Coding with Asymmetric Numeral Systems
(long version)

Jeremy Gibbons

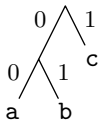
University of Oxford

Abstract. Asymmetric Numeral Systems (ANS) are an entropy-based encoding method introduced by Jarek Duda, combining the Shannon-optimal compression *effectiveness* of arithmetic coding with the execution *efficiency* of Huffman coding. Existing presentations of the ANS encoding and decoding algorithms are somewhat obscured by the lack of suitable presentation techniques; we present here an equational derivation, calculational where it can be, and highlighting the creative leaps where it cannot.

1 Introduction

Entropy encoding techniques compress symbols according to a model of their expected frequencies, with common symbols being represented by fewer bits than rare ones. The best known entropy encoding technique is *Huffman coding* (HC) [20], taught in every undergraduate course on algorithms and data structures: a classic greedy algorithm uses the symbol frequencies to construct a trie, from which an optimal *prefix-free binary code* can be read off. For example, suppose an alphabet of $n = 3$ symbols $s_0 =$ `'a'`, $s_1 =$ `'b'`, $s_2 =$ `'c'` with respective expected relative frequencies $c_0 = 2$, $c_1 = 3$, $c_2 = 5$ (that is, `'a'` is expected $^2/_{2+3+5} = 20\%$ of the time, and so on); then HC might construct the trie and prefix-free code shown in Figure 1. A text is then encoded as the concatenation of its symbol codes; thus, the text `"cbcacbcacb"` encodes to 1 01 1 00 1 01 1 00 1 01. This is optimal, in the sense that no prefix-free binary code yields a shorter encoding of any text that matches the expected symbol frequencies.

But HC is only 'optimal' among encodings that use a whole number of bits per symbol; if that constraint is relaxed, more effective encoding becomes possible. Note that the two symbols `'a'` and `'b'` were given equal-length codes 00 and 01



`'a'` $\mapsto$ 00
`'b'` $\mapsto$ 01
`'c'` $\mapsto$ 1

Fig. 1. A Huffman trie and the corresponding prefix-free code

by HC, despite having unequal frequencies—indeed, any expected frequencies in the same order $c_0 < c_1 < c_2$ will give the same code. More starkly, if the alphabet has only two symbols, HC can do no better than to code each symbol as a single bit, whatever their expected frequencies; that might be acceptable when the frequencies are similar, but is unacceptable when they are not.

Arithmetic coding (AC) [25, 29] is an entropy encoding technique that allows for a fractional number of bits per symbol. In a nutshell, a text is encoded as a half-open subinterval of the unit interval; the model assigns disjoint subintervals of the unit interval to each symbol, in proportion to their expected frequencies (as illustrated on the left of Figure 2); encoding starts with the unit interval, and narrows this interval by the model subinterval for each symbol in turn (the narrowing operation is illustrated on the right of Figure 2). The encoding is the shortest binary fraction in the final interval, without its final '1'. For example, with the model illustrated in Figure 2, the text `"abc"` gets encoded via the narrowing sequence of intervals

$$[0,1) \xrightarrow{\texttt{'a'}} [0,{}^1\!/_5) \xrightarrow{\texttt{'b'}} [{}^1\!/_{25},{}^1\!/_{10}) \xrightarrow{\texttt{'c'}} [{}^7\!/_{100},{}^1\!/_{10})$$

from which we pick the binary fraction ${}^3\!/_{32}$ (since ${}^7\!/_{100} \leqslant {}^3\!/_{32} < {}^1\!/_{10}$) and output the bit sequence 0001. We formalize this sketched algorithm in Section 3.

This doesn't look like much saving: this particular example is only one bit shorter than with HC; and similarly, the arithmetic coding of the text `"cbcacbcacb"` is 14 bits, where HC uses 15 bits. But AC can do much better; for example, it encodes the permutation `"cabbacbccc"` of that text in 7 bits, whereas of course HC uses the same 15 bits as before.

In fact, AC is *Shannon-optimal*: the number of bits used tends asymptotically to the Shannon entropy of the message—the sum $\sum_i -\log_2 p_i$ of the negative logarithms of the symbol probabilities. Moreover, AC can be readily made *adaptive*, whereby the model evolves as the text is read, whereas HC entails separate modelling and encoding phases.

However, AC does have some problems. One problem is a historical accident: specific applications of the technique became mired in software patents in the

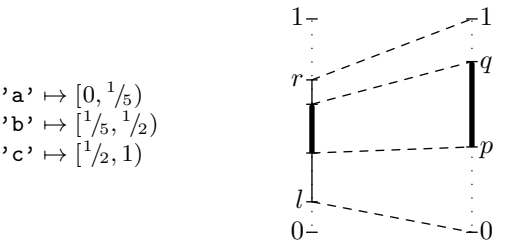


Fig. 2. A text model in interval form. Narrowing interval $[l, r)$ by interval $[p, q)$ yields the interval marked in bold on the left, which stands in relation to $[l, r)$ as $[p, q)$ does to $[0, 1)$.

1980s, and although those patents have now mostly expired, the consequences are still being felt (for example, Seward's `bzip` compressor [26] switched in 1996 from AC to HC because of patents, and has not switched back since). A more fundamental problem is that AC involves a lot of arithmetic, and even after slightly degrading coding effectiveness in order to use only single-word fixed-precision rather than arbitrary-precision arithmetic, state-of-the-art implementations are still complicated and relatively slow.

A recent development that addresses both of these problems has been Jarek Duda's introduction of *asymmetric numeral systems* (ANS) [11, 12, 14]. This is another entropy encoding technique; in a nutshell, rather than encoding longer and longer messages as narrower and narrower subintervals of the unit interval, they are simply encoded as larger and larger integers. Concretely, with the same frequency counts $c_0 = 2, c_1 = 3, c_2 = 5$ as before, and cumulative totals $t_0 = 0, t_1 = t_0 + c_0 = 2, t_2 = t_1 + c_1 = 5, t = t_2 + c_2 = 10$, encoding starts with an accumulator at 0, and for each symbol s_i (traditionally from right to left in the text) maps the current accumulator x to $(x \operatorname{div} c_i) \times t + t_i + (x \bmod c_i)$, as illustrated in Figure 3. Thus, the text `"abc"` gets encoded via the increasing (read from right to left) sequence of integers:

$$70 \xleftarrow{\texttt{'a'}} 14 \xleftarrow{\texttt{'b'}} 5 \xleftarrow{\texttt{'c'}} 0$$

It is evident even from this brief sketch that the encoding process is quite simple, with just a single division and multiplication per symbol; it turns out that decoding is just as simple. The encoding seems quite mysterious, but it is very cleverly constructed, and again achieves Shannon-optimal encoding; ANS combines the *effectiveness* of AC with the *efficiency* of Huffman coding, addressing the more fundamental concern with AC. The purpose of this paper is to motivate and justify the development, using calculational techniques where possible.

As it happens, Duda is also fighting to keep ANS in the public domain, despite corporate opposition [22], thereby addressing the more accidental concern too. These benefits have seen ANS recently adopted by large companies for

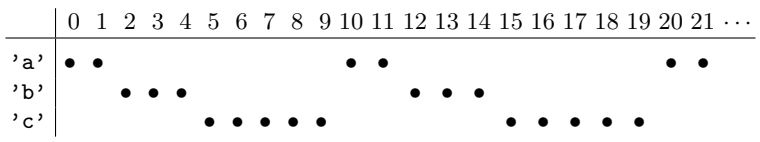


Fig. 3. The start of the coding table for alphabet `'a'`, `'b'`, `'c'` with counts $2, 3, 5$. The indices $0\,..$ are distributed across the alphabet, in proportion to the counts: two for `'a'`, three for `'b'`, and so on. Encoding symbol s with current accumulator x yields the index of the xth blob in row s as the new accumulator. For example, with $x = 4$ and next symbol $s = \texttt{'b'} = s_i$ with $i = 1$, we have $c_i = 3, t_i = 2, t = 10$ so $x' = (x \operatorname{div} c_i) \times t + t_i + (x \bmod c_i) = 13$, and indeed the 4th blob in row `'b'` (counting from zero) is in column 13.

products such as Facebook Zstd [9], Apple LZFSE [10], Google Draco [7], and Dropbox DivANS [24], and ANS is expected [1] to be featured in the forthcoming JPEG XL standard [21].

One disadvantage of ANS is that, whereas AC acts in a first-in first-out manner, ANS acts last-in first-out, in the sense that the decoded text comes out in the reverse order to which it went in. Our development will make clear where this happens. This reversal makes ANS unsuitable for encoding a communications channel, and also makes it difficult to employ adaptive text models. (DivANS [24] processes the input forwards for statistical modelling, and then uses this information backwards to encode the text; one could alternatively batch process the text in fixed-size blocks. In some contexts, such as encoding the video stream of a movie for distribution to set-top boxes, it is worth expending more effort in offline encoding in order to benefit online decoding.)

The remainder of this paper is structured as follows. Section 2 recaps various well-known properties of folds and unfolds on lists. Section 3 presents the relevant basics of AC, and Section 4 a proof of correctness of this basic algorithm. Section 5 presents the key step from AC to ANS, namely the switch from accumulating fractions to accumulating integers. Section 6 shows how to modify this naive ANS algorithm to work in bounded precision, and Section 7 shows how to make the resulting program 'stream' (to start generating output before consuming all the input). Section 8 discusses related work and concludes.

We use Haskell [23] as an algorithmic notation. Note that function application binds tightest of all binary operators, so that for example $f\ x + y$ means $(f\ x) + y$; apart from that, we trust that the notation is self-explanatory. We give definitions of functions from the Haskell standard library as we encounter them. The code from the paper is available online [16], as is a longer version [17] of the paper including proofs and other supporting material.

This longer version of the paper gives proofs and other additional material, for completeness. The differences from the published paper are all highlighted in red, like this paragraph.

2 Origami programming

In this section, we recap some well-studied laws of folds

$$\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ e\ [\,] \qquad = e \\
&\mathit{foldr}\ f\ e\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ e\ [\,] \qquad = e \\
&\mathit{foldl}\ f\ e\ (x : xs) = \mathit{foldl}\ f\ (f\ e\ x)\ xs
\end{aligned}$$

and unfolds

$$\begin{aligned}&\mathit{unfoldr} :: (b \to \mathit{Maybe}\ (a, b)) \to b \to [a]\\ &\mathit{unfoldr}\ f\ y = \mathbf{case}\ f\ y\ \mathbf{of}\ \mathit{Nothing} \quad \to [\,]\\ &\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (x, y') \to x : \mathit{unfoldr}\ f\ y'\end{aligned}$$

on lists.

Folds. The *First Duality Theorem* of *foldl* and *foldr* [5, §3.5.1] states that

$$\mathit{foldr}\ f\ e = \mathit{foldl}\ f\ e$$

when f and e form a monoid. The *Third Duality Theorem*, from the same source, says:

$$\mathit{foldr}\ f\ e \cdot \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ f)\ e$$

where $\mathit{flip}\ f\ a\ b = f\ b\ a$ swaps the arguments of a binary function. (The published version [5, §3.5.1] has the *reverse* on the other side, and holds only for finite lists.)

We will also use the *Fusion Law* for *foldr* [4, §4.6.2]:

$$h \cdot \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e' \quad \Leftarrow \quad h\ e = e' \wedge h\ (f\ x\ y) = f'\ x\ (h\ y)$$

(for strict h, or on finite lists), and its corollaries the *Map Fusion* laws:

$$\begin{aligned}&\mathit{foldr}\ f\ e \cdot \mathit{map}\ g = \mathit{foldr}\ (\lambda x\ y \to f\ (g\ x)\ y)\ e\\ &\mathit{foldl}\ f\ e \cdot \mathit{map}\ g = \mathit{foldl}\ (\lambda x\ y \to f\ x\ (g\ y))\ e\end{aligned}$$

Unfolds. There is a fusion law for unfolds [18] too:

$$\mathit{unfoldr}\ f \cdot g = \mathit{unfoldr}\ f' \quad \Leftarrow \quad f \cdot g = \mathit{fmap}_{\mathit{List}}\ g \cdot f'$$

Here, the *fmap* instance is for the base functor for lists; that is,

$$\begin{aligned}&\mathit{fmap}_{\mathit{List}} :: (b \to c) \to \mathit{Maybe}\ (a, b) \to \mathit{Maybe}\ (a, c)\\ &\mathit{fmap}_{\mathit{List}}\ h\ (\mathit{Just}\ (a, b)) = \mathit{Just}\ (a, h\ b)\\ &\mathit{fmap}_{\mathit{List}}\ g\ \mathit{Nothing} \quad\ = \mathit{Nothing}\end{aligned}$$

As it happens, the Haskell 98 standard does not mention *unfoldr* fusion; the sole law of *unfoldr* it does mention [23, §17.4] gives conditions under which it inverts a *foldr*: if

$$\begin{aligned}&g\ (f\ x\ z) = \mathit{Just}\ (x, z)\\ &g\ e \quad\ = \mathit{Nothing}\end{aligned}$$

for all x and z, then

$$\mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ xs) = xs$$

for all finite lists xs. The proof is a straightforward induction on xs. The base case is $xs = [\,]$, for which we have

$$\begin{aligned}&\mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ [\,])\\ =\ &\quad \{\ \mathit{foldr}\ \}\\ &\mathit{unfoldr}\ g\ e\end{aligned}$$

$$\begin{array}{ll} & \{ \text{ assumption: } g\ e = \mathit{Nothing};\ \mathit{unfoldr}\ \} \\ & [\,] \end{array}$$

For the inductive step, assume that the theorem holds for xs. Then:

$$\begin{array}{ll} & \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ (x : xs)) \\ = & \quad \{\ \mathit{foldr}\ \} \\ & \mathit{unfoldr}\ g\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\ = & \quad \{ \text{ assumption: } g\ (f\ x\ z) = \mathit{Just}\ (x, z)\ \} \\ & x : \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ xs) \\ = & \quad \{ \text{ inductive hypothesis } \} \\ & x : xs \end{array}$$

We call this the *Unfoldr–Foldr Theorem*.

We make two generalisations to this theorem. The first, the *Unfoldr–Foldr Theorem with Junk*, allows the unfold to continue after reconstructing the original list: if only

$$g\ (f\ x\ z) = \mathit{Just}\ (x, z)$$

holds, for all x and z, then

$$\exists ys\ .\ \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ xs) = xs +\!\!+ ys$$

for all finite lists xs—that is, the *unfoldr* inverts the *foldr* except for appending some (possibly infinite) junk ys to the output. (The proof is again by induction on xs: the base case vacuously holds, and the inductive step is essentially the same as without junk.)

The second generalisation is the *Unfoldr–Foldr Theorem with Invariant*. We say that predicate p is an invariant of $\mathit{foldr}\ f\ e$ and $\mathit{unfoldr}\ g$ if

$$\begin{array}{lcl} p\ (f\ x\ z) & \Leftarrow & p\ z \\ p\ z' & \Leftarrow & p\ z \wedge g\ z = \mathit{Just}\ (x, z') \end{array}$$

for all x, z, z'. The theorem states that if p is such an invariant, and the conditions

$$\begin{array}{lcl} g\ (f\ x\ z) = \mathit{Just}\ (x, z) & \Leftarrow & p\ z \\ g\ e = \mathit{Nothing} & \Leftarrow & p\ e \end{array}$$

hold for all x and z, then

$$\mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ xs) = xs \quad \Leftarrow \quad p\ e$$

for all finite lists xs.

Again, the proof is by induction on xs. Note the auxilliary lemma that $p\ e$ implies $p\ (\mathit{foldr}\ f\ e\ xs)$ for any finite xs. Now assume that $p\ e$ holds. Then for $xs = [\,]$, we have

$$\begin{array}{ll} & \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ [\,]) \\ = & \quad \{\ \mathit{foldr}\ \} \\ & \mathit{unfoldr}\ g\ e \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ by assumption, } g\ e = \mathit{Nothing} \ \} \\
 & [\,]
\end{array}
$$

and for the inductive step, assume that the result holds for xs, and then we have:

$$
\begin{array}{ll}
 & \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ (x : xs)) \\
= & \quad \{\ \mathit{foldr}\ \} \\
 & \mathit{unfoldr}\ g\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\
= & \quad \{\ p\ (\mathit{foldr}\ f\ e\ xs) \text{ holds, so } g\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) = \mathit{Just}\ (x, \mathit{foldr}\ f\ e\ xs)\ \} \\
 & x : \mathit{unfoldr}\ g\ (\mathit{foldr}\ f\ e\ xs) \\
= & \quad \{ \text{ inductive hypothesis } \} \\
 & x : xs
\end{array}
$$

And of course, there is an *Unfoldr–Foldr Theorem with Junk and Invariant*, incorporating both generalisations; this is the version we will actually use.

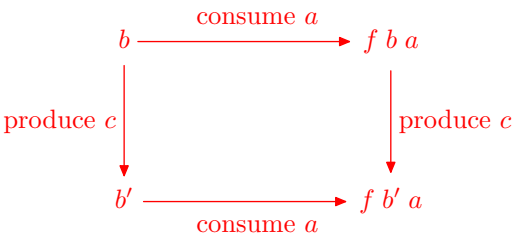
Metamorphisms. A *metamorphism* [15] consists of an *unfoldr* after a *foldl*:

$$\mathit{unfoldr}\ g \cdot \mathit{foldl}\ f\ e$$

In this plain form, it consumes one list and produces another. Because the *foldl* is tail-recursive, it consumes the whole input before producing any output. However, sometimes an initial segment of the input is sufficient to determine a corresponding initial segment of the output; then that part of the output may be produced early, before consuming the rest of the input. The condition under which this can be done is called the *streaming condition* for g and f:

$$g\ b = \mathit{Just}\ (c, b') \quad \Rightarrow \quad \forall a\ .\ g\ (f\ b\ a) = \mathit{Just}\ (c, f\ b'\ a)$$

Informally, the streaming condition states that for any productive state b, from which the producer g produces an output c and moves to a new state b', the consumption of any input a preserves productivity of the state $f\ b\ a$, and moreover, the same output c will be produced, and the new state $f\ b'\ a$ will be the same as would have resulted from producing c before consuming a. That is, the operations 'produce c' and 'consume a' commute:



If the streaming condition holds for g and f, then

$$\mathit{unfold}\ g\ (\mathit{foldl}\ f\ e\ x) = \mathit{stream}\ g\ f\ e\ x$$

for all finite lists x, where

$$
\begin{array}{l}
\mathit{stream} :: (b \to \mathit{Maybe}\ (c, b)) \to (b \to a \to b) \to b \to [a] \to [c] \\
\mathit{stream}\ g\ f\ b\ x = \mathbf{case}\ g\ b\ \mathbf{of} \\
\quad \mathit{Just}\ (c, b') \to c : \mathit{stream}\ g\ f\ b'\ x \\
\quad \mathit{Nothing} \quad \to \mathbf{case}\ x\ \mathbf{of} \\
\qquad a : x' \to \mathit{stream}\ g\ f\ (f\ b\ a)\ x' \\
\qquad [\,] \quad \to [\,]
\end{array}
$$

Thus, $\mathit{stream}\ g\ f$ produces an element if it can, consumes one if it cannot produce, and terminates if there is nothing left to consume. Significantly, $\mathit{stream}\ g\ f\ e$ can operate on infinite lists, even though the two-phase process $\mathit{unfoldr}\ g \cdot \mathit{foldl}\ f\ e$ cannot.

Sometimes streaming almost works, but needs to be divided into a 'cautious phase' in which input still remains, followed by a 'reckless phase' once all the input has been consumed:

$$
\begin{array}{l}
\mathit{fstream} :: (b \to \mathit{Maybe}\ (c, b)) \to (b \to [c]) \to (b \to a \to b) \to b \to [a] \to [c] \\
\mathit{fstream}\ g\ h\ f\ b\ x = \mathbf{case}\ g\ b\ \mathbf{of} \\
\quad \mathit{Just}\ (c, b') \to c : \mathit{fstream}\ g\ h\ f\ b'\ x \\
\quad \mathit{Nothing} \quad \to \mathbf{case}\ x\ \mathbf{of} \\
\qquad a : x' \to \mathit{fstream}\ g\ h\ f\ (f\ b\ a)\ x' \\
\qquad [\,] \quad \to h\ b
\end{array}
$$

Compared to plain stream, this takes an additional argument h, which is applied to the final state b in order to 'flush the buffer' once the input is exhausted. The corresponding theorem states that, provided that the streaming condition holds for g and f, then

$$\mathit{apo}\ g\ h \cdot \mathit{foldl}\ f\ e = \mathit{fstream}\ g\ h\ f\ e$$

where apo captures *list apomorphisms* [28]:

$$
\begin{array}{l}
\mathit{apo} :: (b \to \mathit{Maybe}\ (c, b)) \to (b \to [c]) \to b \to [c] \\
\mathit{apo}\ g\ h\ b = \mathbf{case}\ g\ b\ \mathbf{of} \\
\quad \mathit{Just}\ (c, b') \to \mathit{apo}\ g\ h\ b' \\
\quad \mathit{Nothing} \quad \to h\ b
\end{array}
$$

(that is, like an unfold, but switching to a flushing phase using h at the end).

Of course, any unfold is trivially an apomorphism, with the trivial flusher that always yields the empty list. More interestingly, any unfold can be factored into a cautious phase (delivering elements only while a given predicate p holds) followed by a reckless phase (ignoring p, and delivering the elements anyway):

$$\mathit{unfoldr}\ g = \mathit{apo}\ (\mathit{guard}\ p\ g)\ (\mathit{unfoldr}\ g)$$

where

$$
\begin{array}{l}
\mathit{guard} :: (b \to \mathit{Bool}) \to (b \to \mathit{Maybe}\ (c, b)) \to (b \to \mathit{Maybe}\ (c, b)) \\
\mathit{guard}\ p\ g\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ g\ x\ \mathbf{else}\ \mathit{Nothing}
\end{array}
$$

In particular, the streaming condition may hold for the cautious coalgebra $guard\ p\ g$ when it doesn't hold for the reckless coalgebra g itself. We will make use of this construction later.

3 Arithmetic coding

We start from a simplified version of arithmetic coding: we use a fixed rather than adaptive model, and rather than picking the shortest binary fraction within the final interval, we simply pick the lower bound of the interval.

Intervals and symbols. We represent intervals as pairs of rationals,

$$\textbf{type}\ \mathit{Interval} = (\mathit{Rational}, \mathit{Rational})$$

so the unit interval is $\mathit{unit} = (0, 1)$ and the lower bound is obtained by fst. We suppose a symbol table

$$\mathit{counts} :: \mathit{Integral}\ n \Rightarrow [(\mathit{Symbol}, n)]$$

that records a positive count for every symbol in the alphabet (note the *Integral* class constraint: we allow the counts to be both arbitrary-precision *Integer*s here, and fixed-precision *Int*s later). From this we obtain the following functions:

$$\begin{aligned}
&\mathit{symbols} :: [\mathit{Symbol}]\\
&\mathit{symbols} = \mathit{map}\ \mathit{fst}\ \mathit{counts}\\
&\\
&t :: \mathit{Integral}\ n \Rightarrow n\\
&t = \mathit{sum}\ (\mathit{map}\ \mathit{snd}\ \mathit{counts})\\
&\\
&\mathit{freqs} :: [(\mathit{Symbol}, \mathit{Rational})]\\
&\mathit{freqs} = [(s, c \,\%\, t) \mid (s, c) \leftarrow \mathit{counts}]\\
&\\
&\mathit{freq} :: \mathit{Symbol} \to \mathit{Rational}\\
&\mathit{freq} = \mathit{justLookup}\ \mathit{freqs}\\
&\\
&\mathit{fcumuls} :: [(\mathit{Symbol}, \mathit{Rational})]\\
&\mathit{fcumuls} = \mathit{zip}\ \mathit{symbols}\ (\mathit{scanl}\ (+)\ 0\ (\mathit{map}\ \mathit{snd}\ \mathit{freqs}))\\
&\\
&\mathit{fcumul} :: \mathit{Symbol} \to \mathit{Rational}\\
&\mathit{fcumul} = \mathit{justLookup}\ \mathit{fcumuls}\\
&\\
&\mathit{encodeSym} :: \mathit{Symbol} \to \mathit{Interval}\\
&\mathit{encodeSym}\ s = (l, l + \mathit{freq}\ s)\ \textbf{where}\ l = \mathit{fcumul}\ s\\
&\\
&\mathit{decodeSym} :: \mathit{Rational} \to \mathit{Symbol}\\
&\mathit{decodeSym}\ x = \mathit{last}\ [s \mid (s, y) \leftarrow \mathit{fcumuls}, y \leqslant x]
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{justLookup} :: \mathit{Eq}\ a \Rightarrow [(a, b)] \to a \to b\\
&\mathit{justLookup}\ \mathit{abs}\ a = \mathit{fromJust}\ (\mathit{lookup}\ a\ \mathit{abs})
\end{aligned}$$

retrieves a present item from an association list. In particular, *decodeSym* and *encodeSym* satisfy the central property: for $x \in \mathit{unit}$,

$$decodeSym\ x = s \quad \Leftrightarrow \quad x \in encodeSym\ s$$

For example, with the same alphabet of three symbols `'a'`, `'b'`, `'c'` and counts 2, 3, and 5 as before, we have $encodeSym\ \texttt{'b'} = (1/5, 1/2)$ and $decodeSym\ (1/3) = \texttt{'b'}$.

We have operations on intervals:

$$\begin{aligned}
&weight, scale :: Interval \to Rational \to Rational\\
&weight\ (l, r)\ x\ = l + (r - l) \times x\\
&scale\ (l, r)\ y\ \ = (y - l)\ /\ (r - l)
\end{aligned}$$

$$\begin{aligned}
&narrow\ \ :: Interval \to Interval \to Interval\\
&narrow\ i\ (p, q) = (weight\ i\ p, weight\ i\ q)
\end{aligned}$$

that satisfy

$$\begin{aligned}
&weight\ i\ x \in i \quad \Leftrightarrow \quad x \in unit\\
&weight\ i\ x = y \quad \Leftrightarrow \quad scale\ i\ y = x
\end{aligned}$$

Informally, $weight\ (l, r)\ x$ is 'fraction x of the way between l and r', and conversely $scale\ (l, r)\ y$ is 'the fraction of the way y is between l and r'; and *narrow* is illustrated in Figure 2.

Encoding and decoding. Now we can specify arithmetic encoding and decoding by:

$$\begin{aligned}
&encode_1 :: [Symbol] \to Rational\\
&encode_1 = fst \cdot foldl\ estep_1\ unit
\end{aligned}$$

$$\begin{aligned}
&estep_1 :: Interval \to Symbol \to Interval\\
&estep_1\ i\ s = narrow\ i\ (encodeSym\ s)
\end{aligned}$$

$$\begin{aligned}
&decode_1 :: Rational \to [Symbol]\\
&decode_1 = unfoldr\ dstep_1
\end{aligned}$$

$$\begin{aligned}
&dstep_1 :: Rational \to Maybe\ (Symbol, Rational)\\
&dstep_1\ x = \textbf{let}\ s = decodeSym\ x\ \textbf{in}\ Just\ (s, scale\ (encodeSym\ s)\ x)
\end{aligned}$$

For example, with the same alphabet and counts, the input text `"abc"` gets encoded symbol by symbol, from left to right (because of the *foldl*), starting with the unit interval $(0, 1)$, via the narrowing sequence of intervals

$$\begin{aligned}
&estep_1\ (0, 1)\ \texttt{'a'}\ \ \ \ \ \ = (0, 1/5)\\
&estep_1\ (0, 1/5)\ \texttt{'b'}\ \ \ \ \ = (1/25, 1/10)\\
&estep_1\ (1/25, 1/10)\ \texttt{'c'} = (7/100, 1/10)
\end{aligned}$$

from which we select the lower bound $7/100$ of the final interval. Conversely, decoding starts with $7/100$, and proceeds as follows:

$$\begin{aligned}
&dstep_1\ (7/100) = Just\ (\texttt{'a'}, 7/20)\\
&dstep_1\ (7/20)\ \ = Just\ (\texttt{'b'}, 1/2)\\
&dstep_1\ (1/2)\ \ \ = Just\ (\texttt{'c'}, 0)
\end{aligned}$$

$$dstep_1\ 0 \quad = Just\ (\texttt{'a'}, 0)$$
$$\ldots$$

Note that decoding runs forever; but the finite encoded text is a prefix of the decoded output—for any input text xs, there is an infinite sequence of junk ys such that

$$decode_1\ (encode_1\ xs) = xs +\!\!+ ys$$

(indeed, $ys = repeat\ \texttt{'a'}$ when we pick the *fst* of an interval).

4 Correctness of arithmetic coding

Using the laws of folds, we can 'fission' the symbol encoding out of $encode_1$, turn the *foldl* into a *foldr* (because *narrow* and *unit* form a monoid), fuse the *fst* with the *foldr*, and then re-fuse the symbol encoding with the fold:

$$\begin{aligned}
&\ encode_1 \\
=&\quad \{\ \text{definition}\ \} \\
&\ fst \cdot foldl\ estep_1\ unit \\
=&\quad \{\ \text{map fusion for } foldl\text{, backwards}\ \} \\
&\ fst \cdot foldl\ narrow\ unit \cdot map\ encodeSym \\
=&\quad \{\ \text{duality: } narrow \text{ and } unit \text{ form a monoid}\ \} \\
&\ fst \cdot foldr\ narrow\ unit \cdot map\ encodeSym \\
=&\quad \{\ \text{fusion for } foldr \text{ (see below)}\ \} \\
&\ foldr\ weight\ 0 \cdot map\ encodeSym \\
=&\quad \{\ \text{map fusion; let } estep_2\ s\ x = weight\ (encodeSym\ s)\ x\ \} \\
&\ foldr\ estep_2\ 0
\end{aligned}$$

For the fusion step, we have $fst\ unit = 0$, and

$$\begin{aligned}
&\ fst\ (narrow\ i\ (p, q)) \\
=&\quad \{\ narrow\ \} \\
&\ fst\ (weight\ i\ p, weight\ i\ q) \\
=&\quad \{\ fst\ \} \\
&\ weight\ i\ p \\
=&\quad \{\ fst\ \} \\
&\ weight\ i\ (fst\ (p, q))
\end{aligned}$$

as required. So we have calculated $encode_1 = encode_2$, where

$$\begin{aligned}
&encode_2 :: [Symbol] \to Rational \\
&encode_2 = foldr\ estep_2\ 0 \\
&\\
&estep_2 :: Symbol \to Rational \to Rational \\
&estep_2\ s\ x = weight\ (encodeSym\ s)\ x
\end{aligned}$$

Now encoding is a simple *foldr*, which means that it is easier to manipulate.

Inverting encoding. Let us turn now to decoding, and specifically the question of whether it faithfully decodes the encoded text. We use the Unfoldr–Foldr Theorem. Of course, we have to accept junk, because our decoder runs indefinitely. We check the inductive condition:

$$\begin{array}{ll} & dstep_1\ (estep_2\ s\ x) \\ = & \quad \{\ estep_2;\ \text{let}\ x' = weight\ (encodeSym\ s)\ x\ \} \\ & dstep_1\ x' \\ = & \quad \{\ dstep_1;\ \text{let}\ s' = decodeSym\ x'\ \} \\ & Just\ (s', scale\ (encodeSym\ s')\ x') \end{array}$$

Now, we hope to recover the first symbol; that is, we require $s' = s$:

$$\begin{array}{ll} & s' = s \\ \Leftrightarrow & \quad \{\ s' = decodeSym\ x';\ \text{central property}\ \} \\ & x' \in encodeSym\ s \\ \Leftrightarrow & \quad \{\ \text{definition of}\ x'\ \} \\ & weight\ (encodeSym\ s)\ x \in encodeSym\ s \\ \Leftrightarrow & \quad \{\ \text{property of}\ weight\ \} \\ & x \in unit \end{array}$$

Fortunately, it is an invariant of the computation that the state x is in the unit interval: of course $0 \in unit$, and we have:

$$\begin{array}{ll} & estep_2\ s\ x \in unit \\ \Leftrightarrow & \quad \{\ estep_2\ \} \\ & weight\ (encodeSym\ s)\ x \in unit \\ \Leftarrow & \quad \{\ transitivity, \text{since}\ encodeSym\ s \subseteq unit\ \} \\ & weight\ (encodeSym\ s)\ x \in encodeSym\ s \\ \Leftrightarrow & \quad \{\ weight\ \} \\ & x \in unit \end{array}$$

and when $x \in unit$ and $dstep_1\ x = Just\ (s, x')$ we have:

$$\begin{array}{ll} & x' \in unit \\ \Leftrightarrow & \quad \{\ dstep_1;\ \text{let}\ s = decodeSym\ x\ \} \\ & scale\ (encodeSym\ s)\ x \in unit \\ \Leftrightarrow & \quad \{\ \text{property of}\ weight\ \} \\ & weight\ (encodeSym\ s)\ (scale\ (encodeSym\ s)\ x) \in encodeSym\ s \\ \Leftrightarrow & \quad \{\ weight\ i \cdot scale\ i = id\ \} \\ & x \in encodeSym\ s \\ \Leftrightarrow & \quad \{\ decodeSym\ x = s;\ \text{central property of model}\ \} \\ & True \end{array}$$

So indeed $s' = s$. Continuing:

$$\begin{array}{ll} & dstep_1\ (estep_2\ s\ x) \\ = & \quad \{\ \text{above}\ \} \\ & Just\ (s, scale\ (encodeSym\ s)\ (weight\ (encodeSym\ s)\ x)) \end{array}$$

$$\begin{array}{ll} = & \{\ \mathit{scale}\ i \cdot \mathit{weight}\ i = \mathit{id}\ \} \\ & \mathit{Just}\ (s, x) \end{array}$$

as required. Therefore, by the Unfoldr–Foldr Theorem with Junk and Invariant, decoding inverts encoding, up to junk: for all finite xs,

$$\exists ys\ .\ \mathit{decode}_1\ (\mathit{encode}_2\ xs) = xs +\!\!+ ys$$

But we can discard the junk, by pruning to the desired length:

$$\mathit{take}\ (\mathit{length}\ xs)\ (\mathit{decode}_1\ (\mathit{encode}_2\ xs)) = xs$$

for all finite xs. Alternatively, we can use an 'end of text' marker ω that is distinct from all proper symbols:

$$\mathit{takeWhile}\ (\neq \omega)\ (\mathit{decode}_1\ (\mathit{encode}_2\ (xs +\!\!+ [\omega]))) = xs \quad \Leftarrow \quad \omega \notin xs$$

for all finite xs. Either way, arithmetic decoding does indeed faithfully invert arithmetic coding.

5 From fractions to integers

We now make the key step from AC to ANS. Whereas AC encodes longer and longer messages as more and more precise fractions, ANS encodes them as larger and larger integers. Given the symbol table *counts* and sum of counts t as before, we obtain the following functions:

```
count  :: Integral n ⇒ Symbol → n
count  = justLookup counts

cumul  :: Integral n ⇒ Symbol → n
cumul  = justLookup cumuls

cumuls :: Integral n ⇒ [(Symbol, n)]
cumuls = zip symbols (scanl (+) 0 (map snd counts))

find   :: Integral n ⇒ n → Symbol
find x = last [s | (s, y) ← cumuls, y ⩽ x]   -- assuming x < t
```

such that *count* s gives the count of symbol s, *cumul* s gives the cumulative counts of all symbols preceding s in the symbol table, and *find* x looks up an integer $0 \leqslant x < t$:

$$\mathit{find}\ x = s \quad \Leftrightarrow \quad \mathit{cumul}\ s \leqslant x < \mathit{cumul}\ s + \mathit{count}\ s$$

The integer encoding step. We encode a text as an integer x, containing $\log_2 x$ bits of information. The next symbol s to encode has probability $p = \mathit{count}\ s\ /\ t$, and so requires an additional $\log_2 (1/p)$ bits; in total, that makes $\log_2 x + \log_2 (1/p) = \log_2 (x/p) = \log_2 (x \times t\ /\ \mathit{count}\ s)$ bits. So entropy considerations tell us that, roughly speaking, to incorporate symbol s into state x we want to map x to $x' \simeq x \times t\ /\ \mathit{count}\ s$. Of course, in order to decode, we need to

be able to invert this transformation, to extract s and x from x'; this suggests that we should do the division by $\mathit{count}\ s$ first:

$$x' = (x \operatorname{div} \mathit{count}\ s) \times t \quad \text{-- not final}$$

so that the multiplication by the known value t can be undone first:

$$x \operatorname{div} \mathit{count}\ s = x' \operatorname{div} t$$

(we will refine this definition shortly). How do we reconstruct s? Well, there is enough headroom in x' to add any value u with $0 \leqslant u < t$ without affecting the division; in particular, we can add $\mathit{cumul}\ s$ to x', and then we can use find on the remainder:

$$x' = (x \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s \quad \text{-- still not final}$$

so that

$$\begin{array}{ll} x \operatorname{div} \mathit{count}\ s & = x' \operatorname{div} t \\ \mathit{cumul}\ s & = x' \bmod t \\ s & = \mathit{find}\ (\mathit{cumul}\ s) = \mathit{find}\ (x' \bmod t) \end{array}$$

(this version still needs to be refined further). We are still missing some information from the lower end of x through the division, namely $x \bmod \mathit{count}\ s$; so we can't yet reconstruct x. Happily,

$$\mathit{find}\ (\mathit{cumul}\ s) = \mathit{find}\ (\mathit{cumul}\ s + r)$$

for any r with $0 \leqslant r < \mathit{count}\ s$; of course, $x \bmod \mathit{count}\ s$ is in this range, so there is still precisely enough headroom in x' to add this lost information too, without affecting the find, allowing us also to reconstruct x:

$$x' = (x \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s + x \bmod \mathit{count}\ s \quad \text{-- final}$$

so that

$$\begin{array}{ll} x \operatorname{div} \mathit{count}\ s & = x' \operatorname{div} t \\ s & = \mathit{find}\ (\mathit{cumul}\ s + x \bmod \mathit{count}\ s) \\ & = \mathit{find}\ (x' \bmod t) \\ x & = \mathit{count}\ s \times (x \operatorname{div} \mathit{count}\ s) + x \bmod \mathit{count}\ s \\ & = \mathit{count}\ s \times (x' \operatorname{div} t) + x' \bmod t - \mathit{cumul}\ s \end{array}$$

This is finally the transformation we will use for encoding one more symbol.

Integer ANS. We define

$$\begin{array}{l} \mathit{encode}_3 :: [\mathit{Symbol}] \to \mathit{Integer} \\ \mathit{encode}_3 = \mathit{foldr}\ \mathit{estep}_3\ 0 \end{array}$$

$$\begin{array}{l} \mathit{estep}_3 :: \mathit{Integral}\ n \Rightarrow \mathit{Symbol} \to n \to n \\ \mathit{estep}_3\ s\ x = \mathbf{let}\ (q, r) = x \operatorname{divMod} \mathit{count}\ s\ \mathbf{in}\ q \times t + \mathit{cumul}\ s + r \end{array}$$

$$\begin{array}{l} \mathit{decode}_3 :: \mathit{Integer} \to [\mathit{Symbol}] \\ \mathit{decode}_3 = \mathit{unfoldr}\ \mathit{dstep}_3 \end{array}$$

$$
\begin{aligned}
&\mathit{dstep}_3 :: \mathit{Integral}\ n \Rightarrow n \to \mathit{Maybe}\ (\mathit{Symbol}, n)\\
&\mathit{dstep}_3\ x = \mathbf{let}\ (q, r) = x\ \mathrm{divMod}\ t\\
&\qquad\qquad\quad\ \ s = \mathit{find}\ r\\
&\qquad\qquad\ \mathbf{in}\ \mathit{Just}\ (s, \mathit{count}\ s \times q + r - \mathit{cumul}\ s)
\end{aligned}
$$

(Note again that we allow estep_3 and dstep_3 to work with both arbitrary-precision *Integer*s here and fixed-precision *Int*s later.)

Correctness of integer ANS. Using similar reasoning as for AC, we can show that a decoding step inverts an encoding step:

$$
\begin{aligned}
&\mathit{dstep}_3\ (\mathit{estep}_3\ s\ x)\\
=\quad &\{\ \mathit{estep}_3;\ \text{let}\ (q, r) = x\ \mathrm{divMod}\ \mathit{count}\ s,\ x' = q \times t + \mathit{cumul}\ s + r\ \}\\
&\mathit{dstep}_3\ x'\\
=\quad &\{\ \mathit{dstep}_3;\ \text{let}\ (q', r') = x'\ \mathrm{divMod}\ t,\ s' = \mathit{find}\ r'\ \}\\
&\mathit{Just}\ (s', \mathit{count}\ s' \times q' + r' - \mathit{cumul}\ s')\\
=\quad &\{\ r' = \mathit{cumul}\ s + r, 0 \leqslant r < \mathit{count}\ s, \text{so}\ s' = \mathit{find}\ r' = s\ \}\\
&\mathit{Just}\ (s, \mathit{count}\ s \times q' + r' - \mathit{cumul}\ s)\\
=\quad &\{\ r' - \mathit{cumul}\ s = r, q' = x' \operatorname{div} t = q\ \}\\
&\mathit{Just}\ (s, \mathit{count}\ s \times q + r)\\
=\quad &\{\ (q, r) = x\ \mathrm{divMod}\ \mathit{count}\ s\ \}\\
&\mathit{Just}\ (s, x)
\end{aligned}
$$

Therefore decoding inverts encoding, modulo pruning, by the Unfoldr–Foldr Theorem with Junk:

$$\mathit{take}\ (\mathit{length}\ \mathit{xs})\ (\mathit{decode}_3\ (\mathit{encode}_3\ \mathit{xs})) = \mathit{xs}$$

for all finite xs. For example, with the same alphabet and symbol counts as before, encoding the text `"abc"` proceeds (now from right to left, because of the *foldr* in encode_3) as follows:

$$
\begin{aligned}
&\mathit{estep}_3\ \texttt{'c'}\ 0 = 5\\
&\mathit{estep}_3\ \texttt{'b'}\ 5 = 14\\
&\mathit{estep}_3\ \texttt{'a'}\ 14 = 70
\end{aligned}
$$

and the result is 70. Decoding inverts this:

$$
\begin{aligned}
&\mathit{dstep}_3\ 70 = \mathit{Just}\ (\texttt{'a'}, 14)\\
&\mathit{dstep}_3\ 14 = \mathit{Just}\ (\texttt{'b'}, 5)\\
&\mathit{dstep}_3\ 5 = \mathit{Just}\ (\texttt{'c'}, 0)\\
&\mathit{dstep}_3\ 0 = \mathit{Just}\ (\texttt{'a'}, 0)\\
&\ldots
\end{aligned}
$$

Huffman as an instance of ANS. Incidentally, we can see here that ANS is in fact a generalisation of HC. If the symbol counts and their sum are all powers of two, then the arithmetic in estep_3 amounts to simple manipulation of bit vectors by shifting and insertion. For example, with an alphabet of four

symbols `'a'`, `'b'`, `'c'`, `'d'` with counts $4, 2, 1, 1$, encoding operates on a state x with binary expansion $\cdots x_3\ x_2\ x_1\ x_0$ (written most significant bit first) as follows:

$$\begin{aligned}
&\mathit{estep}_3\ \texttt{'a'}\ (\cdots x_3\ x_2\ x_1\ x_0) = \cdots x_3\ x_2\ 0\ x_1\ x_0\\
&\mathit{estep}_3\ \texttt{'b'}\ (\cdots x_3\ x_2\ x_1\ x_0) = \cdots x_3\ x_2\ x_1\ 1\ 0\ x_0\\
&\mathit{estep}_3\ \texttt{'c'}\ (\cdots x_3\ x_2\ x_1\ x_0) = \cdots x_3\ x_2\ x_1\ x_0\ 1\ 1\ 0\\
&\mathit{estep}_3\ \texttt{'d'}\ (\cdots x_3\ x_2\ x_1\ x_0) = \cdots x_3\ x_2\ x_1\ x_0\ 1\ 1\ 1
\end{aligned}$$

That is, the symbol codes $0, 10, 110, 111$ are inserted into rather than appended onto the state so far; the binary expansion of the ANS encoding of a text yields some permutation of the HC encoding of that text. (Clearly, every Huffman tree gives rise to a collection of frequencies that are inverse powers of two, and which sum to one; for example, the Huffman tree in Figure 1 yields frequencies $2^{-2}, 2^{-2}, 2^{-1}$. And the converse holds: every bag of values, each of which is of the form 2^{-k} and which together sum to one, corresponds to a Huffman tree [2].)

A different starting point. As it happens, the inversion property of encode_3 and decode_3 holds, whatever value we use to start encoding with (since this value is not used in the proof); in Section 6, we start encoding with a certain lower bound l rather than 0. Moreover, estep_3 is strictly increasing on states strictly greater than zero, and dstep_3 strictly decreasing; which means that the decoding process can stop when it returns to the lower bound. That is, if we pick some $l > 0$ and define

$$\begin{aligned}
&\mathit{encode}_4 :: [\mathit{Symbol}] \to \mathit{Integer}\\
&\mathit{encode}_4 = \mathit{foldr}\ \mathit{estep}_3\ l
\end{aligned}$$

$$\begin{aligned}
&\mathit{decode}_4 :: \mathit{Integer} \to [\mathit{Symbol}]\\
&\mathit{decode}_4 = \mathit{unfoldr}\ \mathit{dstep}_4
\end{aligned}$$

$$\begin{aligned}
&\mathit{dstep}_4 :: \mathit{Integer} \to \mathit{Maybe}\ (\mathit{Symbol}, \mathit{Integer})\\
&\mathit{dstep}_4\ x = \mathbf{if}\ x == l\ \mathbf{then}\ \mathit{Nothing}\ \mathbf{else}\ \mathit{dstep}_3\ x
\end{aligned}$$

then the stronger version of the Unfoldr–Foldr Theorem (without junk) holds, and we have

$$\mathit{decode}_4\ (\mathit{encode}_4\ \mathit{xs}) = \mathit{xs}$$

for all finite xs.

6 Bounded precision

The previous versions all used arbitrary-precision arithmetic, which is expensive. We now change the approach slightly to use only bounded-precision arithmetic. As usual, there is a trade-off between effectiveness (a bigger bound on the numbers involved means more accurate approximations to ideal entropies) and efficiency (a smaller bound generally means faster operations). Fortunately, the reasoning does not depend much on the actual bounds. We will pick a base b

and a lower bound l, and represent the integer accumulator x as a pair (w, ys) which we call a *Number*:

$$\textbf{type}\ \mathit{Number} = (\mathit{Int}, [\mathit{Int}])$$

such that ys is a list of digits in base b, and w is an integer in the range $l \leqslant w < u$ (where we define $u = l \times b$ for the upper bound), under the abstraction relation $x = \mathit{abstract}\ (w, ys)$ induced by

$$\begin{aligned}&\mathit{abstract} :: \mathit{Number} \to \mathit{Integer}\\ &\mathit{abstract}\ (w, ys) = \mathit{foldl}\ \mathit{inject}\ (\mathit{fromIntegral}\ w)\ (\mathit{map}\ \mathit{fromIntegral}\ ys)\end{aligned}$$

where

$$\mathit{inject}\ w\ y = w \times b + y$$

We call w the 'window' and ys the 'remainder'. For example, with $b = 10$ and $l = 100$, the pair $(123, [4, 5, 6])$ represents the value 123456.

Properties of the window. Specifying a range of the form $l \leqslant w < l \times b$ induces nice properties. If we introduce an operation

$$\mathit{extract}\ w = w\ \mathrm{divMod}\ b$$

as an inverse to *inject*, then we have

$$\begin{aligned}\mathit{inject}\ w\ y < u \quad &\Leftrightarrow \quad w < l\\ l \leqslant \mathit{fst}\ (\mathit{extract}\ w) \quad &\Leftrightarrow \quad u \leqslant w\end{aligned}$$

For the proofs of these properties, we make use of the universal property of integer division:

$$u < v \times w \quad \Leftrightarrow \quad u\ \mathrm{div}\ w < v$$

and its contrapositive:

$$u \geqslant v \times w \quad \Leftrightarrow \quad u\ \mathrm{div}\ w \geqslant v$$

Then:

$$\begin{aligned}&\mathit{inject}\ w\ y < u\\ \Leftrightarrow\quad &\{\ \mathit{inject}\ \}\\ &w \times b + y < u\\ \Leftrightarrow\quad &\{\ u\ \}\\ &w \times b + y < l \times b\\ \Leftrightarrow\quad &\{\ \text{universal property of } \mathit{div}\ \}\\ &(w \times b + y)\ \mathrm{div}\ b < l\\ \Leftrightarrow\quad &\{\ \text{assuming } 0 \leqslant y < b\ \}\\ &w < l\end{aligned}$$

and

$$\begin{aligned}&l \leqslant \mathit{fst}\ (\mathit{extract}\ w)\\ \Leftrightarrow\quad &\{\ \mathit{extract}\ \}\end{aligned}$$

$$
\begin{array}{ll}
& l \leqslant w \mathbin{\mathrm{div}} b \\
\Leftrightarrow & \quad \{ \text{ universal property of } \mathit{div} \ \} \\
& l \times b \leqslant w \\
\Leftrightarrow & \quad \{ \ u \ \} \\
& u \leqslant w
\end{array}
$$

That is, given an in-range window value w, injecting another digit will take it outside (above) the range; but if w is initially below the range, injecting another digit will keep it below the upper bound. So starting below the range and repeatedly injecting digits will eventually land within the range (it cannot hop right over), and injecting another digit would take it outside the range again. Conversely, given an in-range window value w, extracting a digit will take it outside (below) the range; but if w is initially above the range, extracting a digit will keep it at least the lower bound. So starting above the range and repeatedly extracting digits will also eventually land within the range (it cannot hop right over), and extracting another digit would take it outside the range again. This is illustrated in Figure 4. In particular, for any $x \geqslant l$ there is a unique representation of x under *abstract* that has an in-range window.

For fast execution, b should be a power of two, so that multiplication and division by b can be performed by bit shifts; and arithmetic on values up to u should fit within a single machine word. It is also beneficial for t to divide evenly into l, as we shall see shortly.

Encoding with bounded arithmetic. The encoding step acts on the window in the accumulator using estep_3, which risks making it overflow the range; we therefore renormalize with enorm_5 by shifting digits from the window to the remainder until this overflow would no longer happen, before consuming the symbol.

$$
\begin{array}{l}
\mathit{econsume}_5 :: [\mathit{Symbol}] \to \mathit{Number} \\
\mathit{econsume}_5 = \mathit{foldr}\ \mathit{estep}_5\ (l, [\,]) \\
\\
\mathit{estep}_5 :: \mathit{Symbol} \to \mathit{Number} \to \mathit{Number}
\end{array}
$$

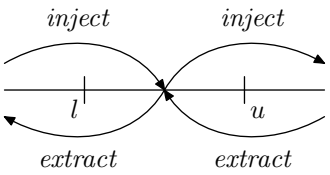


Fig. 4. 'Can't miss it' properties of the range: injecting an extra digit can only land within the range $[l, u)$ when starting below it, and will land above the range when starting within it; and conversely, extracting a digit can only land within the range when starting above it, and will land below the range when starting within it.

$$
\begin{aligned}
&\mathit{estep}_5\ s\ (w, ys) = \mathbf{let}\ (w', ys') = \mathit{enorm}_5\ s\ (w, ys)\ \mathbf{in}\ (\mathit{estep}_3\ s\ w', ys') \\
&\mathit{enorm}_5 :: \mathit{Symbol} \to \mathit{Number} \to \mathit{Number} \\
&\mathit{enorm}_5\ s\ (w, ys) = \mathbf{if}\quad \mathit{estep}_3\ s\ w < u \\
&\qquad\qquad\qquad\quad \mathbf{then}\ (w, ys) \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathbf{let}\ (q, r) = \mathit{extract}\ w\ \mathbf{in}\ \mathit{enorm}_5\ s\ (q, r : ys)
\end{aligned}
$$

That is, enorm_5 preserves the abstract value of a *Number*:

$$\mathit{abstract} \cdot \mathit{enorm}_5 = \mathit{abstract}$$

and leaves the window safe for estep_3 to incorporate the next symbol.

Note that if t divides l, then we can rearrange the guard in enorm_5:

$$
\begin{aligned}
&\quad \mathit{estep}_3\ s\ w < u \\
&\Leftrightarrow \quad \{\ \mathit{estep}_3;\ \text{let}\ (q, r) = w\ \text{divMod}\ \mathit{count}\ s\ \} \\
&\quad q \times t + \mathit{cumul}\ s + r < u \\
&\Leftrightarrow \quad \{\ t\ \text{divides}\ l,\ \text{so}\ u = (u\ \text{div}\ t) \times t\ \} \\
&\quad q \times t + \mathit{cumul}\ s + r < (u\ \text{div}\ t) \times t \\
&\Leftrightarrow \quad \{\ \text{universal property of division: } u < v \times w \Leftrightarrow u\ \text{div}\ w < v\ \} \\
&\quad (q \times t + \mathit{cumul}\ s + r)\ \text{div}\ t < u\ \text{div}\ t \\
&\Leftrightarrow \quad \{\ 0 \leqslant r < \mathit{count}\ s,\ \text{so}\ 0 \leqslant \mathit{cumul}\ s + r < t\ \} \\
&\quad q < u\ \text{div}\ t \\
&\Leftrightarrow \quad \{\ q = w\ \text{div}\ \mathit{count}\ s\ \} \\
&\quad w\ \text{div}\ \mathit{count}\ s < u\ \text{div}\ t \\
&\Leftrightarrow \quad \{\ \text{universal property of } \mathit{div}\ \text{again}\ \} \\
&\quad w < (u\ \text{div}\ t) \times \mathit{count}\ s \\
&\Leftrightarrow \quad \{\ u = l \times b,\ t\ \text{divides}\ l\ \} \\
&\quad w < b \times (l\ \text{div}\ t) \times \mathit{count}\ s
\end{aligned}
$$

This is worthwhile because $b \times (l\ \text{div}\ t)$ is a constant, independent of s, so the comparison can be done with a single multiplication, whereas the definition of estep_3 involves a division by $\mathit{count}\ s$.

For example, consider again encoding the text `"abc"`, with $b = 10$ and $l = 100$. The process is again from right to left, with the accumulator starting at $(100, [\,])$. Consuming the `'c'` then the `'b'` proceeds as before, because the window does not overflow u:

$$
\begin{aligned}
&\mathit{estep}_5\ \texttt{'c'}\ (100, [\,]) = (205, [\,]) \\
&\mathit{estep}_5\ \texttt{'b'}\ (205, [\,]) = (683, [\,])
\end{aligned}
$$

Now directly consuming the `'a'` would make the window overflow, because $\mathit{estep}_3\ \texttt{'a'}\ 683 = 3411 \geqslant u$; so we must renormalize to $(68, [3])$ before consuming the `'a'`, leading to the final state $(340, [3])$:

$$
\begin{aligned}
&\mathit{enorm}_5\ \texttt{'a'}\ (683, [\,]) = (68, [3]) \\
&\mathit{estep}_5\ \texttt{'a'}\ (683, [\,]) \quad = (\mathit{estep}_3\ \texttt{'a'}\ 68, [3]) = (340, [3])
\end{aligned}
$$

Note that the move from arbitrary to fixed precision is not just a data refinement—it is not the case that $\mathit{econsume}_5\ xs$ computes some representation of $\mathit{encode}_4\ xs$.

For example, $encode_4$ `"abc"` $= 3411$, whereas $econsume_5$ `"abc"` $= (340, [3])$, which is not a representation of 3411. We have really sacrificed some effectiveness in encoding in return for the increased efficiency of fixed precision arithmetic.

Decoding with bounded arithmetic. Decoding is an unfold using the accumulator as state. We repeatedly output a symbol from the window; this may make the window underflow the range, in which case we renormalize if possible by injecting digits from the remainder (and if this is not possible, because there are no more digits to inject, it means that we have decoded the entire text).

$$
\begin{aligned}
&dproduce_5 :: Number \to [Symbol] \\
&dproduce_5 = unfoldr\ dstep_5
\end{aligned}
$$

$$
\begin{aligned}
&dstep_5 :: Number \to Maybe\ (Symbol, Number) \\
&dstep_5\ (w, ys) = \ \mathbf{let}\ Just\ (s, w') \quad = dstep_3\ w \\
&\qquad\qquad\qquad\quad\ (w'', ys'') \quad = dnorm_5\ (w', ys) \\
&\qquad\qquad\quad\ \mathbf{in\ if}\ w'' \geqslant l\ \mathbf{then}\ Just\ (s, (w'', ys''))\ \mathbf{else}\ Nothing
\end{aligned}
$$

$$
\begin{aligned}
&dnorm_5 :: Number \to Number \\
&dnorm_5\ (w, y : ys) = \mathbf{if}\ w < l\ \mathbf{then}\ dnorm_5\ (inject\ w\ y, ys)\ \mathbf{else}\ (w, y : ys) \\
&dnorm_5\ (w, [\,]) \quad\ = (w, [\,])
\end{aligned}
$$

Note that decoding is of course symmetric to encoding; in particular, when encoding we renormalize before consuming a symbol; therefore when decoding we renormalize *after* emitting a symbol. For example, decoding the final encoding $(340, [3])$ starts by computing $dstep_3\ 340 = Just\ (\texttt{'a'}, 68)$; the window value 68 has underflowed, so renormalization consumes the remaining digit 3, restoring the accumulator to $(683, [\,])$; then decoding proceeds to extract the `'b'` and `'c'` in turn, returning the accumulator to $(100, [\,])$ via precisely the same states as for encoding, only in reverse order.

$$
\begin{aligned}
&dstep_5\ (340, [3]) = Just\ (\texttt{'a'}, (683, [\,])) \\
&dstep_5\ (683, [\,]) \ = Just\ (\texttt{'b'}, (205, [\,])) \\
&dstep_5\ (205, [\,]) \ = Just\ (\texttt{'c'}, (100, [\,])) \\
&dstep_5\ (100, [\,]) \ = Nothing
\end{aligned}
$$

Correctness of decoding. We can prove that decoding inverts encoding, again using the Unfoldr–Foldr Theorem with Invariant. Here, the invariant p is that the window w is in range $(l \leqslant w < u)$, which is indeed maintained by the consumer $estep_5$ and producer $dstep_5$. We show this in two steps. For $estep_5$, we start from a state (w, ys) with $l \leqslant w < u$. Let $(w', ys') = enorm_5\ s\ (w, ys)$. By construction of $enorm_5$, we have $estep_3\ s\ w' < u$, so we need only check the lower bound. If $enorm_5$ succeeds immediately, we have $w' = w \geqslant l$ as required. Otherwise, consider the last iteration, which proceeds from a penultimate state (w'', ys'') with window w'' such that $estep_3\ s\ w'' \geqslant u$ to the final state with window $w' = w''$ div b such that $estep_3\ s\ w' < u$; then we have

$$
\begin{aligned}
&l \leqslant estep_3\ s\ w' \\
\Leftrightarrow\quad &\{\ estep_3\ \}
\end{aligned}
$$

$$
\begin{array}{ll}
 & l \leqslant (w' \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s + (w' \bmod \mathit{count}\ s) \\
\Leftrightarrow & \quad \{\ t \text{ divides } l, \text{ so } l = (l \operatorname{div} t) \times t; \text{ division }\ \} \\
 & l \operatorname{div} t \leqslant ((w' \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s + (w' \bmod \mathit{count}\ s)) \operatorname{div} t \\
\Leftrightarrow & \quad \{\ 0 \leqslant \mathit{cumul}\ s + (w' \bmod \mathit{count}\ s) < t\ \} \\
 & l \operatorname{div} t \leqslant w' \operatorname{div} \mathit{count}\ s \\
\Leftrightarrow & \quad \{\ w' = w'' \operatorname{div} b; \text{ division, twice }\ \} \\
 & ((l \operatorname{div} t) \times \mathit{count}\ s) \times b \leqslant w'' \\
\Leftrightarrow & \quad \{\ t \text{ divides } l; \text{ multiplication is associative }\ \} \\
 & ((l \times b) \operatorname{div} t) \times \mathit{count}\ s \leqslant w'' \\
\Leftrightarrow & \quad \{\ \text{division; } u = l \times b\ \} \\
 & u \operatorname{div} t \leqslant w'' \operatorname{div} \mathit{count}\ s \\
\Leftrightarrow & \quad \{\ 0 \leqslant \mathit{cumul}\ s + (w'' \bmod \mathit{count}\ s) < t\ \} \\
 & u \operatorname{div} t \leqslant ((w'' \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s + (w'' \bmod \mathit{count}\ s)) \operatorname{div} t \\
\Leftrightarrow & \quad \{\ t \text{ divides } u; \text{ division }\ \} \\
 & u \leqslant (w'' \operatorname{div} \mathit{count}\ s) \times t + \mathit{cumul}\ s + (w'' \bmod \mathit{count}\ s) \\
\Leftrightarrow & \quad \{\ \mathit{estep}_3\ \} \\
 & u \leqslant \mathit{estep}_3\ s\ w'' \\
\Leftrightarrow & \quad \{\ \text{assumption }\ \} \\
 & \mathit{True}
\end{array}
$$

Therefore estep_5 does indeed maintain the window in range. As for dstep_5, note that by construction it can only return a window value $w'' \geqslant l$, so we need only check the upper bound. We start in a state (w, ys) such that $l \leqslant w < u$, then compute $w' < w$ by dstep_3 and tidy up with dnorm_5. If this succeeds immediately, we have $w'' = w' < w < u$ as required. Otherwise, consider the last iteration, which proceeds from a penultimate state w such that $w < l$ to a final state with window $w'' = w \times b + y$ (for some y with $0 \leqslant y < b$) such that $l \leqslant w''$; then we have

$$
\begin{array}{ll}
 & w'' < u \\
\Leftrightarrow & \quad \{\ w''\ \} \\
 & w \times b + y < u \\
\Leftrightarrow & \quad \{\ \text{unique representation property of range }\ \} \\
 & w < l \\
\Leftrightarrow & \quad \{\ \text{assumption }\ \} \\
 & \mathit{True}
\end{array}
$$

Therefore dstep_5 also maintains the window in range, so this indeed an invariant.

As for the conditions of the Unfoldr–Foldr Theorem: in the base case, $\mathit{dstep}_3\ l = \mathit{Just}\ (s, w')$ with $w' < l$, and $\mathit{dnorm}_5\ (w', [\,]) = (w', [\,])$, so indeed

$$\mathit{dstep}_5\ (l, [\,]) = \mathit{Nothing}$$

For the inductive step, suppose that $l \leqslant w < u$; then we have:

$$
\begin{array}{ll}
 & \mathit{dstep}_5\ (\mathit{estep}_5\ s\ (w, ys)) \\
= & \quad \{\ \mathit{estep}_5; \text{ let } (w', ys') = \mathit{enorm}_5\ s\ (w, ys)\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & dstep_5\ (estep_3\ s\ w', ys') \\
= & \quad \{\ dstep_5, dstep_3;\ \text{let}\ (w'', ys'') = dnorm_5\ (w', ys')\ \} \\
 & \mathbf{if}\ w'' \geqslant l\ \mathbf{then}\ Just\ (s, (w'', ys''))\ \mathbf{else}\ Nothing \\
= & \quad \{\ \text{see below: } dnorm_5\ \text{inverts } enorm_5\ s, \text{ so } (w'', ys'') = (w, ys)\ \} \\
 & \mathbf{if}\ w \geqslant l\ \mathbf{then}\ Just\ (s, (w, ys))\ \mathbf{else}\ Nothing \\
= & \quad \{\ \text{invariant holds, so in particular } w \geqslant l\ \} \\
 & Just\ (s, (w, ys))
\end{array}
$$

The remaining proof obligation is to show that

$$dnorm_5\ (enorm_5\ s\ (w, ys)) = (w, ys)$$

when $l \leqslant w < u$. We prove this in several steps. First, note that $dnorm_5$ is idempotent:

$$dnorm_5 \cdot dnorm_5 = dnorm_5$$

Second, when $l \leqslant w$ holds,

$$dnorm_5\ (w, ys) = (w, ys)$$

Finally, the key lemma is that, for $w < u$ (but not necessarily $w \geqslant l$), $dnorm_5$ is invariant under $enorm_5$:

$$dnorm_5\ (enorm_5\ s\ (w, ys)) = dnorm_5\ (w, ys)$$

When additionally $w \geqslant l$, the second property allows us to conclude that $dnorm_5$ inverts $enorm_5$:

$$dnorm_5\ (enorm_5\ s\ (w, ys)) = (w, ys)$$

The 'key lemma' is proved by induction on w. For $w = 0$, we clearly have

$$
\begin{array}{ll}
 & dnorm_5\ (enorm_5\ s\ (w, ys)) \\
= & \quad \{\ estep_3\ s\ 0 = cumul\ s \leqslant t \leqslant l, \text{ so } enorm_5\ s\ (w, ys) = (w, ys)\ \} \\
 & dnorm_5\ (w, ys)
\end{array}
$$

For the inductive step, we suppose that the result holds for all $q < w$, and consider two cases for w itself. In case $estep_3\ s\ w < u$, we have:

$$
\begin{array}{ll}
 & dnorm_5\ (enorm_5\ s\ (w, ys)) \\
= & \quad \{\ \text{assumption; } enorm_5\ \} \\
 & dnorm_5\ (w, ys)
\end{array}
$$

as required. And in case $estep_3\ s\ w \geqslant u$, we have:

$$
\begin{array}{ll}
 & dnorm_5\ (enorm_5\ s\ (w, ys)) \\
= & \quad \{\ \text{assumption; } enorm_5; \text{ let } (q, r) = extract\ w\ \} \\
 & dnorm_5\ (enorm_5\ s\ (q, r : ys)) \\
= & \quad \{\ q < w; \text{ induction}\ \} \\
 & dnorm_5\ (q, r : ys) \\
= & \quad \{\ w < u, \text{ so } q = w \operatorname{div} b < l;\ dnorm_5\ \} \\
 & dnorm_5\ (inject\ q\ r, ys)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\ q, r\ \} \\
& \mathit{dnorm}_5\ (w, \mathit{ys})
\end{array}
$$

Note that we made essential use of the limits of the range: $w < u \Rightarrow w \operatorname{div} b < l$.

Therefore decoding inverts encoding:

$$\mathit{dproduce}_5\ (\mathit{econsume}_5\ \mathit{xs}) = \mathit{xs}$$

for all finite xs.

7 Streaming

The version of encoding in the previous section yields a *Number*, that is, a pair consisting of an integer window and a digit-sequence remainder. It would be more conventional for encoding to take a sequence of symbols to a sequence of digits alone, and decoding to take the sequence of digits back to a sequence of symbols. For encoding, we have to flush the remaining digits out of the window at the end of the process, reducing the window to zero:

$$
\begin{array}{l}
\mathit{eflush}_5 :: \mathit{Number} \to [\mathit{Int}] \\
\mathit{eflush}_5\ (0, \mathit{ys}) \ = \mathit{ys} \\
\mathit{eflush}_5\ (w, \mathit{ys}) = \textbf{let}\ (w', y) = \mathit{extract}\ w\ \textbf{in}\ \mathit{eflush}_5\ (w', y : \mathit{ys})
\end{array}
$$

For example, $\mathit{eflush}_5\ (340, [3]) = [3, 4, 0, 3]$. Then we can define

$$
\begin{array}{l}
\mathit{encode}_5 :: [\mathit{Symbol}] \to [\mathit{Int}] \\
\mathit{encode}_5 = \mathit{eflush}_5 \cdot \mathit{econsume}_5
\end{array}
$$

Correspondingly, decoding should start by populating an initially-zero window from the sequence of digits:

$$
\begin{array}{l}
\mathit{dstart}_5 :: [\mathit{Int}] \to \mathit{Number} \\
\mathit{dstart}_5\ \mathit{ys} = \mathit{dnorm}_5\ (0, \mathit{ys})
\end{array}
$$

For example, $\mathit{dstart}_5\ [3, 4, 0, 3] = (340, [3])$. Then we can define

$$
\begin{array}{l}
\mathit{decode}_5 :: [\mathit{Int}] \to [\mathit{Symbol}] \\
\mathit{decode}_5 = \mathit{dproduce}_5 \cdot \mathit{dstart}_5
\end{array}
$$

Now, dstart_5 inverts eflush_5 on in-range values:

$$\mathit{dstart}_5\ (\mathit{eflush}_5\ (w, \mathit{ys})) = (w, \mathit{ys}) \quad \Leftarrow \quad l \leqslant w < u$$

To prove this, since we have already shown in Section 6 that dnorm_5 is idempotent, and has no effect on an in-range window, it suffices to show that dnorm_5 is invariant under shifting a digit from the window to the remainder. That is, with $(q, r) = \mathit{extract}\ w$, we have:

$$
\begin{array}{ll}
& \mathit{dnorm}_5\ (q, r : \mathit{ys}) \\
= & \{\ l \leqslant w < u, \text{ so } q = w \operatorname{div} b < l\ \} \\
& \mathit{dnorm}_5\ (\mathit{inject}\ q\ r, \mathit{ys})
\end{array}
$$

$$\begin{array}{ll} = & \{\ q, r\ \} \\ & dnorm_5\ (w, ys) \end{array}$$

Therefore again decoding inverts encoding:

$$\begin{array}{ll} & decode_5\ (encode_5\ xs) \\ = & \{\ decode_5, encode_5\ \} \\ & dproduce_5\ (dstart_5\ (eflush_5\ (econsume_5\ xs))) \\ = & \{\ econsume_5 \text{ yields in-range, on which } dstart_5 \text{ inverts } eflush_5\ \} \\ & dproduce_5\ (econsume_5\ xs) \\ = & \{\ dproduce_5 \text{ inverts } econsume_5\ \} \\ & xs \end{array}$$

for all finite xs.

7a Encoding as a metamorphism

We would now like to stream encoding and decoding as metamorphisms: encoding should start delivering digits before consuming all the symbols, and conversely decoding should start delivering symbols before consuming all the digits.

For encoding, we have

$$\begin{array}{l} encode_5 :: [Symbol] \to [Int] \\ encode_5 = eflush_5 \cdot econsume_5 \end{array}$$

with $econsume_5$ a fold; can we turn $eflush_5$ into an unfold? Yes, we can! The remainder in the accumulator should act as a queue: digits get enqueued at the most significant end, so we need to dequeue them from the least significant end. So we define

$$\begin{array}{l} edeal_6 :: [\alpha] \to [\alpha] \\ edeal_6 = unfoldr\ unsnoc\ \textbf{where} \\ \quad unsnoc\ [\,] = Nothing \\ \quad unsnoc\ ys = \textbf{let}\ (ys', y) = (init\ ys, last\ ys)\ \textbf{in}\ Just\ (y, ys') \end{array}$$

to "deal out" the elements of a list one by one, starting with the last. Of course, this reverses the list; so we have the slightly awkward

$$eflush_5 = reverse \cdot edeal_6 \cdot eflush_5$$

Flushing as an unfold. Now we can use unfold fusion to fuse $edeal_6$ and $eflush_5$ into a single unfold, deriving a coalgebra

$$efstep_6 :: Number \to Maybe\ (Int, Number)$$

such that

$$unsnoc\ (eflush_5\ (w, ys)) = fmap_{List}\ eflush_5\ (efstep_6\ (w, ys))$$

In case $w = 0$ and $ys = [\,]$, we have:

$$\begin{array}{ll} & \mathit{unsnoc}\ (\mathit{eflush}_5\ (0, [\,])) \\ = & \quad \{\ \mathit{eflush}_5\ \} \\ & \mathit{unsnoc}\ [\,] \\ = & \quad \{\ \mathit{unsnoc}\ \} \\ & \mathit{Nothing} \\ = & \quad \{\ \mathit{fmap}_{\mathit{List}}\ \} \\ & \mathit{fmap}_{\mathit{List}}\ \mathit{eflush}_5\ \mathit{Nothing} \end{array}$$

so we define

$$\mathit{efstep}_6\ (0, [\,]) = \mathit{Nothing}$$

In case $w > 0$ and $ys = [\,]$, we have:

$$\begin{array}{ll} & \mathit{unsnoc}\ (\mathit{eflush}_5\ (w, [\,])) \\ = & \quad \{\ \mathit{eflush}_5; \text{ let } (q, r) = \mathit{extract}\ w\ \} \\ & \mathit{unsnoc}\ (\mathit{eflush}_5\ (q, [r])) \\ = & \quad \{\ \text{lemma: } \mathit{eflush}_5\ (w, ys +\!\!+ zs) = \mathit{eflush}_5\ (w, ys) +\!\!+ zs\ \} \\ & \mathit{unsnoc}\ (\mathit{eflush}_5\ (q, [\,]) +\!\!+ [r]) \\ = & \quad \{\ \mathit{unsnoc}\ \} \\ & \mathit{Just}\ (r, \mathit{eflush}_5\ (q, [\,])) \\ = & \quad \{\ \mathit{fmap}_{\mathit{List}}\ \} \\ & \mathit{fmap}_{\mathit{List}}\ \mathit{eflush}_5\ (\mathit{Just}\ (r, (q, [\,]))) \end{array}$$

so we define

$$\mathit{efstep}_6\ (w, [\,]) = \mathbf{let}\ (q, r) = \mathit{extract}\ w\ \mathbf{in}\ \mathit{Just}\ (r, (q, [\,]))$$

An output queue. The lemma in the hint above states that already-enqueued digits zs in the remainder do not influence the computation of eflush_5 and simply pass straight through. It is discharged by induction on w: for the base case $w = 0$ we have

$$\mathit{eflush}_5\ (0, ys +\!\!+ zs) = ys +\!\!+ zs = \mathit{eflush}_5\ (0, ys) +\!\!+ zs$$

simply by definition, and for $w > 0$ we assume the inductive hypothesis for all $w' < w$ and proceed:

$$\begin{array}{ll} & \mathit{eflush}_5\ (w, ys +\!\!+ zs) \\ = & \quad \{\ \mathit{eflush}_5; \text{ let } (w', y) = \mathit{extract}\ w\ \} \\ & \mathit{eflush}_5\ (w', y : ys +\!\!+ zs) \\ = & \quad \{\ \text{inductive hypothesis, since } w' < w\ \} \\ & \mathit{eflush}_5\ (w', y : ys) +\!\!+ zs \\ = & \quad \{\ \mathit{eflush}_5\ \} \\ & \mathit{eflush}_5\ (w, ys) +\!\!+ zs \end{array}$$

Finally, when $ys \neq [\,]$,

$$\begin{array}{ll} & \mathit{unsnoc}\ (\mathit{eflush}_5\ (w, ys)) \\ = & \quad \{\ \text{lemma again}\ \} \\ & \mathit{unsnoc}\ (\mathit{eflush}_5\ (w, [\,]) +\!\!+\ ys) \\ = & \quad \{\ \mathit{unsnoc}; \text{let}\ (ys', y) = (\mathit{init}\ ys, \mathit{last}\ ys)\ \} \\ & \mathit{Just}\ (y, \mathit{eflush}_5\ (w, [\,]) +\!\!+\ ys') \\ = & \quad \{\ \text{lemma again}\ \} \\ & \mathit{Just}\ (y, \mathit{eflush}_5\ (w, ys')) \\ = & \quad \{\ \mathit{fmap}_{\mathit{List}}\ \} \\ & \mathit{fmap}_{\mathit{List}}\ \mathit{eflush}_5\ (\mathit{Just}\ (y, (w, ys'))) \end{array}$$

so we define (for non-null ys)

$$\mathit{efstep}_6\ (w, ys) = \mathbf{let}\ (ys', y) = (\mathit{init}\ ys, \mathit{last}\ ys)\ \mathbf{in}\ \mathit{Just}\ (y, (w, ys'))$$

We have derived the coalgebra

$$\begin{array}{l} \mathit{efstep}_6 :: \mathit{Number} \to \mathit{Maybe}\ (\mathit{Int}, \mathit{Number}) \\ \mathit{efstep}_6\ (0, [\,]) \ \ = \mathit{Nothing} \\ \mathit{efstep}_6\ (w, [\,]) \ = \mathbf{let}\ (q, r) = \mathit{extract}\ w\ \mathbf{in}\ \mathit{Just}\ (r, (q, [\,])) \\ \mathit{efstep}_6\ (w, ys) = \mathbf{let}\ (ys', y) = (\mathit{init}\ ys, \mathit{last}\ ys)\ \mathbf{in}\ \mathit{Just}\ (y, (w, ys')) \end{array}$$

for which we have $\mathit{eflush}_5 = \mathit{eflush}_6$ where

$$\begin{array}{l} \mathit{eflush}_6 :: \mathit{Number} \to [\mathit{Int}] \\ \mathit{eflush}_6 = \mathit{reverse} \cdot \mathit{unfoldr}\ \mathit{efstep}_6 \end{array}$$

Streaming encoding. As for the input part, we have a *foldr* where we need a *foldl*. Happily, that is easy to achieve, by the Third Duality Theorem. So we have $\mathit{encode}_5 = \mathit{encode}_6$, where

$$\begin{array}{l} \mathit{encode}_6 :: [\mathit{Symbol}] \to [\mathit{Int}] \\ \mathit{encode}_6 = \mathit{reverse} \cdot \mathit{unfoldr}\ \mathit{efstep}_6 \cdot \mathit{foldl}\ (\mathit{flip}\ \mathit{estep}_5)\ (l, [\,]) \cdot \mathit{reverse} \end{array}$$

It turns out that the streaming condition does not hold for efstep_6 and $\mathit{flip}\ \mathit{estep}_5$. We can indeed stream digits early out of the remainder, which acts as a queue; for example, from the state $(654, [3, 2, 1])$, consuming an `'a'` does not affect the 1 we can produce:

$$\begin{array}{ll} \mathit{efstep}_6\ (654, [3, 2, 1]) & = \mathit{Just}\ (1, (654, [3, 2])) \\ \mathit{estep}_5\ \texttt{'a'}\ (654, [3, 2, 1]) & = (321, [4, 3, 2, 1]) \\ \mathit{efstep}_6\ (\mathit{estep}_5\ \texttt{'a'}\ (654, [3, 2, 1])) & = \mathit{Just}\ (1, (321, [4, 3, 2])) \end{array}$$

However, we cannot stream the last few digits out of the window:

$$\begin{array}{ll} \mathit{efstep}_6\ (123, [\,]) & = \mathit{Just}\ (3, (12, [\,])) \\ \mathit{estep}_5\ \texttt{'c'}\ (123, [\,]) & = (248, [\,]) \\ \mathit{efstep}_6\ (\mathit{estep}_5\ \texttt{'c'}\ (123, [\,])) & = \mathit{Just}\ (8, (24, [\,])) \end{array}$$

We have to resort to *flushing streaming*, which starts from an apomorphism rather than an unfold. We'll use the cautious coalgebra

$$\begin{aligned}&\mathit{efstep}_7 :: \mathit{Number} \to \mathit{Maybe}\ (\mathit{Int}, \mathit{Number})\\&\mathit{efstep}_7 = \mathit{guard}\ (\mathit{not} \cdot \mathit{null} \cdot \mathit{snd})\ \mathit{efstep}_6\end{aligned}$$

which carefully avoids the problematic case when the remainder is empty.

It is now straightforward to verify that the streaming condition holds for efstep_7 and $\mathit{flip}\ \mathit{estep}_5$: the only way efstep_7 can be productive is

$$\mathit{efstep}_7\ (w, ys +\!\!+ [y]) = \mathit{Just}\ (y, (w, ys))$$

and we have:

$$\begin{aligned}&\mathit{efstep}_7\ (\mathit{estep}_5\ s\ (w, ys +\!\!+ [y]))\\=\quad&\{\ \text{let } (w', ys') = \mathit{estep}_5\ s\ (w, ys);\ \text{lemma (see below)}\ \}\\&\mathit{efstep}_7\ (w', ys' +\!\!+ [y])\\=\quad&\{\ \mathit{efstep}_7\ \}\\&\mathit{Just}\ (y, (w', ys'))\\=\quad&\{\ w', ys'\ \}\\&\mathit{Just}\ (y, \mathit{estep}_5\ s\ (w, ys))\end{aligned}$$

An input queue. The lemma hinted at above is analogous to the one for eflush_5, stating that already-enqueued digits zs in the remainder do not influence the computation and simply pass straight through:

$$\mathit{estep}_5\ s\ (w, ys +\!\!+ zs) = \mathbf{let}\ (w', ys') = \mathit{estep}_5\ s\ (w, ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs)$$

and depends on a corresponding lemma for enorm_5:

$$\mathit{enorm}_5\ s\ (w, ys +\!\!+ zs) = \mathbf{let}\ (w', ys') = \mathit{enorm}_5\ s\ (w, ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs)$$

We prove the latter first, by induction on w. When $w = 0$, we have $\mathit{estep}_3\ s\ 0 = \mathit{cumul}\ s < u$, so neither use of enorm_5 has an effect and we have immediately that

$$\begin{aligned}&\mathit{enorm}_5\ s\ (w, ys +\!\!+ zs)\\=\quad&\{\ \mathit{enorm}_5\ \}\\&(w, ys +\!\!+ zs)\\=\quad&\{\ \mathit{enorm}_5\ \}\\&\mathbf{let}\ (w', ys') = \mathit{enorm}_5\ s\ (w, ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs)\end{aligned}$$

For $w > 0$, we assume the result holds for $w' < w$, and conduct a case analysis. When $\mathit{estep}_3\ s\ w < u$, again neither use of enorm_5 has an effect, and the result holds. Finally, when $\mathit{estep}_3\ s\ w \geqslant u$, we have:

$$\begin{aligned}&\mathit{enorm}_5\ s\ (w, ys +\!\!+ zs)\\=\quad&\{\ \mathit{enorm}_5;\ \text{case assumption}\ \}\\&\mathbf{let}\ (q, r) = \mathit{extract}\ w\ \mathbf{in}\ \mathit{enorm}_5\ s\ (q, r : ys +\!\!+ zs)\\=\quad&\{\ \text{induction hypothesis, since } q < w\ \}\\&\mathbf{let}\ (q, r) = \mathit{extract}\ w; (w', ys') = \mathit{enorm}_5\ s\ (q, r : ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs)\\=\quad&\{\ \text{swapping } \mathbf{let}\text{s}\ \}\\&\mathbf{let}\ (w', ys') = (\mathbf{let}\ (q, r) = \mathit{extract}\ w\ \mathbf{in}\ \mathit{enorm}_5\ s\ (q, r : ys))\ \mathbf{in}\ (w', ys' +\!\!+ zs)\end{aligned}$$

$$\begin{array}{ll} = & \{\ enorm_5\text{; case assumption}\ \} \\ & \mathbf{let}\ (w', ys') = enorm_5\ s\ (w, ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs) \end{array}$$

Then the lemma for $estep_5$ follows:

$$\begin{array}{ll} & estep_5\ s\ (w, ys +\!\!+ zs) \\ = & \{\ estep_5\ \} \\ & \mathbf{let}\ (w', ys') = enorm_5\ s\ (w, ys +\!\!+ zs)\ \mathbf{in}\ (estep_3\ s\ w', ys') \\ = & \{\ \text{lemma for } enorm_5\ \} \\ & \mathbf{let}\ (w'', ys'') = enorm_5\ s\ (w, ys); (w', ys') = (w'', ys'' +\!\!+ zs)\ \mathbf{in}\ (estep_3\ s\ w', ys') \\ = & \{\ \text{inlining}\ \} \\ & \mathbf{let}\ (w'', ys'') = enorm_5\ s\ (w, ys)\ \mathbf{in}\ (estep_3\ s\ w'', ys'' +\!\!+ zs) \\ = & \{\ \text{un-inlining}\ \} \\ & \mathbf{let}\ (w'', ys'') = enorm_5\ s\ (w, ys); (w', ys') = (estep_3\ s\ w'', ys'')\ \mathbf{in}\ (w', ys' +\!\!+ zs) \\ = & \{\ estep_5\ \} \\ & \mathbf{let}\ (w', ys') = estep_5\ s\ (w, ys)\ \mathbf{in}\ (w', ys' +\!\!+ zs) \end{array}$$

This at last discharges the lemma; therefore $encode_6 = encode_7$, where

$$\begin{array}{l} encode_7 :: [Symbol] \to [Int] \\ encode_7 = reverse \cdot fstream\ efstep_7\ (unfoldr\ efstep_6)\ (flip\ estep_5)\ (l, [\,]) \cdot reverse \end{array}$$

and where $encode_7$ streams its output. On the downside, this has to read the input text in reverse, and also write the output digit sequence in reverse.

7b Decoding as a metamorphism

Fortunately, decoding is rather easier to stream. We have

$$decode_5 = dproduce_5 \cdot dstart_5$$

with $dproduce_5$ an unfold; can we turn $dstart_5$ into a fold? Yes, we can! In fact, we have $dstart_5 = foldl\ dsstep_8\ (0, [\,])$, where

$$\begin{array}{l} dsstep_8\ (w, [\,])\ y\ = \mathbf{if}\ w < l\ \mathbf{then}\ (inject\ w\ y, [\,])\ \mathbf{else}\ (w, [y]) \\ dsstep_8\ (w, ys)\ y = (w, ys +\!\!+ [y]) \end{array}$$

That is, starting with 0 for the accumulator, digits are injected one by one into the window until this is in range, and thereafter appended to the remainder.

Streaming decoding. Now we have decoding as an *unfoldr* after a *foldl*, and it is straightforward to verify that the streaming condition holds for $dstep_5$ and $dsstep_8$. Suppose (w, ys) is a productive state for $dstep_5$; that is, let

$$\begin{array}{ll} (q, r) & = w\ \text{divMod}\ t \\ s & = find\ r \\ w' & = count\ s \times q + r - cumul\ s \\ (w'', ys'') & = dnorm_5\ (w', ys) \end{array}$$

and suppose $w'' \geqslant l$. Note then that

$$dnorm_5\ (w', ys +\!\!+ [y]) = (w'', ys'' +\!\!+ [y])$$

Then the claim is:

$$dstep_5\ (dsstep_8\ (w, ys)\ y) = Just\ (s, dsstep_8\ (w'', ys'')\ y)$$

In case $ys = [\,]$, we have $(w'', ys'') = dnorm_5\ (w', ys) = (w', [\,])$, so $w > w' = w'' \geqslant l$, and

$$dsstep_8\ (w, ys)\ y = (w, ys +\!\!+ [y])$$

In case $ys \neq [\,]$, we also have

$$dsstep_8\ (w, ys)\ y = (w, ys +\!\!+ [y])$$

Either way, we now have:

$$\begin{array}{ll} & dstep_5\ (dsstep_8\ (w, ys)\ y) \\ = & \quad \{\ \text{case analysis above}\ \} \\ & dstep_5\ (w, ys +\!\!+ [y]) \\ = & \quad \{\ dnorm_5\ (w', ys +\!\!+ [y]) = (w'', ys'' +\!\!+ [y])\ \} \\ & Just\ (s, (w'', ys'' +\!\!+ [y])) \\ = & \quad \{\ w'' \geqslant l;\ dsstep_8\ \} \\ & Just\ (s, dsstep_8\ (w'', ys'')\ y) \end{array}$$

and the streaming condition holds. Therefore $decode_5 = decode_8$ on finite inputs, where

$$\begin{array}{l} decode_8 :: [Int] \to [Symbol] \\ decode_8 = stream\ dstep_5\ dsstep_8\ (0, [\,]) \end{array}$$

and $decode_8$ streams.

Summarizing metamorphisms. Inlining definitions and simplifying, we have ended up with encoding as a flushing stream, reading and writing backwards:

$$\begin{array}{l} encode_9 :: [Symbol] \to [Int] \\ encode_9 = reverse \cdot fstream\ g'\ (unfoldr\ g)\ f\ (l, [\,]) \cdot reverse\ \mathbf{where} \\ \quad f\ (w, ys)\ s = \mathbf{if} \quad w < b \times (l\ \mathrm{div}\ t) \times count\ s \\ \qquad\qquad\quad \mathbf{then\ let}\ (q, r) = w\ \mathrm{divMod}\ count\ s\ \mathbf{in}\ (q \times t + cumul\ s + r, ys) \\ \qquad\qquad\quad \mathbf{else\ \ let}\ (q, r) = extract\ w\ \mathbf{in}\ f\ (q, r : ys)\ s \\ \quad g\ (0, [\,]) \quad\ = Nothing \\ \quad g\ (w, [\,]) \quad\ = \mathbf{let}\ (q, r) = extract\ w\ \mathbf{in}\ Just\ (r, (q, [\,])) \\ \quad g\ (w, ys) \quad = \mathbf{let}\ (ys', y) = (init\ ys, last\ ys)\ \mathbf{in}\ Just\ (y, (w, ys')) \\ \quad g'\ (w, ys) \quad = \mathbf{if}\ null\ ys\ \mathbf{then}\ Nothing\ \mathbf{else}\ g\ (w, ys) \end{array}$$

and decoding as an ordinary stream:

$$\begin{array}{l} decode_9 :: [Int] \to [Symbol] \\ decode_9 = stream\ g\ f\ (0, [\,])\ \mathbf{where} \\ \quad g\ (w, ys) \quad = \mathbf{let}\ (q, r) = w\ \mathrm{divMod}\ t \\ \qquad\qquad\quad\ \ s = find\ r \end{array}$$

$$\begin{array}{lll} & & (w'', ys'') = n\ (count\ s \times q + r - cumul\ s, ys) \\ & & \textbf{in if } w'' \geqslant l \textbf{ then } \quad Just\ (s, (w'', ys'')) \textbf{ else } Nothing \\ n\ (w, ys) & = & \textbf{if } w \geqslant l \vee null\ ys \textbf{ then } (w, ys) \textbf{ else} \\ & & \textbf{let } (y : ys') = ys \textbf{ in } n\ (inject\ w\ y, ys') \\ f\ (w, [\,])\ y & = & n\ (w, [y]) \\ f\ (w, ys)\ y & = & (w, ys +\!\!+ [y]) \end{array}$$

The remainder acts as a queue, with digits being added at one end and removed from the other, so should be represented to make that efficient. But in fact, the remainder can be eliminated altogether, yielding simply the window for the accumulator, as we shall see next.

7c Streaming via tail recursion

As we have just seen, it is possible—with some effort—to persuade the definitions of $encode_5$ and $decode_5$ into *metamorphism* form; however, that turns out to be rather complicated. We outline here a more direct route instead.

Streaming encoding. For encoding, we have

$$encode_5 = eflush_5 \cdot foldr\ estep_5\ (l, [\,])$$

A first step for streaming is to make as much of this as possible tail-recursive. The best we can do is to apply the Third Duality Theorem to transform the *foldr* into a *foldl*:

$$encode_5 = eflush_5 \cdot foldl\ (flip\ estep_5)\ (l, [\,]) \cdot reverse$$

Now we note that the remainder component of the *Number* behaves like a queue, in the sense that already-enqueued digits simply pass through without being further examined:

$$\begin{array}{lll} eflush_5\ (w, ys +\!\!+ zs) & = & eflush_5\ (w, ys) +\!\!+ zs \\ enorm_5\ s\ (w, ys +\!\!+ zs) & = & \textbf{let } (w', ys') = enorm_5\ s\ (w, ys) \textbf{ in } (w', ys' +\!\!+ zs) \\ estep_5\ s\ (w, ys +\!\!+ zs) & = & \textbf{let } (w', ys') = estep_5\ s\ (w, ys) \quad \textbf{in } (w', ys' +\!\!+ zs) \end{array}$$

(These lemmas were proved in Section 7a.) If we then introduce the auxilliary functions e_1, e_2 specified by

$$\begin{array}{lll} reverse\ (e_1\ w\ ss) +\!\!+ ys & = & eflush_5\ (foldl\ (flip\ estep_5)\ (w, ys)\ ss) \\ reverse\ (e_2\ w) +\!\!+ ys & = & eflush_5\ (w, ys) \end{array}$$

and unfold definitions, exploiting the queueing properties, we can synthesize $encode_5 = encode_{10}$, where:

$$\begin{array}{l} encode_{10} :: [Symbol] \to [Int] \\ encode_{10} = reverse \cdot e_1\ l \cdot reverse\ \textbf{where} \\ \quad e_1\ w\ (s : ss) = \textbf{let } (q, r) = w \text{ divMod } count\ s \\ \qquad\qquad\qquad\qquad\quad w' \quad = q \times t + cumul\ s + r\ \textbf{in} \end{array}$$

$$\begin{array}{l}
\qquad\qquad\quad \textbf{if } w' < u \textbf{ then } e_1\ w'\ ss \\
\qquad\qquad\qquad\quad \textbf{else let } (q', r') = w \text{ divMod } b \textbf{ in } r' : e_1\ q'\ (s : ss) \\
e_1\ w\ [\,] = e_2\ w \\
e_2\ w = \textbf{if } w == 0 \textbf{ then } [\,] \textbf{ else let } (w', y) = w \text{ divMod } b \textbf{ in } y : e_2\ w'
\end{array}$$

In this version, the accumulator w simply maintains the window, and digits in the remainder are output as soon as they are generated. Note that the two *reverse*s mean that encoding effectively reads its input and writes its output from right to left; that seems to be inherent to ANS.

Streaming decoding. Decoding is easier, because $dnorm_5$ is already tail-recursive. Similarly specifying functions d_0, d_1, d_2 by

$$\begin{array}{l}
d_0\ w\ ys \quad = dproduce_5\ (dnorm_5\ (w, ys)) \\
d_1\ w\ ys \quad = dproduce_5\ (w, ys) \\
d_2\ s\ w'\ ys = \textbf{let } (w'', ys'') = dnorm_5\ (w', ys) \textbf{ in} \\
\qquad\qquad \textbf{if } w'' \geqslant l \textbf{ then } s : d_1\ w''\ ys'' \textbf{ else } [\,]
\end{array}$$

and unfolding definitions allows us to synthesize directly that $decode_5 = decode_{10}$, where:

$$\begin{array}{l}
decode_{10} :: [Int] \to [Symbol] \\
decode_{10} = d_0\ 0 \textbf{ where} \\
\quad \begin{array}{lll}
d_0\ w\ (y : ys) \mid w < l & = d_0\ (w \times b + y)\ ys \\
d_0\ w\ ys & = d_1\ w\ ys \\
d_1\ w\ ys & = \textbf{let } (q, r) = w \text{ divMod } t \\
 & \qquad s \quad = find\ r \\
 & \qquad w' \quad = count\ s \times q + r - cumul\ s \\
 & \quad \textbf{in } d_2\ s\ w'\ ys \\
d_2\ s\ w\ (y : ys) \mid w < l & = d_2\ s\ (w \times b + y)\ ys \\
d_2\ s\ w\ ys \quad\ \mid w \geqslant l & = s : d_1\ w\ ys \\
d_2\ s\ w\ [\,] & = [\,]
\end{array}
\end{array}$$

Ignoring additions and subtractions, encoding involves one division by *count* s and one multiplication by t for each input symbol s, plus one division by b for each output digit. Conversely, decoding involves one multiplication by b for each input digit, plus one division by t and one multiplication by *count* s for each output symbol s. Encoding and decoding are both tail-recursive. The arithmetic in base b can be simplified to bit shifts by choosing b to be a power of two. They therefore correspond rather directly to simple imperative implementations [19].

8 Conclusion

We have presented a development using the techniques of constructive functional programming of the encoding and decoding algorithms involved in asymmetric numeral systems, including the step from arbitrary- to fixed-precision arithmetic and then to streaming processes. The calculational techniques depend on the

theory of folds and unfolds for lists, especially the duality between *foldr* and *foldl*, fusion, and the Unfoldr–Foldr Theorem. We started out with the hypothesis that the theory of streaming developed by the author together with Richard Bird for arithmetic coding [3, 15] would be a helpful tool; but although it can be applied, it seems here to be more trouble than it is worth.

To be precise, what we have described is the *range* variant (rANS) of ANS. There is also a *tabled* variant (tANS), used by Zstd [9] and LZFSE [10], which essentially tabulates the functions $estep_5$ and $dstep_5$; for encoding this involves a table of size $n \times (u - l)$, the product of the alphabet size and the window width, and for decoding two tables of size $u - l$. Tabulation makes even more explicit that HC is a special case of ANS, with the precomputed table corresponding to the Huffman trie. Tabulation also allows more flexibility in the precise allocation of codes, which slightly improves effectiveness [14]. For example, the coding table in Figure 3 corresponds to the particular arrangement `"aabbbccc"` of the three symbols in proportion to their counts, and lends itself to implementation via arithmetic; but any permutation of this arrangement would still work, and a permutation such as `"cbcacbcacb"` which distributes the symbols more evenly turns out to be slightly more effective and no more difficult to tabulate.

One nice feature of AC is that the switch from arbitrary-precision to fixed-precision arithmetic can be expressed in terms of a carefully chosen adaptive model, which slightly degrades the ideal distribution in order to land on convenient rational endpoints [27]. We do not have that luxury with ANS, because of the awkwardness of incorporating adaptive coding; consequently, it is not clear that there is any simple relationship between the arbitrary-precision and fixed-precision versions. But even with AC, that nice feature only applies to encoding; the approximate arithmetic seems to preclude a correctness argument in terms of the Unfoldr–Foldr Theorem, and therefore a completely different (and more complicated) approach is required for decoding [3].

The ANS algorithms themselves are of course not novel here; they are due to Duda [13, 14]. Our development in Section 5 of the key bit of arithmetic in ANS encoding was informed by a very helpful commentary on Duda's paper published in a series of twelve blog posts [6] by Charles Bloom. The illustration in Figure 3 derives from Duda, and was also used by Roman Cheplyaka [8] as the basis of a (clear but very inefficient) prototype Haskell implementation.

Acknowledgements: I am grateful to participants of IFIP WG2.1 Meeting 78 in Xitou and IFIP WG2.11 Meeting 19 in Salem, who gave helpful feedback on earlier presentations of this work, and to Jarek Duda and Richard Bird who gave much helpful advice on content and presentation. I thank the Programming Research Laboratory at the National Institute of Informatics in Tokyo, for providing a Visiting Professorship during which some of this research was done; in particular, Zhixuan Yang and Josh Ko of NII explained to me the significance of t dividing evenly into l, as exploited in Section 6.

References

1. Jyrki Alakuijala. Google's compression projects. `https://encode.ru/threads/3108-Google-s-compression-projects#post60072`, May 2019.
2. Richard Bird. Proof of equivalence between huffman trees and certain bags of fractions. Personal communication, April 2019.
3. Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming 4*, volume 2638 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2003.
4. Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.
5. Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.
6. Charles Bloom. Understanding ANS. `http://cbloomrants.blogspot.com/2014/01/1-30-14-understanding-ans-1.html`, January 2014.
7. Jamieson Brettle and Frank Galligan. Introducing Draco: Compression for 3D graphics. `https://opensource.googleblog.com/2017/01/introducing-draco-compression-for-3d.html`, January 2017.
8. Roman Cheplyaka. Understanding asymmetric numeral systems. `https://ro-che.info/articles/2017-08-20-understanding-ans`, August 2017.
9. Yann Collet and Chip Turner. Smaller and faster data compression with Zstandard. `https://code.fb.com/core-data/smaller-and-faster-data-compression-with-zstandard/`, August 2016.
10. Sergio De Simone. Apple open-sources its new compression algorithm LZFSE. *InfoQ*, July 2016. `https://www.infoq.com/news/2016/07/apple-lzfse-lossless-opensource`.
11. Jarosław Duda. Kodowanie i generowanie układówstatystycznych za pomocą algorytmów probabilistycznych (Coding and generation of statistical systems using probabilistic algorithms). Master's thesis, Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, 2006.
12. Jarosław Duda. Optimal encoding on discrete lattice with translational invariant constrains using statistical algorithms. *CoRR*, abs/0710.3861, 2007. English translation of [11].
13. Jarosław Duda. Asymmetric numeral systems. *CoRR*, 0902.0271v5, May 2009.
14. Jarosław Duda. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR*, 1311.2540v2, January 2014.
15. Jeremy Gibbons. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139, 2007.
16. Jeremy Gibbons. Coding with asymmetric numeral systems (Haskell code). `http://www.cs.ox.ac.uk/jeremy.gibbons/publications/asymm.hs`, July 2019.
17. Jeremy Gibbons. Coding with asymmetric numeral systems (long version). `http://www.cs.ox.ac.uk/jeremy.gibbons/publications/asymm-long.pdf`, July 2019.
18. Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, April/May 2005.

19. Fabien Giesen. Simple rANS encoder/decoder. `https://github.com/rygorous/ryg_rans`, February 2014.
20. David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.
21. Joint Photographic Experts Group. Overview of JPEG XL. `https://jpeg.org/jpegxl/`.
22. Timothy B. Lee. Inventor says Google is patenting work he put in the public domain. *Ars Technica*, October 2018. `https://arstechnica.com/tech-policy/2018/06/inventor-says-google-is-patenting-work-he-put-in-the-public-domain/`.
23. Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
24. Daniel Reiter Horn and Jongmin Baek. Building better compression together with DivANS. `https://blogs.dropbox.com/tech/2018/06/building-better-compression-together-with-divans/`, June 2018.
25. Jorma J. Rissanen and Glen G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, March 1979.
26. Julian Seward. `bzip2`. `https://en.wikipedia.org/wiki/Bzip2`, 1996.
27. Barney Stratford. *A Formal Treatment of Lossless Data Compression Algorithms*. DPhil thesis, University of Oxford, 2005.
28. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.
29. Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.