Gray Codes for Arbitrary Numeral Systems

Maximilian Spitz

April 18, 2024

Abstract

The original Gray code after Frank Gray, also known as reflected binary code (RBC), is an ordering of the binary numeral system such that two successive values differ only in one bit. We provide a theory for Gray codes of arbitrary numeral systems, which is a generalisation of the original idea to an arbitrary base as presented by Sankar et al. [1]. Contained is the necessary theoretical environment to express and reason about the respective properties.

Contents

1 An Encoding for Natural Numbers

theory *Encoding-Nat*
imports *Main*
begin

At first, an encoding of naturals as lists of digits with respect to an arbitrary base $b \geq 2$ is introduced because the presented Gray code and its properties are reasonably expressed in terms of a word representation of numbers.

1.1 Validity and Valuation

In the context of a given base, not all possible code words are valid number representations. A validity predicate is defined, that checks if a code word is valid and a valuation to obtain the number represented by a valid word.

type-synonym *base* = *nat*

type-synonym *word* = *nat list*

fun *val* :: *base* ⇒ *word* ⇒ *nat* **where**
  val b [] = *0*
| *val b* (*a*#*w*) = *a* + *b***val b w*

fun *valid* :: *base* ⇒ *word* ⇒ *bool* **where**
  valid b [] ⟷ *2*≤*b*
| *valid b* (*a*#*w*) ⟷ *a*<*b* ∧ *valid b w*

Given a base, the value of a valid word is bound by its length.

lemma *val-bound*:
  valid b w ⟹ *val b w* < *b*^*length*(*w*)
proof (*induction w*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a w*)
  hence *IH*: *1*+*val b w* ≤ *b*^*length*(*w*) **by** *simp*
  have *val b* (*a*#*w*) < *b**(*1*+*val b w*) **using** *Cons.prems* **by** *auto*
  also have ... ≤ *b***b*^*length*(*w*) **using** *IH mult-le-mono2* **by** *blast*
  also have ... = *b*^*length*(*a*#*w*) **by** *simp*
  finally show *?case* **by** *blast*
qed

lemma *valid-base*:
  valid b w ⟹ *2*≤*b*
  by (*induction w*) *auto*

1.2 Encoding Numbers as Words

It was stated that not all code words are valid. Similarly, numbers do not have a unique word representation in general. Therefore, it is reasonable to normalise representations with respect to either value or word length. A normal representation w.r.t. value is without leading zeroes. However, if the word length is fixed, numbers can be represented only up to an upper bound. Note that this bound is stated above.

fun *enc* :: *base* ⇒ *nat* ⇒ *word* **where**
  enc - *0* = []
| *enc b n* = (*if 2*≤*b then n mod b*#*enc b* (*n div b*) *else undefined*)

fun *enc-len* :: *base* ⇒ *nat* ⇒ *nat* **where**
enc-len - 0 = 0
| *enc-len b n = (if 2≤b then Suc(enc-len b (n div b)) else undefined)*

fun *lenc* :: *nat* ⇒ *base* ⇒ *nat* ⇒ *word* **where**
lenc 0 - - = []
| *lenc (Suc k) b n = n mod b#lenc k b (n div b)*

definition *normal* :: *base* ⇒ *word* ⇒ *bool* **where**
normal b w ≡ enc-len b (val b w) = length w

1.3 Correctness

Now, the expected properties of above definitions are proven as well as that they interact correctly.

lemma *length-enc*:
2≤b ⟹ length (enc b n) = enc-len b n
by (*induction b n rule*: *enc-len.induct*) *auto*

lemma *length-lenc*:
length (lenc k b n) = k
by (*induction k arbitrary*: *n*) *auto*

lemma *val-correct*:
valid b w ⟹ lenc (length w) b (val b w) = w
by (*induction w*) *auto*

lemma *val-enc*:
2≤b ⟹ val b (enc b n) = n
by (*induction b n rule*: *enc.induct*) *auto*

lemma *val-lenc*:
val b (lenc k b n) = n mod b^k
apply (*induction k arbitrary*: *n*)
by (*auto simp add*: *mod-mult2-eq*)

lemma *valid-enc*:
2≤b ⟹ valid b (enc b n)
by (*induction b n rule*: *enc.induct*) *auto*

lemma *valid-lenc*:
2≤b ⟹ valid b (lenc k b n)
by (*induction k arbitrary*: *n*) *auto*

lemma *encodings-agree*:
2≤b ⟹ lenc (enc-len b n) b n = enc b n
by (*metis length-enc val-correct val-enc valid-enc*)

lemma *inj-enc*:

```
  2≤b ⟹ inj (enc b)
  by (metis val-enc injI)

lemma inj-lenc:
  inj-on (lenc k b) {..<b^k}
proof (rule inj-on-inverseI)
  fix n :: nat
  assume n ∈ {..<b^k}
  thus val b (lenc k b n) = n by (simp add: val-lenc)
qed

lemma normal-enc:
  2≤b ⟹ normal b (enc b n)
  by (simp add: length-enc normal-def val-enc)

lemma normal-eq:
  ⟦valid b v; valid b w; normal b v; normal b w; val b v = val b w⟧ ⟹ v = w
  by (metis normal-def val-correct)

lemma inj-val:
  inj-on (val b) {w. valid b w ∧ normal b w}
proof (rule inj-onI)
  fix u v :: word
  assume 1: val b u = val b v
  assume u ∈ {w. valid b w ∧ normal b w}
    and v ∈ {w. valid b w ∧ normal b w}
  hence valid b u ∧ normal b u ∧ valid b v ∧ normal b v by blast
  with 1 show u = v using normal-eq by blast
qed

lemma enc-val:
  ⟦valid b w; normal b w⟧ ⟹ enc b (val b w) = w
  by (metis encodings-agree normal-def val-correct valid-base)

lemma range-enc:
  2≤b ⟹ range (enc b) = {w. valid b w ∧ normal b w}
proof
  show 2≤b ⟹ range (enc b) ⊆ {w. valid b w ∧ normal b w}
    by (simp add: image-subsetI normal-enc valid-enc)
next
  assume 2≤b
  show {w. valid b w ∧ normal b w} ⊆ range (enc b)
  proof
    fix v :: word
    assume v ∈ {w. valid b w ∧ normal b w}
    hence valid b v ∧ normal b v by blast
    hence enc b (val b v) = v by (simp add: enc-val)
    thus v ∈ range (enc b) by (metis rangeI)
  qed
```

qed

lemma *range-lenc*:
  $2 \le b \Longrightarrow$ *lenc k b* ` $\{..<b$ ^ $k\} = \{w.$ *valid b w* $\wedge$ *length w* $= k\}$
proof
  show $2 \le b \Longrightarrow$ *lenc k b* ` $\{..<b$ ^ $k\} \subseteq \{w.$ *valid b w* $\wedge$ *length w* $= k\}$
    by (*simp add*: *image-subsetI length-lenc valid-lenc*)
next
  assume $2 \le b$
  show $\{w.$ *valid b w* $\wedge$ *length w* $= k\} \subseteq$ *lenc k b* ` $\{..<b$ ^ $k\}$
  proof
    fix *v* :: *word*
    let *?v* = *val b v*
    assume $v \in \{w.$ *valid b w* $\wedge$ *length w* $= k\}$
    hence *1*: *valid b v* $\wedge$ *length v* $= k$ **by** *blast*
    hence *?v* $<$ *b^k* **using** *val-bound* **by** *blast*
    hence *?v* $\in \{..<b\hat{}k\}$ **by** *blast*
    from *1* **have** *lenc k b ?v* $= v$ **using** *val-correct* **by** *blast*
    thus $v \in$ *lenc k b* ` $\{..<b$ ^ $k\}$ **by** (*metis* ‹*?v* $\in \{..<b\hat{}k\}$› *image-eqI*)
  qed
qed

theorem *enc-correct*:
  $2 \le b \Longrightarrow$ *bij-betw* (*enc b*) *UNIV* $\{w.$ *valid b w* $\wedge$ *normal b w*$\}$
  by (*simp add*: *bij-betw-def inj-enc range-enc*)

Given a valid base b and length k, we encode exactly the first b^k numbers.

theorem *lenc-correct*:
  $2 \le b \Longrightarrow$ *bij-betw* (*lenc k b*) $\{..<b\hat{}k\}$ $\{w.$ *valid b w* $\wedge$ *length w* $= k\}$
  by (*simp add*: *bij-betw-def inj-lenc range-lenc*)

1.4 Circular Increment Operation

It is beneficial for our purpose to have an increment operation on words of fixed length that wraps around. Mathematically, this corresponds to adding 1 in the additive group of the factor ring of the integers modulo (b^k). Correctness is proven in terms of previously verified operations.

fun *inc* :: *nat* ⇒ *word* ⇒ *word* **where**
  inc - [] = []
| *inc b* (*a#w*) = *Suc a mod b#*(*if Suc a* ≠ *b then w else inc b w*)

lemma *length-inc*:
  length (*inc b w*) = *length w*
  by (*induction w*) *auto*

lemma *valid-inc*:
  valid b w ⟹ *valid b* (*inc b w*)
  by (*induction w*) *auto*

Note that the following fact shows that we do not only have an encoding in the sense that it is a bijection but we also preserve a certain structure, that is necessary for the purpose of reasoning about Gray codes.

```
theorem val-inc:
  valid b w ⟹ val b (inc b w) = Suc (val b w) mod b^length(w)
proof (induction w)
  case Nil thus ?case by simp
next
  case (Cons a w)
  hence IH: val b (inc b w) = Suc(val b w) mod b^length(w) by simp
  show ?case
  proof cases
    assume 1: Suc a = b
    hence val b (inc b (a#w)) = b∗val b (inc b w) by simp
    also have ... = b∗(Suc(val b w) mod b^length w) using IH by simp
     also have ... = b∗Suc(val b w) mod (b∗b^length w) using mult-mod-right by
blast
    also have ... = (Suc a + b∗val b w) mod (b^length(a#w)) by (simp add: 1)
    also have ... = Suc(val b (a # w)) mod (b^length(a#w)) by simp
    finally show ?thesis by blast
  next
    let ?v = Suc a + b∗val b w
    assume 2: Suc a ≠ b
    with Cons.prems have valid b (inc b (a#w)) by simp
    hence val b (inc b (a#w)) < b^length(inc b (a#w)) using val-bound by blast
    hence val b (inc b (a#w)) < b^length(a#w) using length-inc by metis
    hence ?v < b^length(a#w) using 2 Cons.prems by simp
    hence ?v = ?v mod b^length(a#w) by simp
    thus ?thesis using 2 Cons.prems by auto
  qed
qed

lemma inc-correct:
  inc b (lenc k b n) = lenc k b (Suc n)
  apply (induction k arbitrary: n)
  by (auto simp add: div-Suc mod-Suc)

lemma inc-not-eq: valid b w ⟹ (inc b w = w) = (w = [])
  by (induction w) auto

end
```

2 A Generalised Distance Measure

```
theory Code-Word-Dist
  imports Encoding-Nat
begin
```

In the case of the reflected binary code (RBC) it is sufficient to use the Hamming distance to express the property, because there are only two distinct digits so that one bitflip naturally always corresponds to a distance of 1.

2.1 Distance of Digits

We can interpret a bitflip as an increment modulo 2, which is why for the distance of digits it appears as a natural generalisation to choose the amount of required increments. Mathematically, the distance $d(x, y)$ should be $y - x \pmod b$. For example we have $d(0,1) = d(1,0) = 1$ in the binary numeral system.

definition *dist1* :: *base* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
  dist1 b x y ≡ *if x≤y then y−x else b+y−x*

Note that the distance of digits is in general asymmetric, so that it is in paticular not a metric. However, this is not an issue and in fact the most appropriate generalisation, partly due to the next lemma:

lemma *dist1-eq*:
  ⟦*x* < *b*; *y* < *b*; *dist1 b x y* = *0*⟧ ⟹ *x* = *y*
  by (*auto simp add*: *dist1-def split*: *if-splits*)

lemma *dist1-0*:
  dist1 b x x = *0*
  by (*auto simp add*: *dist1-def*)

lemma *dist1-ge1*:
  ⟦*x* < *b*; *y* < *b*; *x*≠*y*⟧ ⟹ *dist1 b x y* ≥ *1*
  using *dist1-eq* **by** *fastforce*

lemma *dist1-elim-1*:
  ⟦*x* < *b*; *y* < *b*⟧ ⟹ (*dist1 b x y* + *x*) *mod b* = *y*
  by (*auto simp add*: *dist1-def*)

lemma *dist1-elim-2*:
  ⟦*x* < *b*; *y* < *b*⟧ ⟹ *dist1 b x* (*x*+*y*) = *y*
  by (*auto simp add*: *dist1-def*)

lemma *dist1-mod-Suc*:
  ⟦*x* < *b*; *y* < *b*⟧ ⟹ *dist1 b x* (*Suc y mod b*) = *Suc* (*dist1 b x y*) *mod b*
  by (*auto simp add*: *dist1-def mod-Suc*)

lemma *dist1-Suc*:
  ⟦*2* ≤ *b*; *x* < *b*⟧ ⟹ *dist1 b x* (*Suc x mod b*) = *1*
  by (*simp add*: *dist1-0 dist1-mod-Suc*)

lemma *dist1-asym*:
  ⟦*x* < *b*; *y* < *b*⟧ ⟹ (*dist1 b x y* + *dist1 b y x*) *mod b* = *0*

by (*auto simp add*: *dist1-def*)

lemma *dist1-valid*:
$[\![x < b;\ y < b]\!] \Longrightarrow dist1\ b\ x\ y < b$
by (*auto simp add*: *dist1-def*)

lemma *dist1-distr*:
$[\![x < b;\ y < b;\ z < b]\!] \Longrightarrow dist1\ b\ (dist1\ b\ x\ y)\ (dist1\ b\ x\ z) = dist1\ b\ y\ z$
by (*auto simp add*: *dist1-def*)

lemma *dist1-distr2*:
$[\![x < b;\ y < b;\ z < b]\!] \Longrightarrow dist1\ b\ (dist1\ b\ x\ z)\ (dist1\ b\ y\ z) = dist1\ b\ y\ x$
by (*auto simp add*: *dist1-def*)

2.2 (Hamming-) Distance between Words

The total distance between two words of equal length is then defined as the sum of component-wise distances. Note that the Hamming distance is equivalent to this definition for $b = 2$ and is in general a lower bound.

fun *hamming* :: *word* ⇒ *word* ⇒ *nat* **where**
hamming [] [] = *0*
| *hamming* (*a*#*v*) (*b*#*w*) = (*if a*≠*b then 1 else 0*) + *hamming v w*

The Hamming distance is only defined in the case of equal word length. In the following definition of a distance we assume leading zeroes if the word length is not equal:

fun *dist* :: *base* ⇒ *word* ⇒ *word* ⇒ *nat* **where**
dist - [] [] = *0*
| *dist b* (*x*#*xs*) [] = *dist1 b x 0* + *dist b xs* []
| *dist b* [] (*y*#*ys*) = *dist1 b 0 y* + *dist b* [] *ys*
| *dist b* (*x*#*xs*) (*y*#*ys*) = *dist1 b x y* + *dist b xs ys*

lemma *dist-0*:
dist b w w = *0*
apply (*induction w*)
by (*auto simp add*: *dist1-0*)

lemma *dist-eq*:
$[\![$*valid b v*; *valid b w*; *length v*=*length w*; *dist b v w* = *0*$]\!] \Longrightarrow$ *v* = *w*
apply (*induction b v w rule*: *dist.induct*)
by (*auto simp add*: *dist1-eq*)

lemma *dist-posd*:
$[\![$*valid b v*; *valid b w*; *length v*=*length w*$]\!] \Longrightarrow$ (*dist b v w* = *0*) = (*v* = *w*)
using *dist-0 dist-eq* **by** *auto*

lemma *hamming-posd*:

$length\ v{=}length\ w \Longrightarrow (hamming\ v\ w = 0) = (v = w)$
by (*induction v w rule*: *hamming.induct*) *auto*

lemma *hamming-symm*:
$length\ v{=}length\ w \Longrightarrow hamming\ v\ w = hamming\ w\ v$
by (*induction v w rule*: *hamming.induct*) *auto*

theorem *hamming-dist*:
$[\![valid\ b\ v;\ valid\ b\ w;\ length\ v{=}length\ w]\!] \Longrightarrow hamming\ v\ w \leq dist\ b\ v\ w$
apply (*induction b v w rule*: *dist.induct*)
apply *auto*
using *dist1-ge1* **by** *fastforce*

end

3 A non-Boolean Gray code

theory *Non-Boolean-Gray*
imports *Code-Word-Dist*
begin

The function presented below transforms a code word into a gray code and the corresponding decode function is exactly its inverse. The key idea is to shift down a digit by the prefix sum of gray digits. A crucial property is the behavior of this prefix sum under increment as stated below.

fun *to-gray* :: $base \Rightarrow word \Rightarrow word$ **where**
to-gray - [] = []
| *to-gray b* (*a#v*) = (*let g=to-gray b v in dist1 b* (*sum-list g mod b*) *a#g*)

fun *decode* :: $base \Rightarrow word \Rightarrow word$ **where**
decode - [] = []
| *decode b* (*g#c*) = (*g+sum-list c mod b*) *mod b#decode b c*

3.1 The Correctness Proof

The proof of all properties that are necessary for a gray code is presented below. Also, some auxiliary lemmas are required:

lemma *length-gray*:
length (*to-gray b w*) = *length w*
apply (*induction w*)
by (*auto simp add*: *Let-def*)

lemma *valid-gray*:
$valid\ b\ w \Longrightarrow valid\ b\ (to\text{-}gray\ b\ w)$
apply (*induction w*)
by (*auto simp add*: *dist1-valid Let-def*)

The sum of grays is congruent to the value (mod b):

```
lemma prefix-sum:
  valid b w ⟹ sum-list (to-gray b w) mod b = val b w mod b
proof (induction w)
  case Nil thus ?case by simp
next
  case (Cons a w)
  hence IH: sum-list (to-gray b w) mod b = val b w mod b by simp
  let ?s = sum-list (to-gray b w)
  let ?v = val b w mod b
  have (dist1 b ?v a + ?s) mod b = (dist1 b ?v a + ?s mod b) mod b by presburger
  also have ... = (dist1 b ?v a + ?v) mod b using IH by argo
  also have ... = a using Cons.prems dist1-elim-1 by simp
  finally show ?case using Cons by auto
qed

lemma decode-correct:
  valid b w ⟹ decode b (to-gray b w) = w
  apply (induction w)
  by (auto simp add: Let-def dist1-elim-1)
```

The following theorem states that the transformation to gray is an encoding of the valid code words:

```
theorem gray-encoding:
  inj-on (to-gray b) {w. valid b w}
proof (rule inj-on-inverseI)
  fix w :: word
  assume w ∈ {w. valid b w}
  hence valid b w by blast
  thus decode b (to-gray b w) = w using decode-correct by simp
qed

lemma mod-mod-aux: 1 ≤ k ⟹ (a::nat) mod b^k mod b = a mod b
  by (simp add: mod-mod-cancel)

lemma gray-dist:
  valid b w ⟹ dist b (to-gray b w) (to-gray b (inc b w)) ≤ 1
proof (induction w)
  case Nil thus ?case by simp
next
  case (Cons a w)
  have valid b w using Cons.prems by simp
  hence 2 ≤ b using valid-base by auto
  hence 0 < b by simp
  have IH: dist b (to-gray b w) (to-gray b (inc b w)) ≤ 1
    using ‹valid b w› Cons.IH by blast
  have a < b using Cons.prems by simp
  show ?case
  proof (cases w)
    case Nil thus ?thesis
```

```
    using dist1-distr dist1-Suc ‹a < b› ‹2 ≤ b› by simp
next
  case (Cons a' ds')
  hence 1≤length(w) by simp
  let ?a = if Suc a≠b then w else inc b w
  let ?g = sum-list (to-gray b w) mod b
  let ?h = sum-list (to-gray b ?a) mod b
  let ?v = val b w mod b
  let ?u = val b ?a mod b
  let ?l = dist b (to-gray b (a#w)) (to-gray b (inc b (a#w)))
  have valid b ?a using ‹valid b w› valid-inc by simp
  have ?l = dist1 b (dist1 b ?g a) (dist1 b ?h (Suc a mod b))
          + dist b (to-gray b w) (to-gray b ?a)
    by (metis Encoding-Nat.inc.simps(2) dist.simps(4) to-gray.simps(2))
  also have ... = Suc (dist1 b (dist1 b ?g a) (dist1 b ?h a)) mod b
          + dist b (to-gray b w) (to-gray b ?a)
    using ‹a < b› dist1-mod-Suc dist1-valid by simp
  also have ... = Suc (dist1 b ?h ?g) mod b
          + dist b (to-gray b w) (to-gray b ?a)
    using ‹a < b› dist1-distr2 by simp
  also have ... = Suc (dist1 b ?h ?v) mod b
          + dist b (to-gray b w) (to-gray b ?a)
    using ‹valid b w› prefix-sum by simp
  also have ... = Suc (dist1 b ?u ?v) mod b
          + dist b (to-gray b w) (to-gray b ?a)
    using ‹valid b ?a› prefix-sum by simp
  also have ... = (
    if Suc a ≠ b then Suc 0 mod b
    else Suc (dist1 b (val b (inc b w) mod b) ?v) mod b
         + dist b (to-gray b w) (to-gray b (inc b w)))
    using dist-0 dist1-0 by simp
  also have ... = (
    if Suc a ≠ b then Suc 0 mod b
    else Suc (dist1 b (Suc (val b w) mod b^length(w) mod b) ?v) mod b
         + dist b (to-gray b w) (to-gray b (inc b w)))
    using ‹valid b w› valid-inc val-inc by simp
  also have ... = (
    if Suc a ≠ b then Suc 0 mod b
    else Suc (dist1 b (Suc (val b w) mod b) ?v) mod b
         + dist b (to-gray b w) (to-gray b (inc b w)))
    using ‹1≤length(w)› mod-mod-aux by simp
  also have ... = (
    if Suc a ≠ b then Suc 0 mod b
    else dist1 b (Suc (val b w) mod b) (Suc ?v mod b)
         + dist b (to-gray b w) (to-gray b (inc b w)))
    using dist1-mod-Suc by auto
  also have ... = (
    if Suc a ≠ b then Suc 0 mod b
    else dist1 b (Suc ?v mod b) (Suc ?v mod b)
```

```
        + dist b (to-gray b w) (to-gray b (inc b w)))
      using mod-Suc-eq by presburger
    also have ... = (
        if Suc a ≠ b then Suc 0 mod b
        else dist b (to-gray b w) (to-gray b (inc b w)))
      using dist1-0 by simp
    also have ... ≤ 1 using IH by simp
    finally show ?thesis by blast
  qed
qed

lemmas gray-simps = decode-correct dist-posd inc-not-eq length-gray length-inc
valid-gray valid-inc

lemma gray-empty:
  valid b w ⟹ (dist b (to-gray b w) (to-gray b (inc b w)) = 0) = (w = [])
  by (metis gray-simps)
```

The central theorem states, that it requires exactly one increment operation of one place within the word to go from the gray encoding of a number to the gray encoding of its successor. Note also, that we obtain a cyclic gray code in all cases, because the increment operation wraps the last number around to zero. Only the pathological case of an empty word has to be excluded.

```
theorem gray-correct:
  ⟦valid b w; w ≠ []⟧ ⟹ dist b (to-gray b w) (to-gray b (inc b w)) = 1
proof (rule ccontr)
  assume a: dist b (to-gray b w) (to-gray b (inc b w)) ≠ 1
  assume valid b w and w ≠ []
  hence dist b (to-gray b w) (to-gray b (inc b w)) ≠ 0 using gray-empty by blast
  with a have dist b (to-gray b w) (to-gray b (inc b w)) > 1 by simp
  thus False using ‹valid b w› gray-dist by fastforce
qed

lemmas hamming-simps = gray-dist hamming-dist le-trans length-gray length-inc
valid-gray valid-inc

theorem gray-hamming: valid b w ⟹ hamming (to-gray b w) (to-gray b (inc b
w)) ≤ 1
  by (metis hamming-simps)

end
```

References

[1] K. Sankar, V. Pandharipande, and P. Moharir. Generalized gray codes. In *Proceedings of 2004 International Symposium on Intelligent Signal*

Processing and Communication Systems. ISPACS 2004., pages 654–659, 2004. https://doi.org/10.1109/ISPACS.2004.1439140.