# 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 \(x\)
    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 \(x^{\prime} y^{\prime} z^{\prime}\) where \(a: x=\) Some \(x^{\prime} y=\) Some \(y^{\prime} z=\) Some \(z^{\prime}\)
        using assms
        by (cases \(x\), case-tac[!] y, case-tac[!] \(z\), auto simp: opt-comp-def)
    have prefix \(\left(x^{\prime} @ y^{\prime}\right) z^{\prime} \vee\) prefix \(z^{\prime}\left(x^{\prime} @ y^{\prime}\right)\)
        using a assms by (simp add:opt-comp-def)
    hence prefix \(x^{\prime} z^{\prime} \vee\) prefix \(z^{\prime} x^{\prime}\)
        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 encoding = 'a \rightharpoonup bool list
definition is-encoding :: 'a encoding }=>\mathrm{ bool
    where is-encoding f}=(\forallxy.opt-prefix (fx)(fy)\longrightarrowx=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:
```

lemma is-encodingI:
assumes $\bigwedge x x^{\prime} y y^{\prime}$. e $x=$ Some $x^{\prime} \Longrightarrow$ e $y=$ Some $y^{\prime} \Longrightarrow$
assumes $\bigwedge x x^{\prime} y y^{\prime}$. e $x=$ Some $x^{\prime} \Longrightarrow$ e $y=$ Some $y^{\prime} \Longrightarrow$
prefix $x^{\prime} y^{\prime} \Longrightarrow x=y$
prefix $x^{\prime} y^{\prime} \Longrightarrow x=y$
shows is-encoding e
shows is-encoding e
proof -
proof -
have opt-prefix (ex) (ey) $\quad \mathrm{e}=\mathrm{x}=\mathrm{for} x y$
have opt-prefix (ex) (ey) $\quad \mathrm{e}=\mathrm{x}=\mathrm{for} x y$
using assms by (cases e $x$, case-tac[!] e $y$, auto)
using assms by (cases e $x$, case-tac[!] e $y$, auto)
thus ?thesis by (simp add:is-encoding-def)
thus ?thesis by (simp add:is-encoding-def)
qed
qed
lemma is-encodingI-2:
lemma is-encodingI-2:
assumes $\bigwedge x y$. opt-comp (ex) (e y) $\Longrightarrow x=y$
assumes $\bigwedge x y$. opt-comp (ex) (e y) $\Longrightarrow x=y$
shows is-encoding e
shows is-encoding e
using assms by (simp add:opt-comp-def is-encoding-def)
using assms by (simp add:opt-comp-def is-encoding-def)
lemma encoding-triv: is-encoding Map.empty
lemma encoding-triv: is-encoding Map.empty
by (rule is-encodingI-2, simp add:opt-comp-def)
by (rule is-encodingI-2, simp add:opt-comp-def)
lemma is-encodingD:
lemma is-encodingD:
assumes is-encoding e
assumes is-encoding e
assumes opt-comp (ex) (ey)
assumes opt-comp (ex) (ey)
shows $x=y$
shows $x=y$
using assms by (auto simp add:opt-comp-def is-encoding-def)
using assms by (auto simp add:opt-comp-def is-encoding-def)
lemma encoding-imp-inj:
lemma encoding-imp-inj:
assumes is-encoding $f$
assumes is-encoding $f$
shows inj-on $f(\operatorname{dom} f)$
shows inj-on $f(\operatorname{dom} f)$
using assms
using assms
by (intro inj-onI, simp add:is-encoding-def, force)

```
    by (intro inj-onI, simp add:is-encoding-def, force)
```

```
fun bit-count :: bool list option }=>\mathrm{ ereal where
    bit-count None = \infty |
    bit-count (Some x) = ereal (length }x
lemma bit-count-finite-imp-dom:
    bit-count ( }fx)<\infty\Longrightarrowx\in\operatorname{dom}
    by (cases f }x\mathrm{ , 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 $\Rightarrow\left(' a \Rightarrow{ }^{\prime} b\right.$ encoding $) \Rightarrow\left({ }^{\prime} a \times ' b\right)$ encoding
(infixr $\bowtie_{e} 65$ )
where
encode-dependent-prod e fx= opt-append $(e(f s t x))(f(f s t x)(s n d x))$
lemma dependent-encoding:
assumes is-encoding e1
assumes $\bigwedge x . x \in \operatorname{dom}$ e1 $\Longrightarrow$ is-encoding $(e 2 x)$
shows is-encoding $\left(e 1 \bowtie_{e} e 2\right)$
proof (rule is-encodingI-2)
fix $x y$
assume a:opt-comp $\left(\left(e 1 \bowtie_{e} e 2\right) x\right)\left(\left(e 1 \bowtie_{e} e 2\right) y\right)$
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 $b: f s t x=f s t y$
using is-encoding $D[$ 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 $c: s n d x=$ snd $y$
using is-encoding $D[$ OF assms(2)] by simp
show $x=y$
using $b c$ by (simp add: prod-eq-iff)
qed
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 $\left(e_{2} x_{1} x_{2}\right)$
by (simp add: encode-dependent-prod-def bit-count-append)

## 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 $\left(e_{2}(f s t x)(\operatorname{snd} x)\right)$
by (simp add: encode-dependent-prod-def bit-count-append)
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)
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 list where
encode-bounded-nat (Suc l) $n=$
(let $r=n \geq(2 \wedge l)$ in $r \#$ encode-bounded-nat $l(n-o f-b o o l r * 2 へ l)) \mid$
encode-bounded-nat $0-=[]$
lemma encode-bounded-nat-prefix-free:
fixes $u v l::$ nat
assumes $u<2 \uparrow l$
assumes $v<$ 2^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^l)*2^l))
(encode-bounded-nat $l(v-$ of-bool $(v \geq 2 へ l) * 2 \wedge l))$
and $a:\left(u \geq\right.$ 2^ $\left.^{\wedge} l\right)=\left(v \geq\right.$ 2^ $\left.^{\wedge} l\right)$
using Suc(4) by (simp-all add: Let-def split: split-of-bool-asm)
moreover have $u-$ of-bool $(u \geq 2 \mathfrak{2}) * \mathscr{2}\urcorner l<2$ ح $l$
using $\operatorname{Suc}(2)$ by (cases $u<2^{\wedge} l$, auto simp add:of-bool-def)
moreover have $v-$ of-bool $(v \geq 2 \wedge l) * 2 へ l<2 \wedge l$
using $\operatorname{Suc}(3)$ by (cases $v<2 \wedge$ l, auto simp add:of-bool-def)
ultimately have
$u-$ of-bool $(u \geq$ 2^ $l) *$ 2^ $l=v-$ 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 $N b_{e}::$ nat $\Rightarrow$ nat encoding
where $N b_{e} l n=($
if $n<l$
then Some (encode-bounded-nat (floorlog 2 (l-1)) n) else None)
$N b_{e} l$ is encoding for natural numbers strictly smaller than $l$ using a fixed length encoding.
lemma bounded-nat-bit-count:
bit-count $\left(N b_{e} l y\right)=($ if $y<l$ then floorlog $2(l-1)$ else $\infty)$
proof -
have a:length (encode-bounded-nat $h m$ ) $=h$ for $h m$
by (induction $h$ arbitrary: $m, \operatorname{simp}, \operatorname{simp}$ add:Let-def)
show ?thesis
using $a$ by (simp add: $N b_{e}-d e f$ )
qed
lemma bounded-nat-bit-count-2:
assumes $y<l$
shows bit-count $\left(N b_{e} l y\right)=$ floorlog $2(l-1)$
using assms bounded-nat-bit-count by simp
lemma $\operatorname{dom}\left(N b_{e} l\right)=\{. .<l\}$
by (simp add: $N b_{e}$-def dom-def lessThan-def)
lemma bounded-nat-encoding: is-encoding ( $N b_{e} l$ )
proof -
have $x<l \Longrightarrow x<2^{\wedge}$ floorlog $2(l-1)$ for $x::$ nat by (intro floorlog-leD floorlog-mono, auto)
thus ?thesis
using encode-bounded-nat-prefix-free
by (intro is-encodingI, simp add:Nb $e_{e}$-def split:if-splits, blast)
qed
fun encode-unary-nat :: nat $\Rightarrow$ bool list where
encode-unary-nat (Suc l) $=$ False\#(encode-unary-nat $l) \mid$
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=
    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 Nue :: nat encoding
    where Nue 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 \(n\)
    unfolding \(N u_{e}\)-def by (induction \(n\), auto)
lemma unary-encoding: is-encoding \(N u_{e}\)
    using encode-unary-nat-prefix-free
    by (intro is-encodingI, simp add:Nu \(u_{e}\)-def)
```

Encoding for positive numbers using Elias-Gamma code.
definition $N g_{e}$ :: nat encoding where

```
    \(N g_{e} n=\)
        (if \(n>0\)
            then \(\left(N u_{e} \bowtie_{e}\left(\lambda r . N b_{e}\left({ }^{2} r\right)\right)\right)\)
            (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 encoding[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 $))$
proof (cases $n>0$ )
case True
define $r$ where $r=$ floorlog $2 n-$ Suc 0
have floorlog $2 n \neq 0$
using True
by (simp add:floorlog-eq-zero-iff)
hence a:floorlog $2 n>0$ by simp
have $n<2$ 2 (floorlog 2 $n$ )
using True floorlog-bounds by simp
also have $\ldots=2 \wedge(r+1)$
using $a$ by (simp add: $r$-def)
finally have $n<\mathcal{2 N}^{\wedge}(r+1)$ by $\operatorname{simp}$
hence $b: n-2 \uparrow r<2 \widehat{2} r$ by $\operatorname{simp}$
have floorlog 2 (2ヘr-Suc 0) $\leq r$
by (rule floorlog-leI, auto)
moreover have $r \leq$ floorlog $2\left(\right.$ ~ $\left.^{\wedge} r-S u c 0\right)$
by (cases r, simp, auto intro: floorlog-geI)
ultimately have $c$ :floorlog 2 (2 ^r - Suc 0) $=r$
using order-antisym by blast
have bit-count $\left(N g_{e} n\right)=$ bit-count $\left(N u_{e} r\right)+$ bit-count $\left(N b_{e}\left(\mathcal{Z}^{\wedge} r\right)\left(n-\mathcal{Z}^{\wedge} r\right)\right)$
using True by (simp add: $N g_{e}$-def r-def [symmetric] dependent-bit-count)
also have $\ldots=\operatorname{ereal}(r+1)+\operatorname{ereal}(r)$
using $b c$
by (simp add: unary-nat-bit-count bounded-nat-bit-count)
also have $\ldots=2 * r+1$ by simp
also have $\ldots=2 *\lfloor\log 2 n\rfloor+1$
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 $a: \operatorname{inj-on}\left(\lambda x\right.$. let $r=$ floorlog $2 x-1$ in $\left.\left(r, x-2^{\wedge} r\right)\right)$ $\{n .0<n\}$
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 $r=$ floorlog $2 x-$ Suc 0
assume $b:\left(\right.$ let $r=$ floorlog $2 x-1$ in $\left.\left(r, x-2^{\wedge} r\right)\right)=$ (let $r=$ floorlog $2 y-1 \operatorname{in}(r, y-2 \wedge r))$
hence $c: r=$ floorlog $2 y-S u c 0$
by (simp-all add:Let-def r-def)
have $x-2 \widehat{2} r=y-2 \widehat{2} r$ using $b$
by (simp add:Let-def r-def[symmetric] $c[$ symmetric] prod-eq-iff)
moreover have $x \geq 2 \widehat{ } r$
using $r$-def $x$-pos floorlog-bounds by simp
moreover have $y \geq 2 \mathcal{2} r$
using $c$ floorlog-bounds $y$-pos by simp
ultimately show $x=y$ using eq-diff-iff by blast
qed
have is-encoding ( $\lambda n . N g_{e} n$ )
unfolding $N g_{e}$-def using $a$
by (intro encoding-compose[where $f=N u_{e} \bowtie_{e}\left(\lambda r . N b_{e}(2 \widehat{ }\right.$ ) )]
dependent-encoding unary-encoding bounded-nat-encoding) auto
thus ?thesis by simp
qed
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 $N_{e}$
proof -
have is-encoding $\left(\lambda n . N_{e} n\right)$
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 $\left(N_{e} n\right)=2 *\lfloor\log 2(n+1)\rfloor+1$
by (simp add: $N_{e}$-def elias-gamma-bit-count)
lemma exp-golomb-bit-count:
bit-count $\left(N_{e} n\right) \leq(2 * \log 2($ 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 $\left(N_{e} n\right) \leq(2 * \log 2($ real $m+1)+1)$
proof -
have bit-count $\left(N_{e} n\right) \leq(2 * \log 2($ real $n+1)+1)$
using exp-golomb-bit-count by simp
also have $\ldots \leq(2 * \log 2($ real $m+1)+1)$
using assms by simp
finally show? ?thesis by simp
qed

## 6 Integers

definition $I_{e}::$ 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 \(\operatorname{inj}(\lambda x\). nat (if \(x \leq 0\) then \(-2 * x\) else \(2 * x-1)\) )
        by (rule inj-onI, auto simp add:eq-nat-nat-iff, presburger)
    thus ?thesis
        unfolding \(I_{e}\)-def
    by (intro exp-golomb-encoding encoding-compose-2[where \(\left.f=N_{e}\right]\) )
        auto
qed
lemma int-bit-count: bit-count \(\left(I_{e} n\right)=2 *\lfloor\log 2(2 *|n|+1)\rfloor+1\)
proof -
    have \(a: m>0 \Longrightarrow\)
        \(\lfloor\log (\) real 2) \((\) real \((2 * m))\rfloor=\lfloor\log (\) real 2) \((\) real \((2 * m+1))\rfloor\)
        for \(m::\) nat by (rule floor-log-eq-if, auto)
    have \(n>0 \Longrightarrow\)
        \(\lfloor\log 2(2 *\) real-of-int \(n)\rfloor=\lfloor\log 2(2 *\) real-of-int \(n+1)\rfloor\)
        using \(a[\) where \(m=n a t n]\) 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 abs \(n>0\)
    shows bit-count \(\binom{I_{e}}{n}=2 *\lfloor\log 2|n|\rfloor+3\)
proof -
    have \(a: m>0 \Longrightarrow\)
        \(\lfloor\log (\) real 2) \((\operatorname{real}(2 * m))\rfloor=\lfloor\log (\) real 2) \((\operatorname{real}(2 * m+1))\rfloor\)
        for \(m::\) nat by (rule floor-log-eq-if, auto)
    have \(n<0 \Longrightarrow\)
        \(\lfloor\log 2(-2 *\) real-of-int \(n)\rfloor=\lfloor\log 2(1-2 *\) real-of-int \(n)\rfloor\)
        using \(a[\) where \(m=n a t(-n)]\) by (simp add:add.commute)
    hence bit-count \(\left(I_{e} n\right)=2 *\lfloor\log 2(2 *\) real-of-int \(|n|)\rfloor+1\)
        using assms
        by (simp add:I \(I_{e}\)-def exp-golomb-bit-count-exact floorlog-def)
    also have \(\ldots=2 *\lfloor\log 2|n|\rfloor+3\)
        using assms by (subst log-mult, auto)
    finally show ?thesis by simp
qed
lemma int-bit-count-est-1:
    assumes \(|n| \leq r\)
    shows bit-count \(\left(I_{e} n\right) \leq 2 * \log 2(r+1)+3\)
proof (cases abs \(n>0\) )
    case True
```

```
have real-of-int \lfloorlog 2 |real-of-int n|\rfloor\leq log 2 |real-of-int n|
    using of-int-floor-le by blast
    also have ... \leqlog 2 (real-of-int r+1)
    using True assms by force
    finally have
        real-of-int \lfloorlog 2 |real-of-int n|\rfloor\leqlog 2(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\geq0 using assms by simp
    moreover have n=0 using False by simp
    ultimately show ?thesis by (simp add:I}\mp@subsup{I}{e}{}\mathrm{ -def exp-golomb-bit-count-exact)
qed
lemma int-bit-count-est:
    assumes }|n|\leq
    shows bit-count (Ien)\leq2* log 2 (2*r+1)+1
proof -
    have bit-count (I I n)\leq2* log 2 (2* n| +1) +1
    by (simp add:int-bit-count)
    also have }\ldots\leq2*\operatorname{log}2(2*r+1)+
        using assms by simp
    finally show ?thesis by simp
qed
```


## 7 Lists

```
definition }L\mp@subsup{f}{e}{}\mathrm{ where
```

    \(L f_{e}\) e \(n x s=\)
        (if length \(x s=n\)
            then fold ( \(\lambda x\) y. opt-append \(y(e x))\) xs (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 \(n\) )
proof (induction \(n\) )
    case 0
    then show? case
    by (rule is-encodingI-2, simp-all add:Lf \(e_{e}\)-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 \(\left(L f_{e} e(\right.\) Suc \(\left.n) x\right)\left(L f_{e} e(\right.\) Suc \(\left.n) y\right)\)
    have \(b\) :length \(x=\) Suc \(n\) using \(a\)
    by (cases length \(x=\) Suc n, simp-all add:Lf \(f_{e}\)-def opt-comp-def)
    then obtain \(x 1\) x2 where \(x\)-def: \(x=x 1 @[x 2]\) length \(x 1=n\)
    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 \(y=S u c n\), simp-all add:Lf \(f_{e}\)-def opt-comp-def)
    then obtain \(y 1 y 2\) where \(y\)-def: \(y=y 1 @[y 2]\) length \(y 1=n\)
    by (metis length-append-singleton lessI nat.inject order.refl
        take-all take-hd-drop)
    have d: opt-comp (opt-append (Lf \(f_{e}\) e \(n x 1\) ) (ex2))
    (opt-append (Lf e e n y1) (e y2))
    using \(a b c\) by (simp add:Lf \(e_{e}\)-def \(x\)-def \(y\)-def)
    hence opt-comp ( \(L f_{e}\) e \(n x 1\) ) ( \(L f_{e}\) e n y1)
        using opt-comp-append opt-comp-append-2 by blast
    hence \(e: x 1=y 1\)
    using is-encoding \(D[O F S u c]\) by blast
    hence opt-comp (ex2) (e yZ)
    using opt-comp-append-3 \(d\) by simp
    hence \(x 2=y^{2}\)
        using is-encoding \(D[O F\) assms \(]\) by blast
    thus \(x=y\) using e \(x\)-def \(y\)-def by simp
    qed
qed
lemma fixed-list-bit-count:
    bit-count \(\left(L f_{e}\right.\) e \(\left.n x s\right)=\)
        (if length \(x s=n\) then \(\left(\sum x \leftarrow x s .(\right.\) bit-count \(\left.(e x))\right)\) else \(\left.\infty\right)\)
proof (induction \(n\) arbitrary: xs)
    case 0
    then show? case by (simp add:Lf \(f_{e}-d e f\) )
next
    case (Suc n)
    show ?case
    proof (cases length xs \(=\) Suc \(n\) )
    case True
        then obtain \(x 1 x 2\) where \(x\)-def: xs \(=x 1 @[x 2]\) length \(x 1=n\)
            by (metis length-append-singleton lessI nat.inject order.refl
                take-all take-hd-drop)
    have bit-count \(\left(L f_{e}\right.\) e \(\left.n x 1\right)=\left(\sum x \leftarrow x 1\right.\). bit-count \(\left.(e x)\right)\)
            using \(x\)-def(2) Suc by simp
        then show ?thesis by (simp add:Lf \(e_{e}\)-def \(x\)-def bit-count-append)
    next
        case False
        then show ?thesis by (simp add:Lfe \(-d e f\) )
    qed
qed
```

```
definition }\mp@subsup{L}{e}{
    where }\mp@subsup{L}{e}{}\mathrm{ e xs = (Nu}\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 ( \(\left.L_{e} e\right)\)
proof -
    have inj \((\lambda x s\). (length \(x s, x s))\)
        by (simp add: inj-on-def)
```

    hence is-encoding ( \(\lambda x s . L_{e}\) e xs)
        using assms unfolding \(L_{e}\)-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 $a$ :: ereal
shows sum-list (map ( $\lambda$-. a) xs) $=$ length $x s * a$
apply (cases a, simp add:sum-list-triv)
by (induction xs, simp, simp)+
lemma list-bit-count:
bit-count $\left(L_{e}\right.$ e xs $)=\left(\sum x \leftarrow x s\right.$. bit-count $\left.(e x)+1\right)+1$
proof -
have bit-count $\left(L_{e}\right.$ e xs $)=$
ereal $(1+$ real $($ length $x s))+\left(\sum x \leftarrow x s\right.$. bit-count $\left.(e x)\right)$
by (simp add: $L_{e}$-def dependent-bit-count fixed-list-bit-count unary-nat-bit-count)
also have $\ldots=\left(\sum x \leftarrow x s\right.$. bit-count $\left.(e x)\right)+\left(\sum x \leftarrow x s .1\right)+1$
by (simp add:ac-simps group-cancel.add1 sum-list-triv-ereal)
also have $\ldots=\left(\sum x \leftarrow x s\right.$. bit-count $\left.(e x)+1\right)+1$
by (simp add:sum-list-addf)
finally show ?thesis by simp
qed

## 8 Functions

definition encode-fun $::$ ' $a$ list $\Rightarrow$ ' $b$ encoding $\Rightarrow\left({ }^{\prime} a \Rightarrow\right.$ ' $b$ ) encoding
(infixr $\rightarrow_{e} 65$ ) where
encode-fun xs e $f=$
(if $f \in$ extensional (set xs)
then $\left(L f_{e} e\right.$ (length xs) (map $\left.f x s\right)$ )
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 ( \(x s \rightarrow_{e} e\) )
proof -
    have a:inj-on ( \(\lambda x . \operatorname{map} x x s)\{x . x \in\) extensional (set \(x s)\}\)
        by (rule inj-onI) (simp add: extensionalityI)
    have is-encoding \(\left(\lambda x .\left(x s \rightarrow_{e} e\right) x\right)\)
        unfolding encode-fun-def
        by (intro encoding-compose[where \(f=L f_{e} e\) (length xs)]
            fixed-list-encoding assms a)
    thus ?thesis by simp
qed
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 \(\infty\) )
    by (simp add:encode-fun-def fixed-list-bit-count comp-def)
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 \((l e n g t h ~ x s)) * a\)
proof -
    have bit-count \(\left(\left(x s \rightarrow_{e} e\right) f\right)=\left(\sum x \leftarrow x s\right.\). bit-count \(\left.(e(f x))\right)\)
        using assms(1) by (simp add:fun-bit-count)
    also have \(\ldots \leq\left(\sum x \leftarrow x s . a\right)\)
        by (intro sum-list-mono assms(2), simp)
    also have \(\ldots=\operatorname{ereal}(\) real \((\) length \(x s)) * a\)
        by (simp add:sum-list-triv-ereal)
    finally show?thesis by simp
qed
```


## 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 $\left(L_{e} e\right.$ (linorder.sorted-key-list-of-set $(\leq)($ the $\left.\left.\circ e) S\right)\right)$ 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 ( $S_{e} e$ )
proof -
have a:inj-on (the $\circ e$ ) (dom e)
using inj-on-def

```
    by (intro comp-inj-on encoding-imp-inj assms, fastforce)
    interpret folding-insort-key (\leq) (<) (dom e) (the ○ e)
    using a by (unfold-locales) auto
have is-encoding ( }\lambdaS.\mp@subsup{S}{e}{e}eS
    unfolding }\mp@subsup{S}{e}{}\mathrm{ -def using sorted-key-list-of-set-inject
    by (intro encoding-compose[where f=\mp@subsup{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 S e S)= (if finite S then ( }\sumx\inS\mathrm{ . bit-count (e
x)+1)+1 else }\infty
proof (cases finite S)
    case f:True
    have bit-count ( }\mp@subsup{S}{e}{}\mathrm{ e S)}=(\sumx\inS.bit-count (ex)+1)+
    proof (cases S\subseteqdome)
        case True
    have a:inj-on (the o 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 ○ e)S)
        (is distinct?l) using distinct-sorted-key-list-of-set True
                distinct-if-distinct-map by auto
    have bit-count (S}\mp@subsup{S}{e}{}\mathrm{ e S)}=(\sumx\leftarrow?l. bit-count (ex)+1)+
        using}f\mathrm{ True by (simp add:S S S-def list-bit-count)
    also have ... = (\sumx\inS. bit-count (ex)+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 }\existsi\inS. e i=None by forc
        hence }\existsi\inS\mathrm{ . bit-count (e i)= 支 by force
        hence (\sumx\inS.bit-count (ex)+1)=\infty
            by (simp add:sum-Pinfty f)
            then show ?thesis using False by (simp add:S S -def)
    qed
    thus ?thesis using f by simp
next
    case False
```

```
    then show ?thesis by (simp add:S S - def)
qed
lemma sum-triv-ereal:
    fixes a :: ereal
    assumes finite S
    shows (\sum-\inS.a)=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 card S\leqm
    assumes 0\leqa
    assumes }\x.x\inS\Longrightarrowbit-count (fx)\leq
    shows bit-count (S S f S) \leqereal (real m) * (a+1) +1
proof -
    have bit-count (Se S S)=(\sumx\inS. bit-count (fx)+1)+1
    using assms by (simp add:set-bit-count)
    also have ... }\leq(\sumx\inS.a+1)+
    using assms by (intro sum-mono add-mono) auto
    also have ... = ereal (real (card S))*(a+1) +1
    by (simp add:sum-triv-ereal[OF assms(2)])
    also have ...\leqereal (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 $F_{e} f=\left(I_{e} \times_{e} I_{e}\right)($ mantissa $f$, exponent $f)$
lemma float-encoding:
is-encoding $F_{e}$
proof -
have inj $(\lambda x$. (mantissa $x$, 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=
        by (metis real-of-float-inverse)
    qed
    thus is-encoding ( }\lambdaf.\mp@subsup{F}{e}{}f
    unfolding }\mp@subsup{F}{e}{}-de
    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\leq2` n
    by (induction n, simp, simp)
lemma float-bit-count-1:
    bit-count (Fef)\leq6+2*(log 2(|mantissa f | + 1) +
        log 2 (|exponent f| + 1)) (is ?lhs \leq?rhs)
proof -
    have ?lhs = bit-count (I I (mantissa f)) +
        bit-count (I (exponent f))
        by (simp add:Fe-def dependent-bit-count)
    also have ... \leq
        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 \(f \equiv\) float-of ( \(m * 2\) powr e)
    shows bit-count \(\left(F_{e} f\right) \leq\)
        \(6+2 *(\log 2(|m|+2)+\log 2(|e|+1))\)
proof -
    have \(b:(r+1) *\) int \(\left.i \leq r *()^{\text {^ }} i-1\right)+1\)
        if \(b\)-assms: \(r \geq 1\) for \(r::\) int and \(i::\) nat
    proof (cases \(i>0\) )
        case True
        have \((r+1) * \operatorname{int} i=r * i+2 * \operatorname{int}((i-1)+1)-i\)
        using True by (simp add:algebra-simps)
        also have \(\ldots \leq r * i+\operatorname{int}(2 ` 1) * \operatorname{int}(2 `(i-1))-i\)
```

using $b$-assms
by (intro add-mono diff-mono mult-mono of-nat-mono suc-n-le-2-pow-n) simp-all
also have $\ldots=r * i+2 \uparrow i-i$
using True
by (subst of-nat-mult $[$ symmetric $]$, subst power-add [symmetric]) simp
also have $\ldots=r * i+1 *\left(\right.$ 2 $\left.^{\wedge} i-\operatorname{int} i-1\right)+1$ by simp
also have $\ldots \leq r * i+r *\left(2^{\wedge} i-\right.$ int $\left.i-1\right)+1$
using $b$-assms
by (intro add-mono mult-mono, simp-all)
also have $\ldots=r *\left(2^{\wedge} i-1\right)+1$
by (simp add:algebra-simps)
finally show ?thesis by simp
next
case False
hence $i=0$ by $\operatorname{simp}$
then show ?thesis by simp
qed
have a: log 2 $(\mid$ mantissa $f \mid+1)+\log 2(\mid$ exponent $f \mid+1) \leq$
$\log 2(|m|+2)+\log 2(|e|+1)$
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 $f * 2{ }^{\wedge} i$
and $e$-def: $e=$ exponent $f-i$
using denormalize-shift by blast
have mantissa-ge-1: $1 \leq \mid$ mantissa $f \mid$
using False mantissa-noteq-0 by fastforce
have $(\mid$ mantissa $f \mid+1) *(\mid$ exponent $f \mid+1)=$
$(\mid$ mantissa $f \mid+1) *(|e+i|+1)$
by ( $\operatorname{simp}$ add:e-def)
also have $\ldots \leq(\mid$ mantissa $f \mid+1) *((|e|+|i|)+1)$
by (intro mult-mono add-mono, simp-all)
also have $\ldots=(\mid$ mantissa $f \mid+1) *((|e|+1)+i)$
by $\operatorname{simp}$
also have $\ldots=(\mid$ mantissa $f \mid+1) *(|e|+1)+(\mid$ mantissa $f \mid+1) * i$
by (simp add:algebra-simps)
also have $\ldots \leq(\mid$ mantissa $f \mid+1) *(|e|+1)+(\mid$ mantissa $f \mid *$
$(2 \sim i-1)+1)$
by (intro add-mono b mantissa-ge-1, simp)
also have $\ldots=(\mid$ mantissa $f \mid+1) *(|e|+1)+(\mid$ mantissa $f \mid *$

```
(2`i-1)+1)*(1)
        by simp
    also have
        ...\leq(|mantissaf | + 1)*(|e|+1) +( mantissa f |* (\mathscr{2}i-1)+1)*(|e|+1)
        by (intro add-mono mult-left-mono, simp-all)
    also have }\ldots=((|\mathrm{ mantissa f | +1)+(|mantissa f |* (2`i-1)+1))*(|e|+1)
        by (simp add:algebra-simps)
```



```
        by (simp add:algebra-simps)
    also have ... = (|m|+2)*(|e|+1)
        by (simp add:m-def abs-mult)
    finally have }(|\mathrm{ mantissa }f|+1)*(|\mathrm{ exponent f | + 1) }\leq(|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 (Fef)\leq
    6+2*(log2 (|mantissa f | + 1) + log 2 ( |exponent f | + 1))
    using float-bit-count-1 by simp
    also have .. \leq6 +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:F}\mp@subsup{e}{e}{-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}\times\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 example 1
    by (intro bit-count-finite-imp-dom, simp)
define example2
    where example2 \(=[0 . .<42] \rightarrow_{e} N b_{e} 314\)
```

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 \(=\) ? rhs \()\)
    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 \(\ldots=\) ? 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

## 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.

