A Combinator Library for Prefix-Free Codes

Emin Karayel

March 24, 2023

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* |
  opt-prefix - - = *False*

definition *opt-comp x y* = (*opt-prefix x y* ∨ *opt-prefix y x*)

fun *opt-append* :: *bool list option* ⇒ *bool list option* ⇒ *bool list option*
  where
    opt-append (*Some x*) (*Some y*) = *Some* (*x*@*y*) |
    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*
⟨*proof*⟩

lemma *opt-comp-append-2*:
  assumes *opt-comp x* (*opt-append y z*)
  shows *opt-comp x y*
  ⟨*proof*⟩

lemma *opt-comp-append-3*:
  assumes *opt-comp* (*opt-append x y*) (*opt-append x z*)
  shows *opt-comp y z*
  ⟨*proof*⟩

type-synonym *'a encoding* = *'a* ⇀ *bool list*

definition *is-encoding* :: *'a encoding* ⇒ *bool*
  where *is-encoding f* = (∀ *x y*. *opt-prefix* (*f x*) (*f y*) ⟶ *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 Paulson [4] except for the use of partial functions for the practical reasons described in Section 1.

lemma *is-encodingI*:
  assumes ⋀*x x' y y'*. *e x* = *Some x'* ⟹ *e y* = *Some y'* ⟹
    prefix x' y' ⟹ *x* = *y*
  shows *is-encoding e*
⟨*proof*⟩

lemma *is-encodingI-2*:
  assumes ⋀*x y* . *opt-comp* (*e x*) (*e y*) ⟹ *x* = *y*
  shows *is-encoding e*
  ⟨*proof*⟩

lemma *encoding-triv*: *is-encoding Map.empty*
  ⟨*proof*⟩

lemma *is-encodingD*:
  assumes *is-encoding e*
  assumes *opt-comp* (*e x*) (*e y*)
  shows *x* = *y*
  ⟨*proof*⟩

lemma *encoding-imp-inj*:
  assumes *is-encoding f*
  shows *inj-on f* (*dom f*)
  ⟨*proof*⟩

fun *bit-count* :: *bool list option* ⇒ *ereal* **where**
  bit-count None = ∞ |
  bit-count (*Some x*) = *ereal* (*length x*)

lemma *bit-count-finite-imp-dom*:
  bit-count (*f x*) < ∞ ⟹ *x* ∈ *dom f*
  ⟨*proof*⟩

lemma *bit-count-append*:
  bit-count (*opt-append x y*) = *bit-count x* + *bit-count y*
  ⟨*proof*⟩

3 (Dependent) Products

definition *encode-dependent-prod* ::
$'a$ *encoding* $\Rightarrow$ ($'a \Rightarrow 'b$ *encoding*) $\Rightarrow$ ($'a \times 'b$) *encoding*
(**infixr** $\bowtie_e$ *65*)
where
encode-dependent-prod e f x =
opt-append (*e* (*fst x*)) (*f* (*fst x*) (*snd x*))

lemma *dependent-encoding*:
assumes *is-encoding e1*
assumes $\bigwedge x.\ x \in dom\ e1 \Longrightarrow$ *is-encoding* (*e2 x*)
shows *is-encoding* ($e1 \bowtie_e e2$)
⟨*proof*⟩

lemma *dependent-bit-count*:
bit-count (($e_1 \bowtie_e e_2$) (x_1,x_2)) =
bit-count (e_1 x_1) + *bit-count* (e_2 x_1 x_2)
⟨*proof*⟩

lemma *dependent-bit-count-2*:
bit-count (($e_1 \bowtie_e e_2$) *x*) =
bit-count (e_1 (*fst x*)) + *bit-count* (e_2 (*fst x*) (*snd x*))
⟨*proof*⟩

This abbreviation is for non-dependent products.

abbreviation *encode-prod* ::
$'a$ *encoding* $\Rightarrow$ $'b$ *encoding* $\Rightarrow$ ($'a \times 'b$) *encoding*
(**infixr** $\times_e$ *65*)
where
encode-prod e1 e2 $\equiv$ $e1 \bowtie_e (\lambda$-. $e2)$

4 Composition

lemma *encoding-compose*:
assumes *is-encoding f*
assumes *inj-on g* {*x. p x*}
shows *is-encoding* (λx. *if p x then f* (*g x*) *else None*)
⟨*proof*⟩

lemma *encoding-compose-2*:
assumes *is-encoding f*
assumes *inj g*
shows *is-encoding* (λx. *f* (*g x*))
⟨*proof*⟩

5 Natural Numbers

fun *encode-bounded-nat* :: *nat* ⇒ *nat* ⇒ *bool list* **where**
  encode-bounded-nat (*Suc l*) *n* =
    (*let r* = *n* ≥ (*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 *u v l* :: *nat*
  assumes *u* < *2^l*
  assumes *v* < *2^l*
  assumes *prefix* (*encode-bounded-nat l u*) (*encode-bounded-nat l v*)
  shows *u* = *v*
  ⟨*proof*⟩

definition Nb_e :: *nat* ⇒ *nat encoding*
  where Nb_e *l n* = (
    if n < *l*
      then Some (*encode-bounded-nat* (*floorlog 2* (*l−1*)) *n*)
      else None)

Nb_e *l* is encoding for natural numbers strictly smaller than *l* using a fixed length encoding.

lemma *bounded-nat-bit-count*:
  bit-count (Nb_e *l y*) = (*if y* < *l then floorlog 2* (*l−1*) *else* ∞)
⟨*proof*⟩

lemma *bounded-nat-bit-count-2*:
  assumes *y* < *l*
  shows *bit-count* (Nb_e *l y*) = *floorlog 2* (*l−1*)
  ⟨*proof*⟩

lemma *dom* (Nb_e *l*) = {..<*l*}
  ⟨*proof*⟩

lemma *bounded-nat-encoding*: *is-encoding* (Nb_e *l*)
⟨*proof*⟩

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 *u v* :: *nat*
  assumes *prefix* (*encode-unary-nat u*) (*encode-unary-nat v*)
  shows *u* = *v*
  ⟨*proof*⟩

definition Nu_e :: *nat encoding*

where $Nu_e\ n = Some\ (encode\text{-}unary\text{-}nat\ n)$

Nu_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*:
$bit\text{-}count\ (Nu_e\ n) = Suc\ n$
$\langle proof \rangle$

lemma *unary-encoding*: $is\text{-}encoding\ Nu_e$
$\langle proof \rangle$

Encoding for positive numbers using Elias-Gamma code.

definition $Ng_e :: nat\ encoding$ **where**
$Ng_e\ n =$
$(if\ n > 0$
$then\ (Nu_e \bowtie_e (\lambda r.\ Nb_e\ (2\hat{\ }r)))$
$(let\ r = floorlog\ 2\ n - 1\ in\ (r,\ n - 2\hat{\ }r))$
$else\ None)$

Ng_e is an encoding for positive numbers using Elias-Gamma encoding[3].

lemma *elias-gamma-bit-count*:
$bit\text{-}count\ (Ng_e\ n) = (if\ n > 0\ then\ 2 * \lfloor log\ 2\ n \rfloor + 1\ else\ (\infty::ereal))$
$\langle proof \rangle$

lemma *elias-gamma-encoding*: $is\text{-}encoding\ Ng_e$
$\langle proof \rangle$

definition $N_e :: nat\ encoding$ **where** $N_e\ x = Ng_e\ (x+1)$

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*: $is\text{-}encoding\ N_e$
$\langle proof \rangle$

lemma *exp-golomb-bit-count-exact*:
$bit\text{-}count\ (N_e\ n) = 2 * \lfloor log\ 2\ (n+1) \rfloor + 1$
$\langle proof \rangle$

lemma *exp-golomb-bit-count*:
$bit\text{-}count\ (N_e\ n) \le (2 * log\ 2\ (real\ n+1) + 1)$
$\langle proof \rangle$

lemma *exp-golomb-bit-count-est*:
assumes $n \le m$
shows $bit\text{-}count\ (N_e\ n) \le (2 * log\ 2\ (real\ m+1) + 1)$
$\langle proof \rangle$

6 Integers

definition I_e :: *int encoding* **where**
  $I_e\ x = N_e\ (nat\ (if\ x \le 0\ then\ (-2 * x)\ else\ (2{*}x{-}1)))$

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
⟨*proof*⟩

lemma *int-bit-count*: $bit\text{-}count\ (I_e\ n) = 2 * \lfloor log\ 2\ (2{*}|n|{+}1) \rfloor + 1$
⟨*proof*⟩

lemma *int-bit-count-1*:
  assumes $abs\ n > 0$
  shows $bit\text{-}count\ (I_e\ n) = 2 * \lfloor log\ 2\ |n| \rfloor + 3$
⟨*proof*⟩

lemma *int-bit-count-est-1*:
  assumes $|n| \le r$
  shows $bit\text{-}count\ (I_e\ n) \le 2 * log\ 2\ (r{+}1) + 3$
⟨*proof*⟩

lemma *int-bit-count-est*:
  assumes $|n| \le r$
  shows $bit\text{-}count\ (I_e\ n) \le 2 * log\ 2\ (2{*}r{+}1) + 1$
⟨*proof*⟩

7 Lists

definition Lf_e **where**
  $Lf_e\ e\ n\ xs =$
    $(if\ length\ xs = n$
      $then\ fold\ (\lambda x\ y.\ opt\text{-}append\ y\ (e\ x))\ xs\ (Some\ [])$
      $else\ None)$

$Lf_e\ e\ n$ 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 $is\text{-}encoding\ (Lf_e\ e\ n)$
⟨*proof*⟩

lemma *fixed-list-bit-count*:

bit-count $(Lf_e\ e\ n\ xs) =$
$(\textit{if length } xs = n \textit{ then } (\sum x \leftarrow xs.\ (\textit{bit-count}\ (e\ x)))\ \textit{else}\ \infty)$
⟨*proof*⟩

definition L_e
where $L_e\ e\ xs = (Nu_e \bowtie_e (\lambda n.\ Lf_e\ e\ n))\ (\textit{length}\ xs,\ xs)$

$L_e\ e$ 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* $(L_e\ e)$
⟨*proof*⟩

lemma *sum-list-triv-ereal*:
fixes a :: *ereal*
shows *sum-list* $(\textit{map}\ (\lambda\text{-}.\ a)\ xs) = \textit{length}\ xs * a$
⟨*proof*⟩

lemma *list-bit-count*:
bit-count $(L_e\ e\ xs) = (\sum x \leftarrow xs.\ \textit{bit-count}\ (e\ x) + 1) + 1$
⟨*proof*⟩

8 Functions

definition *encode-fun* :: $'a\ \textit{list} \Rightarrow\ 'b\ \textit{encoding} \Rightarrow ('a \Rightarrow\ 'b)\ \textit{encoding}$
(**infixr** $\rightarrow_e$ *65*) **where**
encode-fun xs e f =
$(\textit{if } f \in \textit{extensional}\ (\textit{set}\ xs)$
then $(Lf_e\ e\ (\textit{length}\ xs)\ (\textit{map}\ f\ xs))$
else None)

$xs \rightarrow_e e$ is an encoding for functions whose domain is *set xs*, where the values are encoding using the encoder e.

lemma *fun-encoding*:
assumes *is-encoding e*
shows *is-encoding* $(xs \rightarrow_e e)$
⟨*proof*⟩

lemma *fun-bit-count*:
bit-count $((xs \rightarrow_e e)\ f) =$
$(\textit{if } f \in \textit{extensional}\ (\textit{set}\ xs) \textit{ then } (\sum x \leftarrow xs.\ \textit{bit-count}\ (e\ (f\ x)))$
$\textit{else}\ \infty)$
⟨*proof*⟩

lemma *fun-bit-count-est*:
assumes $f \in \textit{extensional}\ (\textit{set}\ xs)$
assumes $\bigwedge x.\ x \in \textit{set}\ xs \Longrightarrow \textit{bit-count}\ (e\ (f\ x)) \le a$

shows *bit-count* $((xs \rightarrow_e e)\ f) \le$ *ereal* (*real* (*length xs*)) $*$ *a*
⟨*proof*⟩

9 Finite Sets

definition S_e :: *'a encoding* ⇒ *'a set encoding* **where**
S_e *e S* =
(*if finite S* ∧ *S* ⊆ *dom e*
then (L_e *e* (*linorder.sorted-key-list-of-set* (≤) (*the* ∘ *e*) *S*))
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_e *e*)
⟨*proof*⟩

lemma *set-bit-count*:
assumes *is-encoding e*
shows *bit-count* (S_e *e S*) = (*if finite S then* ($\sum x \in S$. *bit-count* (*e x*)+*1*)+*1* *else* ∞)
⟨*proof*⟩

lemma *sum-triv-ereal*:
fixes *a* :: *ereal*
assumes *finite S*
shows ($\sum$ - ∈ *S*. *a*) = *card S* ∗ *a*
⟨*proof*⟩

lemma *set-bit-count-est*:
assumes *is-encoding f*
assumes *finite S*
assumes *card S* ≤ *m*
assumes *0* ≤ *a*
assumes $\bigwedge x$. *x* ∈ *S* ⟹ *bit-count* (*f x*) ≤ *a*
shows *bit-count* (S_e *f S*) ≤ *ereal* (*real m*) ∗ (*a*+*1*) + *1*
⟨*proof*⟩

10 Floating point numbers

definition F_e **where** F_e *f* = ($I_e \times_e I_e$) (*mantissa f*,*exponent f*)

lemma *float-encoding*:
is-encoding F_e
⟨*proof*⟩

lemma *suc-n-le-2-pow-n*:

fixes n :: *nat*
shows $n + 1 \le 2 \,\hat{}\, n$
⟨*proof*⟩

lemma *float-bit-count-1*:
bit-count $(F_e\ f) \le 6 + 2 * (\mathit{log}\ 2\ (|\mathit{mantissa}\ f| + 1) +$
$\mathit{log}\ 2\ (|\mathit{exponent}\ f| + 1))$ (**is** *?lhs* $\le$ *?rhs*)
⟨*proof*⟩

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* $(m * 2\ \mathit{powr}\ e)$
shows *bit-count* $(F_e\ f) \le$
$6 + 2 * (\mathit{log}\ 2\ (|m| + 2) + \mathit{log}\ 2\ (|e| + 1))$
⟨*proof*⟩

lemma *float-bit-count-zero*:
bit-count $(F_e$ (*float-of* $0)) = 2$
⟨*proof*⟩

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* $= N_e \times_e N_e$

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* $= [0..<42] \rightarrow_e Nb_e\ 314$

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

have *bit-count* (*example2 f*) $= 42*9$ (**is** *?lhs* = *?rhs*)
if a:$f \in \{0..<42\} \rightarrow_E \{0..<314\}$ **for** f
proof −
have *?lhs* $= (\sum x \leftarrow [0..<42].$ *bit-count* $(Nb_e\ 314\ (f\ x)))$
using *a* **by** (*simp add*:*example2-def fun-bit-count PiE-def*)
also have ... $= (\sum x \leftarrow [0..<42].$ *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* $= N_e \bowtie_e (\lambda n.\ [0..<42] \rightarrow_e Nb_e\ n)$

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.