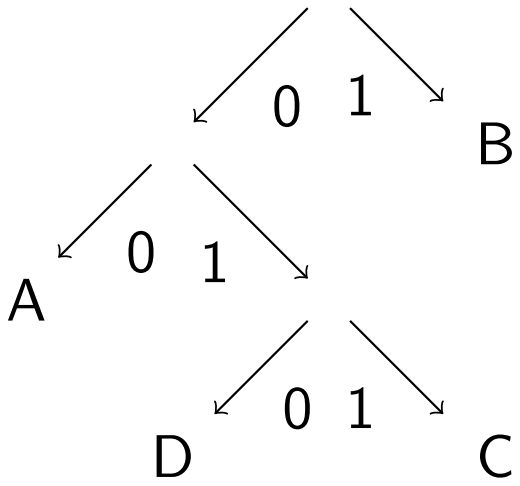
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- The coding is prefix-free $\Rightarrow$ we get a coding tree

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



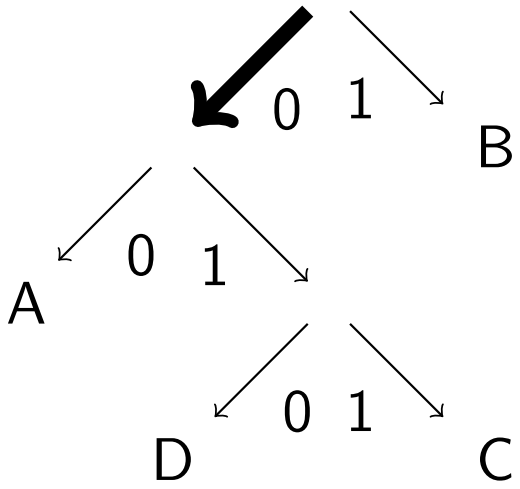
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- The coding is prefix-free $\Rightarrow$ we get a coding tree

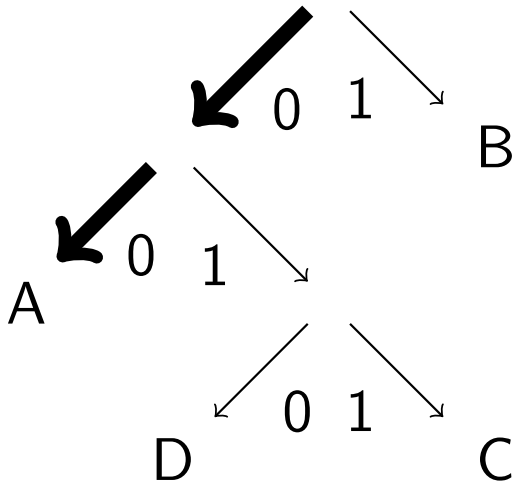
$\Rightarrow$ encoding/decoding injective:

$\downarrow$

00011001

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- The coding is prefix-free $\Rightarrow$ we get a coding tree

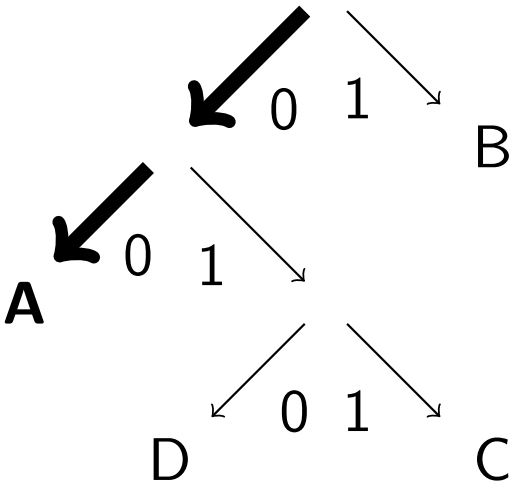
$\Rightarrow$ encoding/decoding injective:

$\downarrow\downarrow$

00011001

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



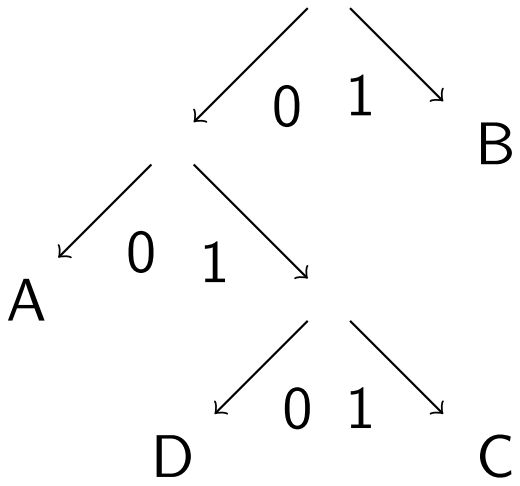
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
A

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



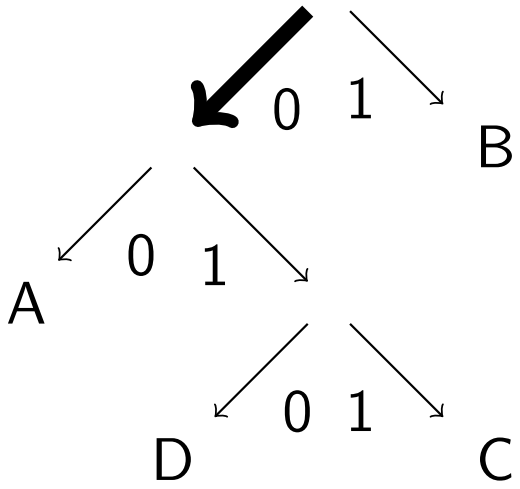
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
A

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



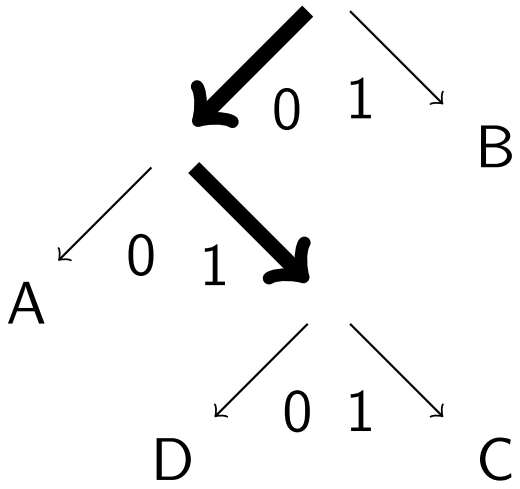
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$\underset{A}{\overset{\downarrow}{00}011001}$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- The coding is prefix-free $\Rightarrow$ we get a coding tree

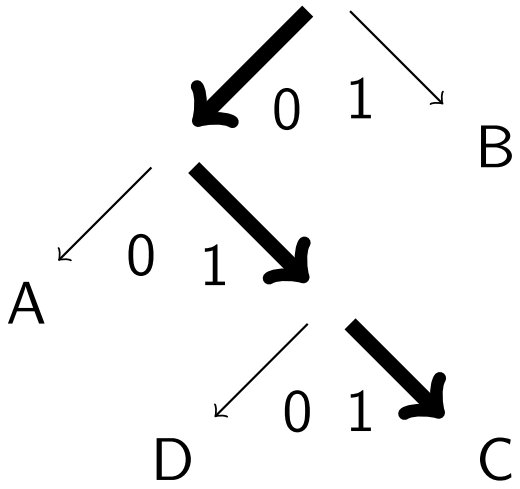
$\Rightarrow$ encoding/decoding injective:

$$\overset{\downarrow\downarrow}{00}011001$$

$$A$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



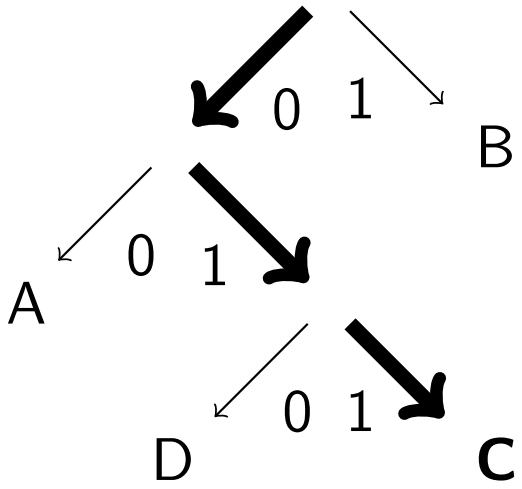
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$\overset{\downarrow\downarrow\downarrow}{00011001}$$
$$A$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



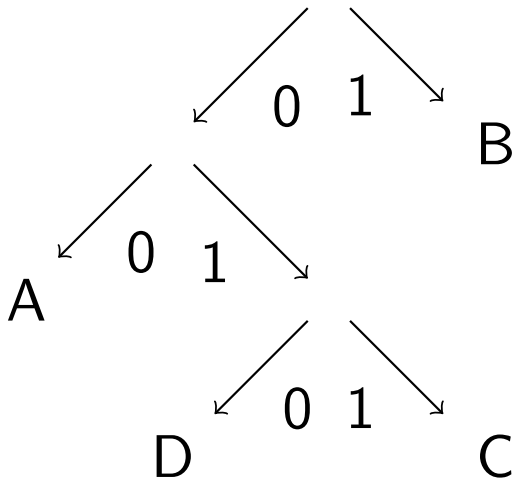
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
A C

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



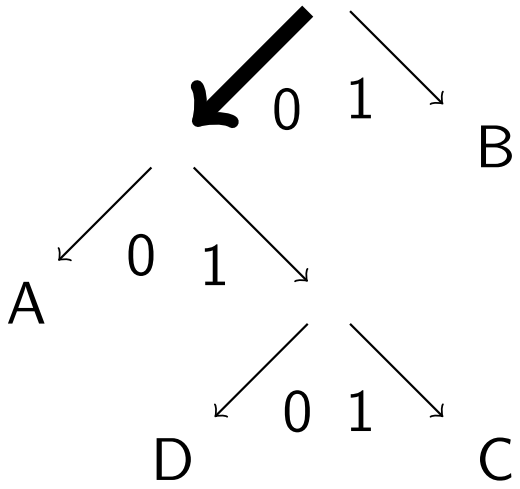
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
A C

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



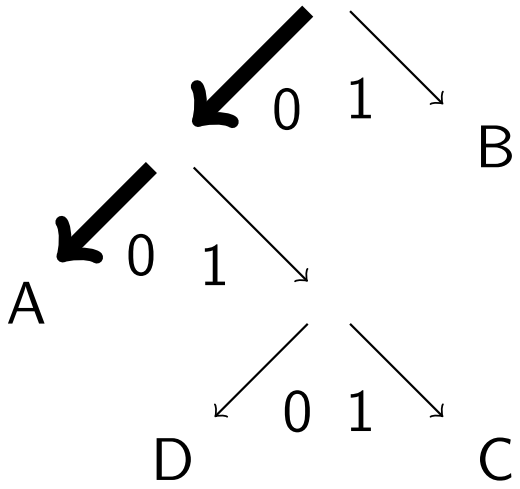
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$\begin{array}{c} \downarrow \\ 00011001 \\ A\ \ C \end{array}$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



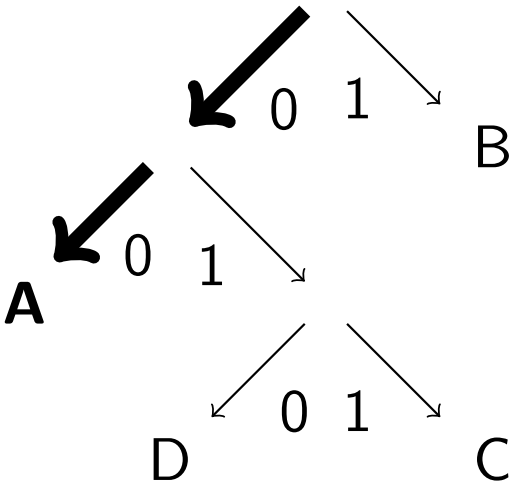
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$\begin{array}{cccccccc} & & & & \downarrow & \downarrow & & \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ A & & & & C & & & \end{array}$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



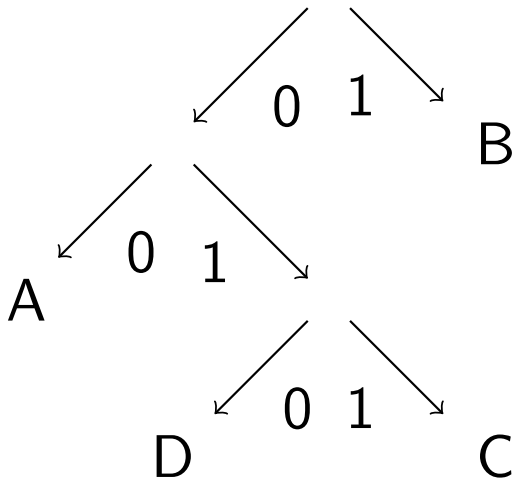
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$\begin{array}{cccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ & & & & \downarrow & \downarrow & & \\ A & & & & C & & A & \end{array}$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



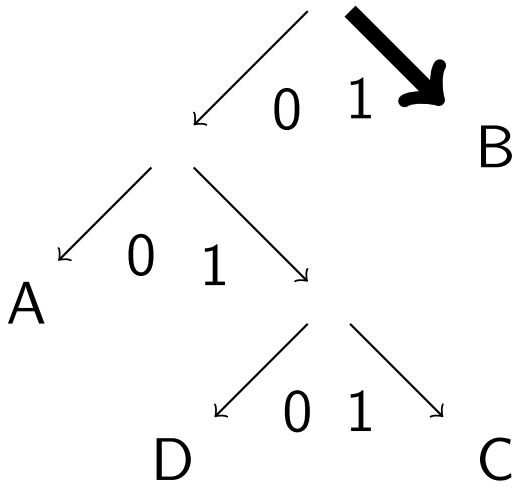
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
$A \quad C \quad A$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



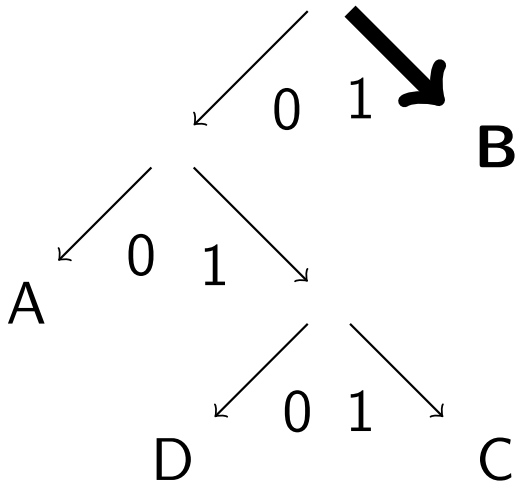
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

$$00011001$$
$$\downarrow$$
$$A\ \ C\ \ A$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



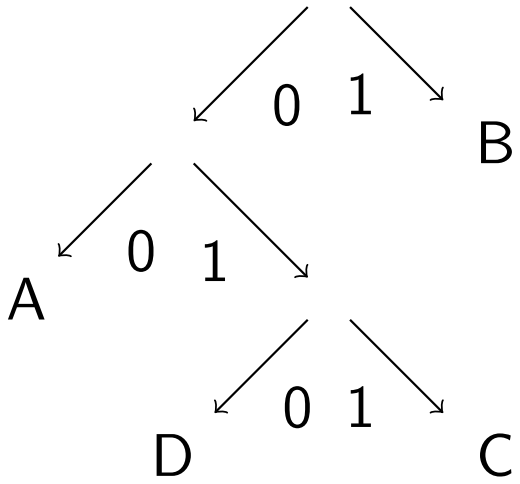
- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
$A \quad C \quad A\,B$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- The coding is prefix-free $\Rightarrow$ we get a coding tree

$\Rightarrow$ encoding/decoding injective:

00011001
$A \quad C \quad A\,B$

Mathematically, we exclude the code of length 0, because it usually denotes that some code does not occur.

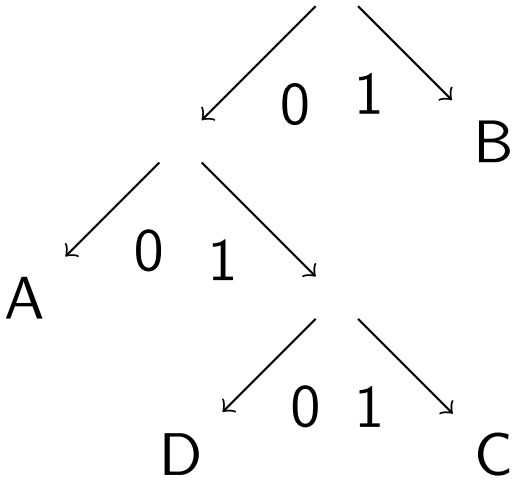
$$\forall_{ab}.a \neq b \to \lceil a \rceil \neq [] \to \lceil a \rceil \preccurlyeq \lceil b \rceil \to \bot$$

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.

Prefix-Free Codings

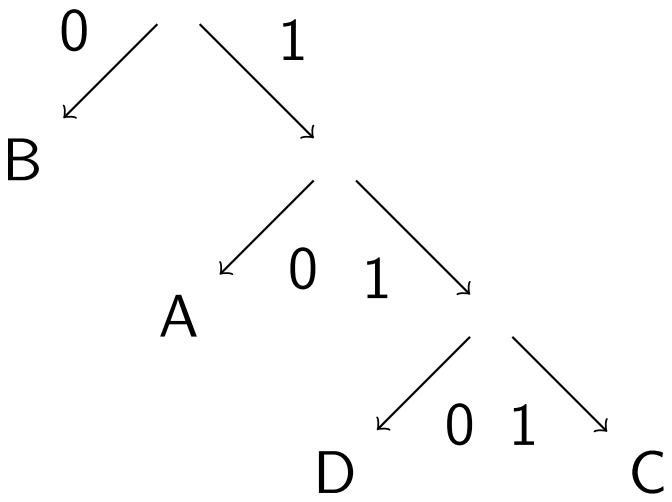
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Shorter codes lexicographically precede longer codes.

Prefix-Free Codings

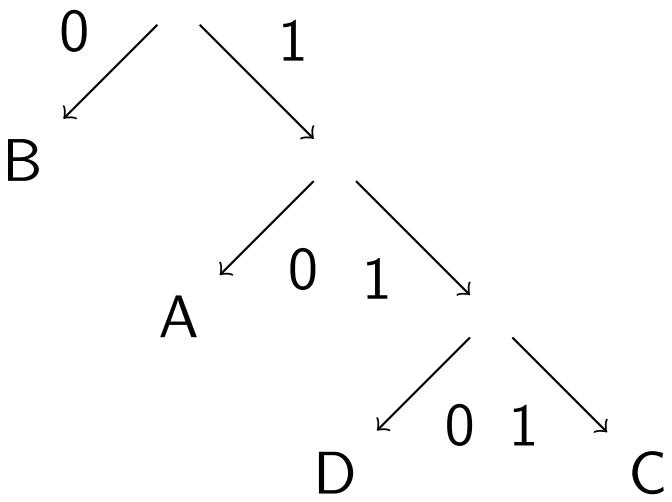
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Shorter codes lexicographically precede longer codes.

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.

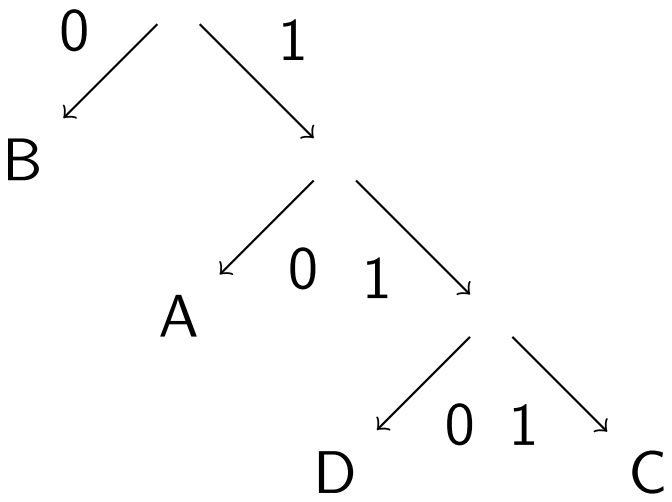


- Shorter codes lexicographically precede longer codes.

$$\forall_{ab}.\, \mathsf{len}\lceil a \rceil < \mathsf{len}\lceil b \rceil \to \lceil a \rceil \sqsubseteq \lceil b \rceil$$

Prefix-Free Codings

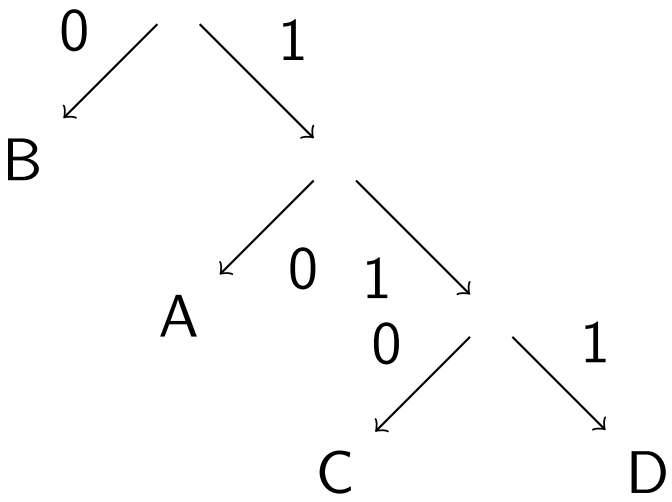
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Codes of equal length are sorted according to the characters they encode.

Prefix-Free Codings

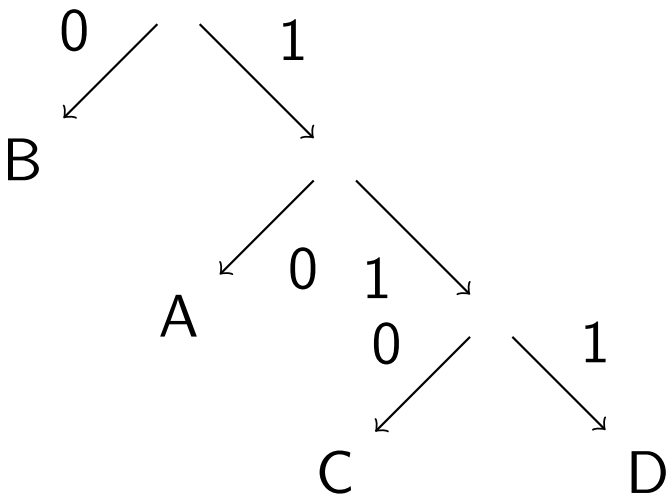
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Codes of equal length are sorted according to the characters they encode.

Prefix-Free Codings

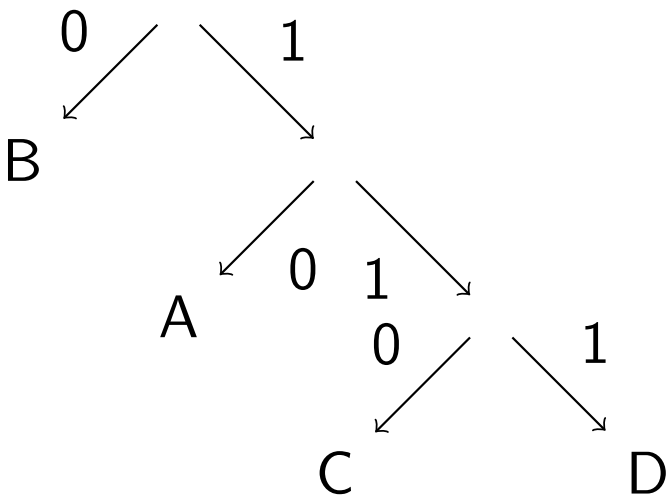
Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Codes of equal length are sorted according to the characters they encode.
- The RFC wants consecutivity here. We omit it at this point.

Prefix-Free Codings

Consider $A \to 00$, $B \to 1$, $C \to 011$, $D \to 010$.



- Codes of equal length are sorted according to the characters they encode.
- The RFC wants consecutivity here. We omit it at this point.

$$\forall_{ab}.\, \mathsf{len}\lceil a \rceil = \mathsf{len}\lceil b \rceil \to a \leq b \to \lceil a \rceil \sqsubseteq \lceil b \rceil$$

Deflate Codings

- Consider $0 \to [0], 1 \to [1,0,1], 2 \to [1,1,0], 3 \to [1,1,1]$
- This coding has a "gap" between $[0]$ and $[1,0,1]$.
- We need another property that differs from the RFC that forbids such gaps:

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

Deflate Codings

In conclusion, we say that $\lceil\cdot\rceil : \{0,\ldots,n-1\} \to \{0,1\}$ is a **Deflate Coding**, if

$$\forall_{ab}. a \neq b \to \lceil a\rceil \neq [] \to \lceil a\rceil \preccurlyeq \lceil b\rceil \to \bot$$

$$\forall_{ab}. \operatorname{len}\lceil a\rceil < \operatorname{len}\lceil b\rceil \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

$$\forall_{ab}. \operatorname{len}\lceil a\rceil = \operatorname{len}\lceil b\rceil \to a \leq b \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

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

Theorem: There is a 1:1 correspondence between such codings and the sequences $(a_i)_i$ of code-lengths (or 0 if the code does not occur) which satisfy **Kraft's inequality** $\sum_{i,a_i\neq 0} 2^{-a_i} \leq 1$. $\Rightarrow$ Only a sequence of code lengths has to be saved to reconstruct the coding.

Uniqueness and Existence

This 1:1 correspondence can be split into uniqueness and existence:

- Uniqueness proves an equality, and will therefore only be used for specification, and not extracted to an algorithm or used at runtime.
- Existence will be extracted to an algorithm, so we need to care about efficiency.
- We use a recursive algorithm similar to the algorithm in RFC 1951, but not the same, as the standardized algorithm makes extensive use of stateful operations.
- We prove the existence of Deflate codings for Deflate sequences, and extracted a working algorithm from this existence proof.

A Theory of Parsers - 1

We work with relations of the form `output -> input -> Prop`. Such a relation must satisfy two conditions to be suitable for parsing:

- **Strong Uniqueness**, meaning that for a given string, there is at most one initial segment that can be parsed:

$$\begin{aligned}\text{StrongUnique}\, R \quad :\Leftrightarrow \forall_{a,b,l_a,l'_a,l_b,l'_b}.\quad & l_a +\!\!+ l'_a = l_b +\!\!+ l'_b \rightarrow \\ & R\, a\, l_a \rightarrow R\, b\, l_b \rightarrow \\ & a = b \wedge l_a = l_b\end{aligned}$$

- **Strong Decidability**, meaning that whether an initial segment exists is decidable.

$$\begin{aligned}\text{StrongDec}\, R :\Leftrightarrow \forall_l.\qquad & (\exists_{a,x,y}. l = x +\!\!+ y \wedge R\, a\, x) \vee \\ & (\neg(\exists_{a,x,y}. l = x +\!\!+ y \wedge R\, a\, x) \wedge \quad E)\end{aligned}$$

  where E is some error type. This resembles an exception monad.
- Parsing relations can be combined in a monadic fashion.

A Theory of Parsers - 2

- Advantages of our theory:
  - It is simple, it needs no sophisticated library.
  - It yields algorithms that can be extracted directly from proofs.
- Disadvantages:
  - The input must be total. For cototal lists, some relations would not be decidable.
  - Combining exception monads consumes stack space.

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).
- Example:

```
ananas_banana_batata
```

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).
- Example:

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

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).
- Example:

```
   22
ananas_banana_batata
 | |
 +-+
```

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).
- Example:

```
   222
ananas_banana_batata ⇐ Run-Length-Encoding
   | |
   +-+
```

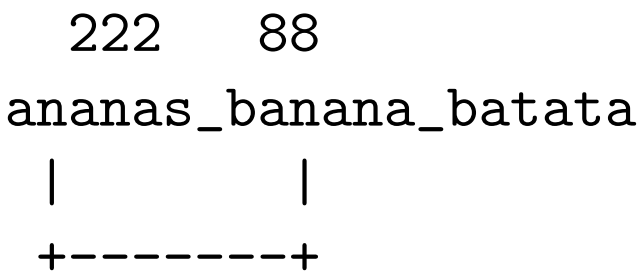
Backreferences - 1

- 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).
- Example:

```
  222    8
ananas_banana_batata
|       |
+-------+
```

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).
- Example:



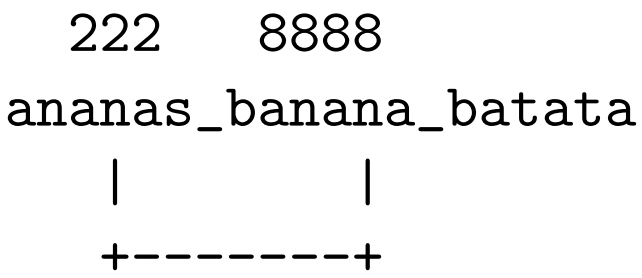
Backreferences - 1

- 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).
- Example:

```
   222    888
ananas_banana_batata
   |        |
   +--------+
```

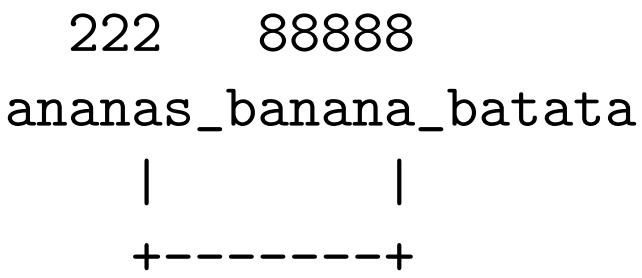
Backreferences - 1

- 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).
- Example:



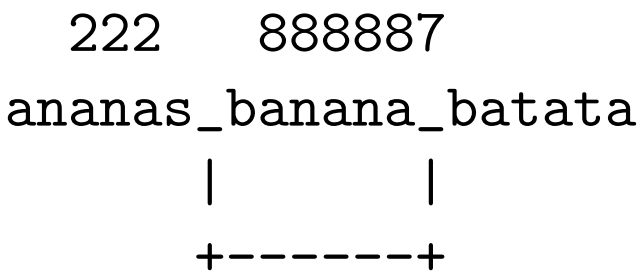
Backreferences - 1

- 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).
- Example:



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).
- Example:



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).
- Example:

```
  222    8888877
ananas_banana_batata
       |       |
       +------+
```

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).
- Example:

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

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).
- Example:

```
  222    88888777 2
ananas_banana_batata
                  | |
                  +-+
```

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).
- Example:

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

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).
- Example:

```
  222    88888777 222
ananas_banana_batata ⇐ Run-Length-Encoding
                   | |
                   +-+
```

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).
- Example:

```
   3        5    3    3
  222    88888777 222
ananas_banana_batata
```

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).
- Example:

```
   3        5    3    3
  222    88888777 222
ananas_banana_batata
```

$\Rightarrow$ `an` $\langle 3, 2 \rangle$ `s_b` $\langle 5, 8 \rangle$ $\langle 3, 7 \rangle$ `t` $\langle 3, 2 \rangle$

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).
- Example:

  3 5 3 3
  222 88888777 222
  `ananas_banana_batata`
  $\Rightarrow$ `an` $\langle 3, 2 \rangle$ `s_b` $\langle 5, 8 \rangle$ $\langle 3, 7 \rangle$ `t` $\langle 3, 2 \rangle$

- The resolution of backreferences has been the major bottleneck from the beginning. We tried several optimizations (using reverse lists, or Okasaki's catenable deques).

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):
  `start` `[]` `[]`

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):

```
push 1          [1] []
```

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):

```
push 2        [2;1] []
```

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):

```
push 3     [3;2;1] []
```

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` in a **Queue of Doom**
- Example of a queue of doom with 3 elements (instead of 32 KiB):
  `push 4` `[4]` `[3;2;1]` `[]` → ☠

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` 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]
```

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` 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]
```

Backreferences - 2

- 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.
```

- The advantage of `ExpLists` is that `nth` and `cons` consume logarithmic time.
- We only need to save a 32KiB history, as backreferences are limited. We therefore use two `ExpLists` 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]` → 

Further Work

- Iteratees are structures that should be similar to our strongly decidable and strongly unique relations, but do not directly require lazy i/o and totality.
- We have tested the performance of several (not yet verified) efficient algorithms for resulution of backreferences.
  - A purely functional algorithm that uses pairing heaps.
  - Using adjustable references (DiffArrays).
- Having a verified and highly optimized low-level implementation (for example in Bedrock or CompCert) would be an ultimate goal.