

A Combinator Library for Prefix-Free Codes

Emin Karayel

May 26, 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 |
  opt-prefix - - = False
```

```
definition opt-comp x y = (opt-prefix x y  $\vee$  opt-prefix 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) |
  opt-append - - = None
```

```
lemma opt-comp-sym: opt-comp x y = opt-comp y x
by (simp add:opt-comp-def, blast)
```

```
lemma opt-comp-append:
assumes opt-comp (opt-append x y) z
shows opt-comp x z
```

```
proof -
obtain x' y' z' where a: x = Some x' y = Some y' z = Some z'
using assms
by (cases x, case-tac [!] y, case-tac [!] z, auto simp: opt-comp-def)
have prefix (x'@y') z'  $\vee$  prefix z' (x'@y')
using a assms by (simp add:opt-comp-def)
hence prefix x' z'  $\vee$  prefix z' x'
using prefix-same-cases append-prefixD by blast
thus ?thesis
using a by (simp add:opt-comp-def)
qed
```

```
lemma opt-comp-append-2:
assumes opt-comp x (opt-append y z)
```

shows *opt-comp* $x\ y$
using *opt-comp-append* *opt-comp-sym* *assms* **by** *blast*

lemma *opt-comp-append-3*:
assumes *opt-comp* (*opt-append* $x\ y$) (*opt-append* $x\ z$)
shows *opt-comp* $y\ z$
using *assms*
by (*cases* x , *case-tac*![] y , *case-tac*![] z , *auto* *simp*: *opt-comp-def*)

type-synonym $'a\ \text{encoding} = 'a \rightarrow \text{bool list}$

definition *is-encoding* :: $'a\ \text{encoding} \Rightarrow \text{bool}$
where *is-encoding* $f = (\forall x\ y. \text{opt-prefix } (f\ x)\ (f\ y) \longrightarrow x = y)$

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 Paulson [4] except for the use of partial functions for the practical reasons described in Section 1.

lemma *is-encodingI*:
assumes $\bigwedge x\ x'\ y\ y'. e\ x = \text{Some } x' \Longrightarrow e\ y = \text{Some } y' \Longrightarrow$
 $\text{prefix } x'\ y' \Longrightarrow x = y$
shows *is-encoding* e

proof –

have *opt-prefix* ($e\ x$) ($e\ y$) $\Longrightarrow x = y$ **for** $x\ y$
using *assms* **by** (*cases* $e\ x$, *case-tac*![] $e\ y$, *auto*)
thus *?thesis* **by** (*simp* *add*:*is-encoding-def*)

qed

lemma *is-encodingI-2*:
assumes $\bigwedge x\ y. \text{opt-comp } (e\ x)\ (e\ y) \Longrightarrow x = y$
shows *is-encoding* e
using *assms* **by** (*simp* *add*:*opt-comp-def* *is-encoding-def*)

lemma *encoding-triv*: *is-encoding* *Map.empty*
by (*rule* *is-encodingI-2*, *simp* *add*:*opt-comp-def*)

lemma *is-encodingD*:
assumes *is-encoding* e
assumes *opt-comp* ($e\ x$) ($e\ y$)
shows $x = y$
using *assms* **by** (*auto* *simp* *add*:*opt-comp-def* *is-encoding-def*)

lemma *encoding-imp-inj*:
assumes *is-encoding* f
shows *inj-on* f (*dom* f)
using *assms*
by (*intro* *inj-onI*, *simp* *add*:*is-encoding-def*, *force*)

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

lemma *bit-count-finite-imp-dom*:
bit-count (*f* *x*) < ∞ \implies *x* \in *dom* *f*
by (*cases* *f* *x*, *auto*)

lemma *bit-count-append*:
bit-count (*opt-append* *x* *y*) = *bit-count* *x* + *bit-count* *y*
by (*cases* *x*, *case-tac*! *y*, *simp-all*)

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 $\bigwedge x. x \in \text{dom } e1 \implies \text{is-encoding } (e2\ x)$
shows *is-encoding* (*e1* \bowtie_e *e2*)
proof (*rule is-encodingI-2*)
fix *x* *y*
assume *a*:*opt-comp* ((*e1* \bowtie_e *e2*) *x*) ((*e1* \bowtie_e *e2*) *y*)
have *d*:*opt-comp* (*e1* (*fst* *x*)) (*e1* (*fst* *y*))
using *a* **unfolding** *encode-dependent-prod-def*
by (*metis opt-comp-append opt-comp-append-2*)
hence *b*:*fst* *x* = *fst* *y*
using *is-encodingD*[*OF assms*(1)] **by** *simp*
hence *opt-comp* (*e2* (*fst* *x*) (*snd* *x*)) (*e2* (*fst* *x*) (*snd* *y*))
using *a* **unfolding** *encode-dependent-prod-def* **by** (*metis opt-comp-append-3*)
moreover **have** *fst* *x* \in *dom* *e1* **using** *d* *b*
by (*cases* *e1* (*fst* *x*), *simp-all add:opt-comp-def dom-def*)
ultimately **have** *c*:*snd* *x* = *snd* *y*
using *is-encodingD*[*OF assms*(2)] **by** *simp*
show *x* = *y*
using *b* *c* **by** (*simp add:prod-eq-iff*)
qed

lemma *dependent-bit-count*:
bit-count ((*e1* \bowtie_e *e2*) (*x1*,*x2*)) =
bit-count (*e1* *x1*) + *bit-count* (*e2* *x1* *x2*)
by (*simp add: encode-dependent-prod-def bit-count-append*)

lemma *dependent-bit-count-2*:
 $bit_count ((e_1 \times_e e_2) x) =$
 $bit_count (e_1 (fst x)) + bit_count (e_2 (fst x) (snd x))$
by (*simp add: encode-dependent-prod-def bit-count-append*)

This abbreviation is for non-dependent products.

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

4 Composition

lemma *encoding-compose*:
assumes *is-encoding f*
assumes *inj-on g {x. p x}*
shows *is-encoding ($\lambda x. \text{if } p\ x \text{ then } f\ (g\ x) \text{ else } None$)*
using *assms* **by** (*simp add: comp-def is-encoding-def inj-onD*)

lemma *encoding-compose-2*:
assumes *is-encoding f*
assumes *inj g*
shows *is-encoding ($\lambda x. f\ (g\ x)$)*
using *assms* **by** (*simp add: comp-def is-encoding-def inj-onD*)

5 Natural Numbers

fun *encode-bounded-nat* :: $nat \Rightarrow nat \Rightarrow bool \text{ list}$ **where**
 $encode_bounded_nat\ (Suc\ l)\ n =$
 $(let\ r = n \geq (2^\wedge l)\ \text{in } r \# encode_bounded_nat\ l\ (n - of_bool\ r * 2^\wedge l)) \mid$
 $encode_bounded_nat\ 0 - = []$

lemma *encode-bounded-nat-prefix-free*:
fixes $u\ v\ l :: nat$
assumes $u < 2^\wedge l$
assumes $v < 2^\wedge l$
assumes *prefix (encode-bounded-nat l u) (encode-bounded-nat l v)*
shows $u = v$
using *assms*
proof (*induction l arbitrary: u v*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc l*)
have *prefix (encode-bounded-nat l (u - of_bool (u \geq 2 $^\wedge$ l) * 2 $^\wedge$ l))*
 $(encode_bounded_nat\ l\ (v - of_bool\ (v \geq 2^\wedge l) * 2^\wedge l))$
and $a:(u \geq 2^\wedge l) = (v \geq 2^\wedge l)$

using *Suc(4)* **by** (*simp-all add: Let-def split: split-of-bool-asm*)
moreover have $u - \text{of-bool } (u \geq 2^l) * 2^l < 2^l$
using *Suc(2)* **by** (*cases u < 2^l, auto simp add:of-bool-def*)
moreover have $v - \text{of-bool } (v \geq 2^l) * 2^l < 2^l$
using *Suc(3)* **by** (*cases v < 2^l, auto simp add:of-bool-def*)
ultimately have
 $u - \text{of-bool } (u \geq 2^l) * 2^l = v - \text{of-bool } (v \geq 2^l) * 2^l$
by (*intro Suc(1), simp-all*)
thus $u = v$ **using** *a* **by** (*simp split: split-of-bool-asm*)
qed

definition $Nb_e :: \text{nat} \Rightarrow \text{nat}$ *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:*
 $\text{bit-count } (Nb_e \ l \ y) = (\text{if } y < l \text{ then } \text{floorlog } 2 \ (l-1) \ \text{else } \infty)$

proof –
have $a:\text{length } (\text{encode-bounded-nat } h \ m) = h$ **for** $h \ m$
by (*induction h arbitrary: m, simp, simp add:Let-def*)
show *?thesis*
using *a* **by** (*simp add:Nb_e-def*)
qed

lemma *bounded-nat-bit-count-2:*
assumes $y < l$
shows $\text{bit-count } (Nb_e \ l \ y) = \text{floorlog } 2 \ (l-1)$
using *assms bounded-nat-bit-count* **by** *simp*

lemma $\text{dom } (Nb_e \ l) = \{..<l\}$
by (*simp add:Nb_e-def dom-def lessThan-def*)

lemma *bounded-nat-encoding: is-encoding* ($Nb_e \ l$)
proof –
have $x < l \implies x < 2 \wedge \text{floorlog } 2 \ (l-1)$ **for** $x :: \text{nat}$
by (*intro floorlog-leD floorlog-mono, auto*)
thus *?thesis*
using *encode-bounded-nat-prefix-free*
by (*intro is-encodingI, simp add:Nb_e-def split:if-splits, blast*)
qed

fun *encode-unary-nat* $:: \text{nat} \Rightarrow \text{bool list}$ **where**
 $\text{encode-unary-nat } (\text{Suc } l) = \text{False}\#(\text{encode-unary-nat } l) \mid$
 $\text{encode-unary-nat } 0 = [\text{True}]$

```

lemma encode-unary-nat-prefix-free:
  fixes  $u\ v :: \text{nat}$ 
  assumes prefix (encode-unary-nat  $u$ ) (encode-unary-nat  $v$ )
  shows  $u = v$ 
  using assms
proof (induction  $u$  arbitrary:  $v$ )
  case  $0$ 
  then show ?case by (cases  $v$ , simp-all)
next
  case (Suc  $u$ )
  then show ?case by (cases  $v$ , simp-all)
qed

```

```

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

```

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-count ( $Nu_e\ n$ ) = Suc  $n$ 
  unfolding Nu_e-def by (induction  $n$ , auto)

```

```

lemma unary-encoding: is-encoding  $Nu_e$ 
  using encode-unary-nat-prefix-free
  by (intro is-encodingI, simp add:Nu_e-def)

```

Encoding for positive numbers using Elias-Gamma code.

```

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

```

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

```

lemma elias-gamma-bit-count:
  bit-count ( $Ng_e\ n$ ) = (if  $n > 0$  then  $2 * \lfloor \log 2\ n \rfloor + 1$  else ( $\infty :: \text{ereal}$ ))
proof (cases  $n > 0$ )
  case True
  define  $r$  where  $r = \text{floorlog}\ 2\ n - \text{Suc}\ 0$ 
  have  $\text{floorlog}\ 2\ n \neq 0$ 
  using True
  by (simp add:floorlog-eq-zero-iff)
  hence  $a:\text{floorlog}\ 2\ n > 0$  by simp

  have  $n < 2^{\wedge}(\text{floorlog}\ 2\ n)$ 

```

using *True floorlog-bounds* by *simp*
 also have $\dots = 2^{\wedge(r+1)}$
 using *a* by (*simp add:r-def*)
 finally have $n < 2^{\wedge(r+1)}$ by *simp*
 hence $b:n - 2^{\wedge r} < 2^{\wedge r}$ by *simp*
 have $\text{floorlog } 2 (2^{\wedge r} - \text{Suc } 0) \leq r$
 by (*rule floorlog-leI, auto*)
 moreover have $r \leq \text{floorlog } 2 (2^{\wedge r} - \text{Suc } 0)$
 by (*cases r, simp, auto intro: floorlog-geI*)
 ultimately have $c:\text{floorlog } 2 (2^{\wedge r} - \text{Suc } 0) = r$
 using *order-antisym* by *blast*

have $\text{bit-count } (N_{g_e} n) = \text{bit-count } (N_{u_e} r) +$
 $\text{bit-count } (N_{b_e} (2^{\wedge r}) (n - 2^{\wedge r}))$
 using *True* by (*simp add:N_{g_e}-def r-def[symmetric] dependent-bit-count*)
 also have $\dots = \text{ereal } (r + 1) + \text{ereal } (r)$
 using *b c*
 by (*simp add: unary-nat-bit-count bounded-nat-bit-count*)
 also have $\dots = 2 * r + 1$ by *simp*
 also have $\dots = 2 * \lfloor \log 2 n \rfloor + 1$
 using *True* by (*simp add:floorlog-def r-def*)
 finally show *?thesis* using *True* by *simp*

next
 case *False*
 then show *?thesis* by (*simp add:N_{g_e}-def*)
 qed

lemma *elias-gamma-encoding: is-encoding N_{g_e}*

proof –

have *a*: *inj-on* $(\lambda x. \text{let } r = \text{floorlog } 2 x - 1 \text{ in } (r, x - 2^{\wedge r}))$
 $\{n. 0 < n\}$
 proof (*rule inj-onI*)
 fix *x y* :: *nat*
 assume $x \in \{n. 0 < n\}$
 hence *x-pos*: $0 < x$ by *simp*
 assume $y \in \{n. 0 < n\}$
 hence *y-pos*: $0 < y$ by *simp*
 define *r* where $r = \text{floorlog } 2 x - \text{Suc } 0$
 assume *b*: $(\text{let } r = \text{floorlog } 2 x - 1 \text{ in } (r, x - 2^{\wedge r})) =$
 $(\text{let } r = \text{floorlog } 2 y - 1 \text{ in } (r, y - 2^{\wedge r}))$
 hence *c*: $r = \text{floorlog } 2 y - \text{Suc } 0$
 by (*simp-all add:Let-def r-def*)
 have $x - 2^{\wedge r} = y - 2^{\wedge r}$ using *b*
 by (*simp add:Let-def r-def[symmetric] c[symmetric] prod-eq-iff*)
 moreover have $x \geq 2^{\wedge r}$
 using *r-def x-pos floorlog-bounds* by *simp*
 moreover have $y \geq 2^{\wedge r}$
 using *c floorlog-bounds y-pos* by *simp*
 ultimately show $x = y$ using *eq-diff-iff* by *blast*

qed

have *is-encoding* ($\lambda n. Ng_e n$)
unfolding *Ng_e-def* **using** *a*
by (*intro encoding-compose*[**where** $f=Nu_e \times_e (\lambda r. Nb_e (2^{\widehat{r}}))$]
dependent-encoding unary-encoding bounded-nat-encoding) *auto*
thus *?thesis* **by** *simp*
qed

definition $N_e :: \text{nat encoding where } N_e x = Ng_e (x+1)$

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-encoding* N_e

proof –

have *is-encoding* ($\lambda n. N_e n$)
unfolding *N_e-def*
by (*intro encoding-compose-2*[**where** $g=(\lambda n. n + 1)$] *elias-gamma-encoding*,
auto)
thus *?thesis* **by** *simp*
qed

lemma *exp-golomb-bit-count-exact:*

bit-count ($N_e n$) = $2 * \lfloor \log 2 (n+1) \rfloor + 1$
by (*simp add:N_e-def elias-gamma-bit-count*)

lemma *exp-golomb-bit-count:*

bit-count ($N_e n$) $\leq (2 * \log 2 (\text{real } n+1) + 1)$
by (*simp add:exp-golomb-bit-count-exact add.commute*)

lemma *exp-golomb-bit-count-est:*

assumes $n \leq m$
shows *bit-count* ($N_e n$) $\leq (2 * \log 2 (\text{real } m+1) + 1)$
proof –
have *bit-count* ($N_e n$) $\leq (2 * \log 2 (\text{real } n+1) + 1)$
using *exp-golomb-bit-count* **by** *simp*
also have $\dots \leq (2 * \log 2 (\text{real } m+1) + 1)$
using *assms* **by** *simp*
finally show *?thesis* **by** *simp*
qed

6 Integers

definition $I_e :: \text{int encoding where}$

$I_e x = N_e (\text{nat } (\text{if } x \leq 0 \text{ then } (-2 * x) \text{ else } (2*x-1)))$

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 –

have *inj* ($\lambda x. \text{nat } (\text{if } x \leq 0 \text{ then } -2 * x \text{ else } 2 * x - 1)$)
by (*rule inj-onI*, *auto simp add: eq-nat-nat-iff*, *presburger*)
thus *?thesis*
unfolding $I_e\text{-def}$
by (*intro exp-golomb-encoding encoding-compose-2* [**where** $f=N_e$])
auto

qed

lemma *int-bit-count: bit-count* $(I_e n) = 2 * \lfloor \log 2 (2 * |n| + 1) \rfloor + 1$

proof –

have $a:m > 0 \implies$
 $\lfloor \log (\text{real } 2) (\text{real } (2 * m)) \rfloor = \lfloor \log (\text{real } 2) (\text{real } (2 * m + 1)) \rfloor$
for $m :: \text{nat}$ **by** (*rule floor-log-eq-if*, *auto*)
have $n > 0 \implies$
 $\lfloor \log 2 (2 * \text{real-of-int } n) \rfloor = \lfloor \log 2 (2 * \text{real-of-int } n + 1) \rfloor$
using a [**where** $m=\text{nat } n$] **by** (*simp add: add commute*)
thus *?thesis*
by (*simp add: I_e-def exp-golomb-bit-count-exact floorlog-def*)

qed

lemma *int-bit-count-1:*

assumes $\text{abs } n > 0$
shows $\text{bit-count } (I_e n) = 2 * \lfloor \log 2 |n| \rfloor + 3$

proof –

have $a:m > 0 \implies$
 $\lfloor \log (\text{real } 2) (\text{real } (2 * m)) \rfloor = \lfloor \log (\text{real } 2) (\text{real } (2 * m + 1)) \rfloor$
for $m :: \text{nat}$ **by** (*rule floor-log-eq-if*, *auto*)
have $n < 0 \implies$
 $\lfloor \log 2 (-2 * \text{real-of-int } n) \rfloor = \lfloor \log 2 (1 - 2 * \text{real-of-int } n) \rfloor$
using a [**where** $m=\text{nat } (-n)$] **by** (*simp add: add commute*)
hence $\text{bit-count } (I_e n) = 2 * \lfloor \log 2 (2 * \text{real-of-int } |n|) \rfloor + 1$
using *assms*
by (*simp add: I_e-def exp-golomb-bit-count-exact floorlog-def*)
also have $\dots = 2 * \lfloor \log 2 |n| \rfloor + 3$
using *assms* **by** (*subst log-mult*, *auto*)
finally show *?thesis* **by** *simp*

qed

lemma *int-bit-count-est-1:*

assumes $|n| \leq r$
shows $\text{bit-count } (I_e n) \leq 2 * \log 2 (r+1) + 3$

proof (*cases abs n > 0*)

case *True*

```

have real-of-int  $\lfloor \log 2 \mid \text{real-of-int } n \mid \rfloor \leq \log 2 \mid \text{real-of-int } n \mid$ 
  using of-int-floor-le by blast
also have  $\dots \leq \log 2 (\text{real-of-int } r+1)$ 
  using True assms by force
finally have
  real-of-int  $\lfloor \log 2 \mid \text{real-of-int } n \mid \rfloor \leq \log 2 (\text{real-of-int } r + 1)$ 
  by simp
then show ?thesis
  using True assms by (simp add:int-bit-count-1)
next
  case False
  have  $r \geq 0$  using assms by simp
  moreover have  $n = 0$  using False by simp
  ultimately show ?thesis by (simp add:Ie-def exp-golomb-bit-count-exact)
qed

```

```

lemma int-bit-count-est:
  assumes  $|n| \leq r$ 
  shows bit-count (Ie  $n$ )  $\leq 2 * \log 2 (2*r+1) + 1$ 
proof -
  have bit-count (Ie  $n$ )  $\leq 2 * \log 2 (2*|n|+1) + 1$ 
    by (simp add:int-bit-count)
  also have  $\dots \leq 2 * \log 2 (2* r + 1) + 1$ 
    using assms by simp
  finally show ?thesis by simp
qed

```

7 Lists

definition *Lf_e* **where**

```

Lfe  $e$   $n$   $xs =$ 
  (if length  $xs = n$ 
   then fold ( $\lambda x y. \text{opt-append } y (e\ x)$ )  $xs$  (Some  $\square$ )
   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-encoding (Lfe  $e$   $n$ )

```

proof (*induction* n)

case 0

then show *?case*

by (*rule is-encodingI-2, simp-all add:Lf_e-def opt-comp-def split:if-splits*)

next

case (*Suc* n)

show *?case*

proof (*rule is-encodingI-2*)

fix x y

```

assume  $a$ :opt-comp ( $Lf_e e (Suc n) x$ ) ( $Lf_e e (Suc n) y$ )
have  $b$ :length  $x = Suc n$  using  $a$ 
  by (cases length  $x = Suc n$ , simp-all add:Lf_e-def opt-comp-def)
then obtain  $x1 x2$  where  $x$ -def:  $x = x1@[x2]$  length  $x1 = n$ 
  by (metis length-append-singleton lessI nat.inject order.refl
    take-all take-hd-drop)
have  $c$ :length  $y = Suc n$  using  $a$ 
  by (cases length  $y = Suc n$ , simp-all add:Lf_e-def opt-comp-def)
then obtain  $y1 y2$  where  $y$ -def:  $y = y1@[y2]$  length  $y1 = n$ 
  by (metis length-append-singleton lessI nat.inject order.refl
    take-all take-hd-drop)
have  $d$ : opt-comp (opt-append ( $Lf_e e n x1$ ) ( $e x2$ ))
  (opt-append ( $Lf_e e n y1$ ) ( $e y2$ ))
  using  $a b c$  by (simp add:Lf_e-def  $x$ -def  $y$ -def)
hence opt-comp ( $Lf_e e n x1$ ) ( $Lf_e e n y1$ )
  using opt-comp-append opt-comp-append-2 by blast
hence  $e$ : $x1 = y1$ 
  using is-encodingD[OF Suc] by blast
hence opt-comp ( $e x2$ ) ( $e y2$ )
  using opt-comp-append-3  $d$  by simp
hence  $x2 = y2$ 
  using is-encodingD[OF assms] by blast
thus  $x = y$  using  $e$   $x$ -def  $y$ -def by simp
qed
qed

```

lemma *fixed-list-bit-count*:

```

bit-count ( $Lf_e e n xs$ ) =
  (if length  $xs = n$  then ( $\sum x \leftarrow xs. (bit-count (e x))$ ) else  $\infty$ )

```

proof (*induction* n *arbitrary*: xs)

case 0

then show *?case* **by** (*simp* *add:Lf_e-def*)

next

case ($Suc n$)

show *?case*

proof (*cases* *length* $xs = Suc n$)

case *True*

then obtain $x1 x2$ **where** x -*def*: $xs = x1@[x2]$ *length* $x1 = n$

by (*metis* *length-append-singleton* *lessI* *nat.inject* *order.refl*
take-all *take-hd-drop*)

have *bit-count* ($Lf_e e n x1$) = ($\sum x \leftarrow x1. bit-count (e x)$)

using x -*def*(2) *Suc* **by** *simp*

then show *?thesis* **by** (*simp* *add:Lf_e-def* x -*def* *bit-count-append*)

next

case *False*

then show *?thesis* **by** (*simp* *add:Lf_e-def*)

qed

qed

definition L_e

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

$L_e e$ 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 –

have *inj* $(\lambda xs. (length\ xs, xs))$

by (*simp add: inj-on-def*)

hence *is-encoding* $(\lambda xs. L_e e xs)$

using *assms unfolding* $L_e\text{-def}$

by (*intro encoding-compose-2* [**where** $g = (\lambda x. (length\ x, x))$]
dependent-encoding unary-encoding fixed-list-encoding) *auto*

thus *?thesis* **by** *simp*

qed

lemma *sum-list-triv-ereal*:

fixes $a :: \text{ereal}$

shows *sum-list* $(\text{map } (\lambda-. a) xs) = length\ xs * a$

apply (*cases a, simp add:sum-list-triv*)

by (*induction xs, simp, simp*) $+$

lemma *list-bit-count*:

bit-count $(L_e e xs) = (\sum x \leftarrow xs. \text{bit-count } (e\ x) + 1) + 1$

proof –

have *bit-count* $(L_e e xs) =$

ereal $(1 + \text{real } (length\ xs)) + (\sum x \leftarrow xs. \text{bit-count } (e\ x))$

by (*simp add: L_e-def dependent-bit-count fixed-list-bit-count unary-nat-bit-count*)

also have $\dots = (\sum x \leftarrow xs. \text{bit-count } (e\ x)) + (\sum x \leftarrow xs. 1) + 1$

by (*simp add:ac-simps group-cancel.add1 sum-list-triv-ereal*)

also have $\dots = (\sum x \leftarrow xs. \text{bit-count } (e\ x) + 1) + 1$

by (*simp add:sum-list-addrf*)

finally show *?thesis* **by** *simp*

qed

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* $f \in \text{extensional } (set\ xs)$

then $(Lf_e e (length\ xs) (\text{map } f\ xs))$

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 →_e e)*
proof –
have *a:inj-on (λx. map x xs) {x. x ∈ extensional (set xs)}*
by (*rule inj-onI (simp add: extensionalityI)*)
have *is-encoding (λx. (xs →_e e) x)*
unfolding *encode-fun-def*
by (*intro encoding-compose[where f=Lf_e e (length xs)]*
fixed-list-encoding assms a)
thus *?thesis* **by** *simp*
qed

lemma *fun-bit-count*:
bit-count ((xs →_e e) f) =
(if f ∈ extensional (set xs) then (∑ x ← xs. bit-count (e (f x)))
else ∞)
by (*simp add:encode-fun-def fixed-list-bit-count comp-def*)

lemma *fun-bit-count-est*:
assumes *f ∈ extensional (set xs)*
assumes $\bigwedge x. x \in \text{set } xs \implies \text{bit-count } (e (f x)) \leq a$
shows *bit-count ((xs →_e e) f) ≤ ereal (real (length xs)) * a*
proof –
have *bit-count ((xs →_e e) f) = (∑ x ← xs. bit-count (e (f x)))*
using *assms(1)* **by** (*simp add:fun-bit-count*)
also have $\dots \leq (\sum x \leftarrow xs. a)$
by (*intro sum-list-mono assms(2), simp*)
also have $\dots = \text{ereal } (\text{real } (\text{length } xs)) * a$
by (*simp add:sum-list-triv-ereal*)
finally show *?thesis* **by** *simp*
qed

9 Finite Sets

definition $S_e :: 'a \text{ encoding} \Rightarrow 'a \text{ set encoding where}$
 $S_e e S =$
(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 –
have *a:inj-on (the ∘ e) (dom e)*
using *inj-on-def*

by (*intro comp-inj-on encoding-imp-inj assms, fastforce*)

interpret *folding-insort-key* (\leq) ($<$) (*dom e*) (*the* \circ *e*)
using *a* **by** (*unfold-locales*) *auto*

have *is-encoding* ($\lambda S. S_e e S$)
unfolding *S_e-def* **using** *sorted-key-list-of-set-inject*
by (*intro encoding-compose*[**where** *f=L_e e*] *list-encoding assms*(1)
inj-onI, simp)
thus *?thesis* **by** *simp*

qed

lemma *set-bit-count*:
assumes *is-encoding e*
shows *bit-count* (*S_e e S*) = (*if finite S then* $(\sum x \in S. \text{bit-count } (e x)+1)+1$ *else* ∞)
proof (*cases finite S*)
case *f:True*
have *bit-count* (*S_e e S*) = $(\sum x \in S. \text{bit-count } (e x)+1)+1$
proof (*cases S* \subseteq *dom e*)
case *True*

have *a:inj-on* (*the* \circ *e*) (*dom e*)
using *inj-on-def* **by** (*intro comp-inj-on encoding-imp-inj*[*OF*
assms], *fastforce*)

interpret *folding-insort-key* (\leq) ($<$) (*dom e*) (*the* \circ *e*)
using *a* **by** (*unfold-locales*) *auto*

have *b:distinct* (*linorder.sorted-key-list-of-set* (\leq) (*the* \circ *e*) *S*)
(is distinct ?l) **using** *distinct-sorted-key-list-of-set True*
distinct-if-distinct-map **by** *auto*

have *bit-count* (*S_e e S*) = $(\sum x \leftarrow ?l. \text{bit-count } (e x) + 1) + 1$
using *f True* **by** (*simp add:S_e-def list-bit-count*)
also have $\dots = (\sum x \in S. \text{bit-count } (e x)+1)+1$
by (*simp add: sum-list-distinct-conv-sum-set*[*OF b*]
set-sorted-key-list-of-set[*OF True f*])
finally show *?thesis* **by** *simp*

next
case *False*
hence $\exists i \in S. e i = \text{None}$ **by** *force*
hence $\exists i \in S. \text{bit-count } (e i) = \infty$ **by** *force*
hence $(\sum x \in S. \text{bit-count } (e x) + 1) = \infty$
by (*simp add:sum-Pinf* *f*)
then show *?thesis* **using** *False* **by** (*simp add:S_e-def*)

qed
thus *?thesis* **using** *f* **by** *simp*

next
case *False*

then show *?thesis* **by** (*simp add:S_e-def*)
qed

lemma *sum-triv-ereal*:
fixes *a :: ereal*
assumes *finite S*
shows $(\sum - \in S. a) = \text{card } S * a$
proof (*cases a*)
case (*real r*)
then show *?thesis* **by** *simp*
next
case *PInf*
show *?thesis* **using** *assms PInf*
by (*induction S rule:finite-induct, auto*)
next
case *MInf*
show *?thesis* **using** *assms MInf*
by (*induction S rule:finite-induct, auto*)
qed

lemma *set-bit-count-est*:
assumes *is-encoding f*
assumes *finite S*
assumes $\text{card } S \leq m$
assumes $0 \leq a$
assumes $\bigwedge x. x \in S \implies \text{bit-count } (f x) \leq a$
shows $\text{bit-count } (S_e f S) \leq \text{ereal } (\text{real } m) * (a+1) + 1$
proof –
have $\text{bit-count } (S_e f S) = (\sum x \in S. \text{bit-count } (f x) + 1) + 1$
using *assms* **by** (*simp add:set-bit-count*)
also have $\dots \leq (\sum x \in S. a + 1) + 1$
using *assms* **by** (*intro sum-mono add-mono*) *auto*
also have $\dots = \text{ereal } (\text{real } (\text{card } S)) * (a + 1) + 1$
by (*simp add:sum-triv-ereal[OF assms(2)]*)
also have $\dots \leq \text{ereal } (\text{real } m) * (a+1) + 1$
using *assms(3,4)* **by** (*intro add-mono ereal-mult-right-mono*) *auto*
finally show *?thesis* **by** *simp*
qed

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 –
have *inj* $(\lambda x. (\text{mantissa } x, \text{exponent } x))$ (**is** *inj ?g*)
proof (*rule injI*)
fix *x y*


```

assume (mantissa x, exponent x) = (mantissa y, exponent y)
hence real-of-float x = real-of-float y
by (simp add:mantissa-exponent)
thus x = y
by (metis real-of-float-inverse)
qed
thus is-encoding (λf. F_e f)
unfolding F_e-def
by (intro encoding-compose-2[where g=?g]
    dependent-encoding int-encoding) auto
qed

```

```

lemma suc-n-le-2-pow-n:
fixes n :: nat
shows n + 1 ≤ 2 ^ n
by (induction n, simp, simp)

```

```

lemma float-bit-count-1:
  bit-count (F_e f) ≤ 6 + 2 * (log 2 (|mantissa f| + 1) +
    log 2 (|exponent f| + 1)) (is ?lhs ≤ ?rhs)
proof -
have ?lhs = bit-count (I_e (mantissa f)) +
  bit-count (I_e (exponent f))
by (simp add:F_e-def dependent-bit-count)
also have ... ≤
  ereal (2 * log 2 (real-of-int (|mantissa f| + 1)) + 3) +
  ereal (2 * log 2 (real-of-int (|exponent f| + 1)) + 3)
by (intro int-bit-count-est-1 add-mono) auto
also have ... = ?rhs
by simp
finally show ?thesis by simp
qed

```

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 ≡ float-of (m * 2 powr e)
shows bit-count (F_e f) ≤
  6 + 2 * (log 2 (|m| + 2) + log 2 (|e| + 1))
proof -
have b: (r + 1) * int i ≤ r * (2 ^ i - 1) + 1
if b-assms: r ≥ 1 for r :: int and i :: nat
proof (cases i > 0)
case True
have (r + 1) * int i = r * i + 2 * int ((i-1)+1) - i
using True by (simp add:algebra-simps)
also have ... ≤ r * i + int (2^1) * int (2^(i-1)) - i

```

```

    using b-assms
  by (intro add-mono diff-mono mult-mono of-nat-mono suc-n-le-2-pow-n)
    simp-all
  also have ... = r * i + 2i - i
    using True
    by (subst of-nat-mult[symmetric], subst power-add[symmetric])
      simp
  also have ... = r * i + 1 * (2i - int i - 1) + 1 by simp
  also have ... ≤ r * i + r * (2i - int i - 1) + 1
    using b-assms
    by (intro add-mono mult-mono, simp-all)
  also have ... = r * (2i - 1) + 1
    by (simp add:algebra-simps)
  finally show ?thesis by simp
next
case False
hence i = 0 by simp
then show ?thesis by simp
qed

have a: log 2 (|mantissa f| + 1) + log 2 (|exponent f| + 1) ≤
  log 2 (|m|+2) + log 2 (|e|+1)
proof (cases f=0)
case True then show ?thesis by simp
next
case False
moreover have f = Float m e
  by (simp add:f-def Float.abs-eq)
ultimately obtain i :: nat
  where m-def: m = mantissa f * 2i
    and e-def: e = exponent f - i
  using denormalize-shift by blast

have mantissa-ge-1: 1 ≤ |mantissa f|
  using False mantissa-noteq-0 by fastforce

have (|mantissa f| + 1) * (|exponent f| + 1) =
  (|mantissa f| + 1) * (|e+i|+1)
  by (simp add:e-def)
also have ... ≤ (|mantissa f| + 1) * ((|e|+|i|)+1)
  by (intro mult-mono add-mono, simp-all)
also have ... = (|mantissa f| + 1) * ((|e|+1)+i)
  by simp
also have ... = (|mantissa f| + 1) * (|e|+1) + (|mantissa f|+1)*i
  by (simp add:algebra-simps)
also have ... ≤ (|mantissa f| + 1) * (|e|+1) + (|mantissa f| *
  (2i-1)+1)
  by (intro add-mono b mantissa-ge-1, simp)
also have ... = (|mantissa f| + 1) * (|e|+1) + (|mantissa f| *

```

```

(2i-1+1)*(1)
  by simp
  also have
    ... ≤ (|mantissa f| + 1) * (|e|+1) + (|mantissa f| * (2i-1+1))*(|e|+1)
    by (intro add-mono mult-left-mono, simp-all)
  also have ... = (|mantissa f| + 1) + (|mantissa f| * (2i-1+1))*(|e|+1)
    by (simp add: algebra-simps)
  also have ... = (|mantissa f| * 2i+2) * (|e|+1)
    by (simp add: algebra-simps)
  also have ... = (|m|+2) * (|e|+1)
    by (simp add: m-def abs-mult)
  finally have (|mantissa f| + 1) * (|exponent f| + 1) ≤ (|m|+2) * (|e|+1)
    by simp

```

```

  hence (|real-of-int (mantissa f)| + 1) * (|of-int (exponent f)| +
1) ≤
  (|of-int m|+2) * (|of-int e|+1)
  by (simp flip: of-int-abs) (metis (mono-tags, opaque-lifting) nu-
meral-One

```

```

    of-int-add of-int-le-iff of-int-mult of-int-numeral)
  then show ?thesis by (simp add: log-mult[symmetric])
qed
have bit-count (Fe f) ≤
  6 + 2 * (log 2 (|mantissa f| + 1) + log 2 (|exponent f| + 1))
  using float-bit-count-1 by simp
also have ... ≤ 6 + 2 * (log 2 (|m| + 2) + log 2 (|e| + 1))
  using a by simp
finally show ?thesis by simp
qed

```

```

lemma float-bit-count-zero:
  bit-count (Fe (float-of 0)) = 2
  by (simp add: Fe-def dependent-bit-count int-bit-count
    zero-float.abs-eq[symmetric])

```

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 ×e Ne

```

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) ∈ dom example1
by (intro bit-count-finite-imp-dom, simp)
```

```
define example2
where example2 = [0..<42] →e Nbe 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 $\{...$

```
have bit-count (example2 f) = 42*9 (is ?lhs = ?rhs)
if a:f ∈ {0..<42} →E {0..<314} for f
proof -
have ?lhs = (∑ x←[0..<42]. bit-count (Nbe 314 (f x)))
using a by (simp add:example2-def fun-bit-count PiE-def)
also have ... = (∑ x←[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 = Ne ⋈e (λn. [0..<42] →e Nbe n)
```

The third example is more complex and illustrates the use of dependent encoders, consider a function with domain $\{.. whose values are natural numbers in the interval $\{... 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.