# A Combinator Library for Prefix-Free Codes 

Emin Karayel

March 11, 2024


#### Abstract

This entry contains a set of binary encodings for primitive data types, such as natural numbers, integers, floating-point numbers as well as combinators to construct encodings for products, lists, sets or functions of/between such types.

For natural numbers and integers, the entry contains various encodings, such as Elias-Gamma-Codes and exponential Golomb Codes, which are efficient variable-length codes in use by current compression formats.

A use-case for this library is measuring the persisted size of a complex data structure without having to hand-craft a dedicated encoding for it, independent of Isabelle's internal representation.


## 1 Introduction

```
theory Prefix-Free-Code-Combinators
    imports
        HOL-Library.Extended-Real
        HOL-Library.Float
        HOL-Library.FuncSet
        HOL-Library.List-Lexorder
        HOL-Library.Log-Nat
        HOL-Library.Sublist
begin
```

The encoders are represented as partial prefix-free functions. The advantage of prefix free codes is that they can be easily combined by concatenation. The approach of using prefix free codes (on the byte-level) for the representation of complex data structures is common in many industry encoding libraries (cf. [2]).
The reason for representing encoders using partial functions, stems from some use-cases where the objects to be encoded may be in a much smaller sets, as their type may suggest. For example a natural number may be known to have a given range, or a function may be encodable because it has a finite domain.

Note: Prefix-free codes can also be automatically derived using Huffmans' algorithm, which was formalized by Blanchette [1]. This is especially useful if it is possible to transmit a dictionary before the data. On the other hand these standard codes are useful, when the above is impractical and/or the distribution of the input is unknown or expected to be close to the one's implied by standard codes.
The following section contains general definitions and results, followed by Section 3 to 10 where encoders for primitive types and combinators are defined. Each construct is accompanied by lemmas verifying that they form prefix free codes as well as bounds on the bit count to encode the data. Section 11 concludes with a few examples.

## 2 Encodings

fun opt-prefix where
opt-prefix (Some $x$ ) (Some $y)=$ prefix $x y \mid$
opt-prefix - - = False
definition opt-comp $x y=($ opt-prefix $x y \vee o p t-p r e f i x ~ y ~ x) ~$
fun opt-append $::$ bool list option $\Rightarrow$ bool list option $\Rightarrow$ bool list option where opt-append $($ Some $x)($ Some $y)=$ Some $(x @ y) \mid$ opt-append -- = None
lemma opt-comp-sym: opt-comp $x$ y $=$ opt-comp y $x$〈proof〉
lemma opt-comp-append:
assumes opt-comp (opt-append $x$ y) z
shows opt-comp $x z$
$\langle$ proof $\rangle$
lemma opt-comp-append-2:
assumes opt-comp $x$ (opt-append $y z$ )
shows opt-comp $x y$
$\langle p r o o f\rangle$
lemma opt-comp-append-3:
assumes opt-comp (opt-append $x$ y) (opt-append $x z$ )
shows opt-comp y z
$\langle p r o o f\rangle$
type-synonym 'a encoding $=$ ' $a \rightharpoonup$ bool list

```
definition is-encoding :: 'a encoding \(\Rightarrow\) bool
    where is-encoding \(f=(\forall x y\). opt-prefix \((f x)(f y) \longrightarrow x=y)\)
```

An encoding function is represented as partial functions into lists of booleans，where each list element represents a bit．Such a function is defined to be an encoding，if it is prefix－free on its domain．This is similar to the formalization by Hibon and Paul－ son［4］except for the use of partial functions for the practical reasons described in Section 1.
lemma is－encodingI：
assumes $\bigwedge x x^{\prime}$ y $y^{\prime}$ ．e $x=$ Some $x^{\prime} \Longrightarrow e y=$ Some $y^{\prime} \Longrightarrow$ prefix $x^{\prime} y^{\prime} \Longrightarrow x=y$
shows is－encoding e
〈proof〉
lemma is－encodingI－2：
assumes $\backslash x y$ ．opt－comp $(e x)(e y) \Longrightarrow x=y$
shows is－encoding e
〈proof〉
lemma encoding－triv：is－encoding Map．empty
$\langle p r o o f\rangle$
lemma is－encodingD：
assumes is－encoding $e$
assumes opt－comp（ex）（ey）
shows $x=y$
〈proof〉
lemma encoding－imp－inj：
assumes is－encoding $f$
shows inj－on $f(\operatorname{dom} f)$
〈proof〉
fun bit－count ：：bool list option $\Rightarrow$ ereal where
bit－count None $=\infty \mid$
bit－count $($ Some $x)=$ ereal $($ length $x)$
lemma bit－count－finite－imp－dom：
bit－count $(f x)<\infty \Longrightarrow x \in \operatorname{dom} f$
〈proof〉
lemma bit－count－append：
bit－count（opt－append $x y$ ）$=$ bit－count $x+$ bit－count $y$
$\langle p r o o f\rangle$

## 3 (Dependent) Products

definition encode-dependent-prod ::
'a encoding $\Rightarrow\left(' a \Rightarrow{ }^{\prime} b\right.$ encoding $) \Rightarrow\left({ }^{\prime} a \times ' b\right)$ encoding
(infixr $\bowtie_{e} 65$ )
where
encode-dependent-prod e $f x=$ opt-append $(e(f s t x))(f(f s t x)(s n d x))$
lemma dependent-encoding:
assumes is-encoding e1
assumes $\bigwedge x . x \in$ dom e1 $\Longrightarrow$ is-encoding $(e 2 x)$
shows is-encoding $\left(e 1 \bowtie_{e}\right.$ e2 $)$
〈proof〉
lemma dependent-bit-count:

```
bit-count \(\left(\left(e_{1} \bowtie_{e} e_{2}\right)\left(x_{1}, x_{2}\right)\right)=\)
        bit-count \(\left(e_{1} x_{1}\right)+\) bit-count \(\left(e_{2} x_{1} x_{2}\right)\)
\(\langle p r o o f\rangle\)
```

lemma dependent-bit-count-2:
bit-count $\left(\left(e_{1} \bowtie_{e} e_{2}\right) x\right)=$ bit-count $\left(e_{1}(f s t x)\right)+$ bit-count $\left(e_{2}(f s t x)(\operatorname{snd} x)\right)$ $\langle p r o o f\rangle$

This abbreviation is for non-dependent products.
abbreviation encode-prod ::
'a encoding $\Rightarrow$ ' $b$ encoding $\Rightarrow\left({ }^{\prime} a \times\right.$ 'b) encoding
(infixr $\times_{e}$ 65)
where
encode-prod e1 e2 $\equiv e 1 \bowtie_{e}(\lambda-. e 2)$

## 4 Composition

lemma encoding-compose:
assumes is-encoding $f$
assumes inj-on $g\{x . p x\}$
shows is-encoding ( $\lambda x$. if $p x$ then $f(g x)$ else None)
$\langle p r o o f\rangle$
lemma encoding-compose-2:
assumes is-encoding $f$
assumes inj $g$
shows is-encoding $(\lambda x . f(g x))$
$\langle p r o o f\rangle$

## 5 Natural Numbers

```
fun encode-bounded-nat :: nat }=>\mathrm{ nat }=>\mathrm{ bool list where
    encode-bounded-nat (Suc l) n=
        (let r=n \geq(2^l) in r#encode-bounded-nat l (n-of-bool r*2`l))
    encode-bounded-nat 0-= []
lemma encode-bounded-nat-prefix-free:
    fixes uv l:: nat
    assumes u<2`l
    assumes v<2`l
    assumes prefix (encode-bounded-nat l u) (encode-bounded-nat l v)
    shows }u=
    \langleproof\rangle
definition }N\mp@subsup{b}{e}{}:: nat => nat encodin
    where }N\mp@subsup{b}{e}{}ln=
        if n<l
            then Some (encode-bounded-nat (floorlog 2 (l-1)) n)
            else None)
Nb}\mp@subsup{e}{e}{l}\mathrm{ is encoding for natural numbers strictly smaller than l
using a fixed length encoding.
lemma bounded-nat-bit-count:
    bit-count (Nb, l y) = (if y<l then floorlog 2 (l-1) else }\infty
<proof\rangle
lemma bounded-nat-bit-count-2:
    assumes y<l
    shows bit-count (Nb e l y) = floorlog 2 (l-1)
    <proof\rangle
lemma dom (N\mp@subsup{b}{e}{}l)={..<l}
    \langleproof\rangle
lemma bounded-nat-encoding: is-encoding ( }N\mp@subsup{b}{e}{}l\mathrm{ l)
\langleproof\rangle
fun encode-unary-nat :: nat => bool list where
    encode-unary-nat (Suc l) = False#(encode-unary-nat l)|
    encode-unary-nat 0 = [True]
lemma encode-unary-nat-prefix-free:
    fixes uv :: nat
    assumes prefix (encode-unary-nat u) (encode-unary-nat v)
    shows u=v
    <proof\rangle
```

definition $N u_{e}::$ nat encoding

```
where \(N u_{e} n=\) Some (encode-unary-nat \(n\) )
```

$N u_{e}$ is encoding for natural numbers using unary encoding．It is inefficient except for special cases，where the probability of large numbers decreases exponentially with its magnitude．

```
lemma unary-nat-bit-count:
    bit-count \(\left(N u_{e} n\right)=\) Suc \(n\)
    \(\langle p r o o f\rangle\)
lemma unary-encoding: is-encoding \(N u_{e}\)
        \(\langle p r o o f\rangle\)
```

Encoding for positive numbers using Elias-Gamma code.
definition $N g_{e}::$ nat encoding where
$N g_{e} n=$
(if $n>0$
then $\left(N u_{e} \bowtie_{e}\left(\lambda r . N b_{e}\left(\mathcal{Z}^{\widehat{ } r}\right)\right)\right)$
(let $r=$ floorlog $2 n-1$ in $(r, n-2 \widehat{2}))$
else None)
$N g_{e}$ is an encoding for positive numbers using Elias－Gamma en－ coding［3］．
lemma elias－gamma－bit－count： bit－count $\left(N g_{e} n\right)=($ if $n>0$ then $2 *\lfloor\log 2 n\rfloor+1$ else $(\infty::$ ereal $))$ $\langle p r o o f\rangle$
lemma elias－gamma－encoding：is－encoding $N g_{e}$〈proof〉
definition $N_{e}::$ nat encoding where $N_{e} x=N g_{e}(x+1)$
$N_{e}$ is an encoding for all natural numbers using exponential Golomb encoding［6］．Exponential Golomb codes are also used in video compression applications［5］．
lemma exp－golomb－encoding：is－encoding $N_{e}$
$\langle p r o o f\rangle$
lemma exp－golomb－bit－count－exact：

$$
\text { bit-count }\left(N_{e} n\right)=2 *\lfloor\log 2(n+1)\rfloor+1
$$

$\langle p r o o f\rangle$

```
lemma exp-golomb-bit-count:
    bit-count \(\left(N_{e} n\right) \leq(2 * \log 2(\) real \(n+1)+1)\)
    〈proof〉
lemma exp-golomb-bit-count-est
    assumes \(n \leq m\)
    shows bit-count \(\left(N_{e} n\right) \leq(2 * \log 2(\) real \(m+1)+1)\)
\(\langle p r o o f\rangle\)
```


## 6 Integers

definition $I_{e}::$ int encoding where
$I_{e} x=N_{e}($ nat $($ if $x \leq 0$ then $(-2 * x)$ else $(2 * x-1)))$
$I_{e}$ is an encoding for integers using exponential Golomb codes by embedding the integers into the natural numbers, specifically the positive numbers are embedded into the odd-numbers and the negative numbers are embedded into the even numbers. The embedding has the benefit, that the bit count for an integer only depends on its absolute value.
lemma int-encoding: is-encoding $I_{e}$ $\langle p r o o f\rangle$
lemma int-bit-count: bit-count $\left(I_{e} n\right)=2 *\lfloor\log 2(2 *|n|+1)\rfloor+1$ $\langle p r o o f\rangle$
lemma int-bit-count-1:
assumes abs $n>0$
shows bit-count $\left(I_{e} n\right)=2 *\lfloor\log 2|n|\rfloor+3$
$\langle p r o o f\rangle$
lemma int-bit-count-est-1:
assumes $|n| \leq r$
shows bit-count $\left(I_{e} n\right) \leq 2 * \log 2(r+1)+3$
$\langle p r o o f\rangle$
lemma int-bit-count-est:
assumes $|n| \leq r$
shows bit-count $\left(I_{e} n\right) \leq 2 * \log 2(2 * r+1)+1$
$\langle$ proof $\rangle$

## 7 Lists

```
definition \(L f_{e}\) where
    \(L f_{e}\) e \(n x s=\)
        (if length \(x s=n\)
            then fold \((\lambda x y\). opt-append \(y(e x)) x s\) (Some [])
            else None)
```

$L f_{e} e n$ is an encoding for lists of length $n$, where the elements are encoding using the encoder $e$.
lemma fixed-list-encoding:
assumes is-encoding e
shows is-encoding ( $L f_{e}$ e $n$ )
$\langle p r o o f\rangle$
lemma fixed-list-bit-count:

```
    bit-count (Lf e e n xs) =
    (if length xs = n then ( }\sumx\leftarrowxs.(bit-count (ex))) else \infty
<proof>
definition }\mp@subsup{L}{e}{
    where L}\mp@subsup{L}{e}{e}exs=(N\mp@subsup{u}{e}{}\mp@subsup{\bowtie}{e}{}(\lambdan.L\mp@subsup{f}{e}{}en))(length xs,xs
```

$L_{e} e$ is an encoding for arbitrary length lists, where the elements are encoding using the encoder $e$.
lemma list-encoding:
assumes is-encoding e
shows is-encoding ( $\left.L_{e} e\right)$
$\langle p r o o f\rangle$
lemma sum-list-triv-ereal:
fixes $a$ :: ereal
shows sum-list (map ( $\lambda$-. a) $x s$ ) $=$ length $x s * a$
$\langle p r o o f\rangle$
lemma list-bit-count:
bit-count $\left(L_{e}\right.$ e xs $)=\left(\sum x \leftarrow x s\right.$. bit-count $\left.(e x)+1\right)+1$
$\langle p r o o f\rangle$

## 8 Functions

definition encode-fun :: 'a list $\Rightarrow$ ' $b$ encoding $\Rightarrow(' a \Rightarrow$ 'b) encoding (infixr $\rightarrow_{e} 65$ ) where
encode-fun xs e $f=$ (if $f \in$ extensional (set xs) then $\left(L f_{e} e(l e n g t h x s)(\operatorname{map} f x s)\right)$ else None)
$x s \rightarrow_{e} e$ is an encoding for functions whose domain is set $x s$, where the values are encoding using the encoder $e$.

```
lemma fun-encoding:
    assumes is-encoding e
    shows is-encoding (xs \(\left.\rightarrow_{e} e\right)\)
\(\langle p r o o f\rangle\)
lemma fun-bit-count:
    bit-count \(\left(\left(x s \rightarrow_{e} e\right) f\right)=\)
        (if \(f \in\) extensional (set xs) then \(\left(\sum x \leftarrow x s\right.\). bit-count \(\left.(e(f x))\right)\)
else \(\infty\) )
    \(\langle p r o o f\rangle\)
lemma fun-bit-count-est:
    assumes \(f \in\) extensional (set xs)
    assumes \(\bigwedge x . x \in\) set \(x s \Longrightarrow\) bit-count \((e(f x)) \leq a\)
```

```
    shows bit-count \(\left(\left(x s \rightarrow_{e} e\right) f\right) \leq \operatorname{ereal}(\) real \((\) length \(x s)) * a\)
```

$\langle p r o o f\rangle$

## 9 Finite Sets

definition $S_{e}::$ 'a encoding $\Rightarrow$ 'a set encoding where
$S_{e}$ e $S=$
(if finite $S \wedge S \subseteq$ dom e
then $\left(L_{e}\right.$ e (linorder.sorted-key-list-of-set $(\leq)($ the $\circ$ e) $\left.S)\right)$ else None)
$S_{e} e$ is an encoding for finite sets whose elements are encoded using the encoder $e$.

```
lemma set-encoding:
    assumes is-encoding e
    shows is-encoding (S Se)
<proof>
lemma set-bit-count:
    assumes is-encoding e
    shows bit-count (Se e S) = (if finite S then ( }\sumx\inS.bit-count (
x)+1)+1 else \infty)
<proof>
lemma sum-triv-ereal:
    fixes a :: ereal
    assumes finite S
    shows (\sum-\inS.a) = card S*a
<proof>
lemma set-bit-count-est:
    assumes is-encoding f
    assumes finite S
    assumes card S\leqm
    assumes 0\leqa
    assumes }\x.x\inS\Longrightarrowbit-count (fx)\leq
    shows bit-count (S S fS) \leqereal (real m)*(a+1)+1
<proof\rangle
```


## 10 Floating point numbers

definition $F_{e}$ where $F_{e} f=\left(I_{e} \times_{e} I_{e}\right)($ mantissa $f$, exponent $f)$
lemma float-encoding:
is-encoding $F_{e}$
〈proof〉
lemma suc-n-le-2-pow-n:

```
fixes \(n\) :: nat
shows \(n+1 \leq 2^{\wedge} n\)
\(\langle p r o o f\rangle\)
lemma float-bit-count-1:
bit-count \(\left(F_{e} f\right) \leq 6+2 *(\log 2(\mid\) mantissa \(f \mid+1)+\)
    \(\log 2(\mid\) exponent \(f \mid+1))(\) is ?lhs \(\leq\) ?rhs \()\)
\(\langle p r o o f\rangle\)
```

The following establishes an estimate for the bit count of a floating point number in non-normalized representation:

```
lemma float-bit-count-2:
    fixes \(m\) :: int
    fixes \(e::\) int
    defines \(f \equiv\) float-of ( \(m * 2\) powr \(e\) )
    shows bit-count \(\left(F_{e} f\right) \leq\)
        \(6+2 *(\log 2(|m|+2)+\log 2(|e|+1))\)
\(\langle p r o o f\rangle\)
lemma float-bit-count-zero:
    bit-count \(\left(F_{e}(\right.\) float-of 0\(\left.)\right)=2\)
    \(\langle p r o o f\rangle\)
```

end

## 11 Examples

```
theory Examples
    imports Prefix-Free-Code-Combinators
begin
```

The following introduces a few examples for encoders:

```
notepad
begin
    define example1 where example1 = Ne}\mp@subsup{\}{e}{}\mp@subsup{N}{e}{
```

This is an encoder for a pair of natural numbers using exponential Golomb codes.

Given a pair it is possible to estimate the number of bits necessary to encode it using the bit-count lemmas.

```
have bit-count (example1 (0,1)) = 4
by (simp add:example1-def dependent-bit-count exp-golomb-bit-count-exact)
```

Note that a finite bit count automatically implies that the encoded element is in the domain of the encoding function. This
means usually it is possible to establish a bound on the size of the datastructure and verify that the value is encodable simultaneously.

```
hence \((0,1) \in\) dom example1
    by (intro bit-count-finite-imp-dom, simp)
define example2
    where example2 \(=[0 . .<42] \rightarrow_{e} \mathrm{Nb}_{e} 314\)
```

The second example illustrates the use of the combinator $\left(\rightarrow_{e}\right)$, which allows encoding functions with a known finite encodable domain, here we assume the values are smaller than $314:^{\prime} a$ on the domain $\left\{. .<42::^{\prime} a\right\}$.

```
have bit-count (example2 f) \(=42 * 9\) (is ?lhs \(=\) ? \(r h s\) )
    if \(a: f \in\{0 . .<42\} \rightarrow_{E}\{0 . .<314\}\) for \(f\)
proof -
    have ?lhs \(=\left(\sum x \leftarrow[0 . .<42]\right.\). bit-count \(\left.\left(N b_{e} 314(f x)\right)\right)\)
        using \(a\) by (simp add:example2-def fun-bit-count PiE-def)
    also have \(\ldots=\left(\sum x \leftarrow[0 . .<42]\right.\). ereal (floorlog 2 313) \()\)
        using a Pi-def PiE-def bounded-nat-bit-count
        by (intro arg-cong[where \(f=\) sum-list \(]\) map-cong, auto)
    also have...\(=\) ?rhs
        by (simp add: compute-floorlog sum-list-triv)
    finally show ?thesis by simp
qed
define example3
    where example3 \(=N_{e} \bowtie_{e}\left(\lambda n .[0 . .<42] \rightarrow_{e} N b_{e} n\right)\)
```

The third example is more complex and illustrates the use of dependent encoders, consider a function with domain $\{. .<42\}$ whose values are natural numbers in the interval $\{. .<n\}$. Let us assume the bound is not known in advance and needs to be encoded as well. This can be done using a dependent product encoding, where the first component encodes the bound and the second component is an encoder parameterized by that value.

```
end
```

end

## References

[1] J. C. Blanchette. The textbook proof of huffman's algorithm. Archive of Formal Proofs, Oct. 2008. https://isa-afp.org/entries/ Huffman.html, Formal proof development.
[2] C. Bormann and P. E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, Dec. 2020.
[3] P. Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2):194-203, 1975.
[4] Q. Hibon and L. C. Paulson. Source coding theorem. Archive of Formal Proofs, Oct. 2016. https://isa-afp.org/entries/Source Coding_Theorem.html, Formal proof development.
[5] I. E. Richardson. H. 264 Transform and Coding, chapter 7, pages 179-221. John Wiley \& Sons, Ltd, 2010.
[6] J. Teuhola. A compression method for clustered bit-vectors. Information Processing Letters, 7(6):308-311, 1978.

