Streaming Data Compression

Jeremy Gibbons

1. Metamorphisms

Tail-recursive (accumulative) list consumption:

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

Coinductive list production:

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

A *metamorphism* is their composition:

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

2. Examples of metamorphisms

$$
\begin{aligned}
\mathit{regroup}\ n &= \mathit{group}\ n \circ \mathit{concat} \\
\mathit{heapsort} &= \mathit{flattenHeap} \circ \mathit{buildHeap} \\
\mathit{baseconv}\ (b, c) &= \mathit{toBase}\ b \circ \mathit{fromBase}\ c \\
\mathit{arithCode} &= \mathit{toBits} \circ \mathit{narrow}
\end{aligned}
$$

3. Streaming

Interleaving production and consumption:

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

Streaming condition for g and f:

$$g\ z = \mathit{Just}\ (y, z') \Longrightarrow \forall x\ .\ g\ (f\ z\ x) = \mathit{Just}\ (y, f\ z'\ x)$$

Theorem: if the streaming condition holds for g and f, then for all finite xs

$$\mathit{stream}\ g\ f\ z\ \mathit{xs} = \mathit{unfoldr}\ g\ (\mathit{foldl}\ f\ z\ \mathit{xs})$$

Moreover, *stream* can be productive on infinite inputs.

4. Example of streaming

Buffering process $\mathit{unfoldr}\ \mathit{uncons} \circ \mathit{foldl}\ (+\!\!+)\ [\,]$, where

$$
\begin{aligned}
\mathit{uncons}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ & \\
x : \mathit{xs}' &\rightarrow \mathit{Just}\ (x, \mathit{xs}') \\
[\,] &\rightarrow \mathit{Nothing}
\end{aligned}
$$

Since $\mathit{unfoldr}\ \mathit{uncons} = \mathit{id}$, buffering is just concat.

The streaming condition holds for uncons and $+\!\!+$, so concat can be streamed.

5. A non-example

concat is special, because production can always exhaust the internal state.

In contrast, consider $regroup\ n = unfoldr\ (chunk\ n) \circ foldl\ (+\!\!+)\ [\,]$, where

$$
\begin{aligned}
&chunk\ n\ [\,] = Nothing \\
&chunk\ n\ xs = Just\ (splitAt\ n\ xs)
\end{aligned}
$$

Streaming condition fails: *chunk* is too aggressive, and may produce short chunks when there is still remaining input.

Try more cautious producer $chunk'$:

$$
\begin{aligned}
chunk'\ n\ xs \mid & n \leqslant length\ xs &= Just\ (splitAt\ n\ xs) \\
\mid & otherwise &= Nothing
\end{aligned}
$$

But this *never* produces a short chunk.

Process should remain cautious while input remains, then throw caution to the winds.

6. Flushing

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

Theorem: if the streaming condition holds for g and f, then for all finite xs

$$
\mathit{fstream}\ g\ h\ f\ z\ \mathit{xs} = \mathit{apo}\ g\ h\ (\mathit{foldl}\ f\ z\ \mathit{xs})
$$

where

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

In particular, $\mathit{regroup}\ n = \mathit{fstream}\ (\mathit{chunk}'\ \ n)\ (\mathit{unfoldr}\ (\mathit{chunk}\ n))\ (+\!\!+)\ [\,]$.

7. Change of base

Consider conversion from base 3 to base 7:

$$
\begin{aligned}
\mathit{fromBase3} &= \mathit{foldr}\ \mathit{stepr}\ 0\ \mathbf{where}\ \mathit{stepr}\ d\ x = (d + x)\ /\ 3 \\
\mathit{toBase7} &= \mathit{unfoldr}\ \mathit{next}\ \mathbf{where}\ \mathit{next}\ 0 = \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\quad \mathit{next}\ x = \mathbf{let}\ y = 7 \times x\ \mathbf{in}\ \mathit{Just}\ (\lfloor y \rfloor, y - \lfloor y \rfloor)
\end{aligned}
$$

(assume input digits are all in range: $0, 1, 2$). Wrong kind of fold; but

$$
\mathit{fromBase3} = \mathit{extract} \circ \mathit{foldl}\ \mathit{stepl}\ (0, 1)\ \mathbf{where}\ \mathit{stepl}\ (u, v)\ d = (u \times 3 + d, v\ /\ 3)
$$

where $\mathit{extract}\ (u, v) = \mathit{apply}\ (u, v)\ 0$ where $\mathit{apply}\ (u, v)\ x = v \times (u + x)$.

Now the $\mathit{extract}$ is an obstacle; but $\mathit{toBase7} \circ \mathit{extract} = \mathit{unfoldr}\ \mathit{next}'$ where

$$
\begin{aligned}
\mathit{next}'\ (0, v) &= \mathit{Nothing} \\
\mathit{next}'\ (u, v) &= \mathbf{let}\ y = \lfloor 7 \times u \times v \rfloor\ \mathbf{in}\ \mathit{Just}\ (y, (u - y\ /\ (v \times 7), v \times 7))
\end{aligned}
$$

So we have a metamorphism.

8. Streaming change of base

Streaming condition does not hold for *stepl* and *next'*. Eg:

$$\begin{aligned} &\mathit{next}'\ (1, {}^{1}\!/_{3}) && = \mathit{Just}\ (2, ({}^{1}\!/_{7}, {}^{7}\!/_{3})) \\ &\mathit{next}'\ (1, \mathit{stepl}\ (1, {}^{1}\!/_{3})\ 1) = \mathit{next}'\ (4, {}^{1}\!/_{9}) && = \mathit{Just}\ (3, ({}^{1}\!/_{7}, {}^{7}\!/_{9})) \end{aligned}$$

ie $0.1_3 \simeq 0.222_7$ but $0.11_3 \simeq 0.305_7$: *next'* is too aggressive.

Remaining input in unit interval; so possible outputs range from $\mathit{apply}\ (u, v)\ 0$ to $\mathit{apply}\ (u, v)\ 1$. Safe to commit iff these start with same digit in base 7:

$$\mathit{next}''\ (u, v) = \mathbf{if}\ \lfloor u \times v \times 7 \rfloor == \lfloor (u + 1) \times v \times 7 \rfloor\ \mathbf{then}\ \mathit{next}'\ (u, v)\ \mathbf{else}\ \mathit{Nothing}$$

Then streaming condition holds for *stepl* and *next''*, and

$$\begin{aligned} \mathit{toBase7}\ (\mathit{fromBase3}\ \mathit{xs}) &= \mathit{apo}\ \mathit{next}''\ (\mathit{unfoldr}\ \mathit{next}')\ (\mathit{foldl}\ \mathit{stepl}\ (0, 1)\ \mathit{xs}) \\ &= \mathit{fstream}\ \mathit{next}''\ (\mathit{unfoldr}\ \mathit{next}')\ \mathit{stepl}\ (0, 1)\ \mathit{xs} \end{aligned}$$

for finite base 3 digit sequences *xs*. Moreover, also works on (most) infinite *xs*.

9. Coding

- *Huffman coding* (HC)

  efficient; optimally effective for bit-sequence-per-symbol

- *arithmetic coding* (AC)

  Shannon-optimal (fractional entropy); but inefficient

- *asymmetric numeral systems* (ANS)

  efficiency of Huffman, effectiveness of arithmetic coding

- applications of *streaming*

ANS introduced by Jarek Duda (2013).

Used by Facebook (Zstandard), Apple (LZFSE), Google (Draco), Dropbox (DivANS)...

10. Intervals

Pairs of rationals

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

with operations

$$\begin{aligned}
&\mathit{unit} &&= (0, 1) \\
&\mathit{weight}\ (l, r)\ x &&= l + (r - l) \times x \\
&\mathit{narrow}\ i\ (p, q) &&= (\mathit{weight}\ i\ p, \mathit{weight}\ i\ q) \\
&\mathit{scale}\ (l, r)\ x &&= (x - l)\ /\ (r - l) \\
&\mathit{widen}\ i\ (p, q) &&= (\mathit{scale}\ i\ p, \mathit{scale}\ i\ q)
\end{aligned}$$

so that

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

and $\mathit{widen}\ i\ (\mathit{narrow}\ i\ j) = j$. Also, narrow and unit form a monoid.

11. Models

Given

$$counts :: [\,(Symbol, Integer)\,]$$

get

$$encodeSym :: Symbol \to Interval$$
$$decodeSym :: Rational \to Symbol$$

such that

$$decodeSym\ x = s \iff x \in encodeSym\ s$$

Eg alphabet $\{\text{'a'}, \text{'b'}, \text{'c'}\}$ with counts $2, 3, 5$ encoded as $(0, {}^1/_5)$, $({}^1/_5, {}^1/_2)$, and $({}^1/_2, 1)$.

12. Arithmetic coding

$$
\begin{aligned}
&\mathit{encode}_1 :: [\mathit{Symbol}] \to \mathit{Rational}\\
&\mathit{encode}_1 \;=\; \mathit{pick} \circ \mathit{foldl}\ \mathit{estep}_1\ \mathit{unit}\ \textbf{where}\\
&\quad \mathit{estep}_1 :: \mathit{Interval} \to \mathit{Symbol} \to \mathit{Interval}\\
&\quad \mathit{estep}_1\ i\ s = \mathit{narrow}\ i\ (\mathit{encodeSym}\ s)\\
&\mathit{decode}_1 :: \mathit{Rational} \to [\mathit{Symbol}]\\
&\mathit{decode}_1 \;=\; \mathit{unfoldr}\ \mathit{dstep}_1\ \textbf{where}\\
&\quad \mathit{dstep}_1 :: \mathit{Rational} \to \mathit{Maybe}\ (\mathit{Symbol}, \mathit{Rational})\\
&\quad \mathit{dstep}_1\ x = \textbf{let}\ s = \mathit{decodeSym}\ x\ \textbf{in}\ \mathit{Just}\ (s, \mathit{scale}\ (\mathit{encodeSym}\ s)\ x)
\end{aligned}
$$

where $\mathit{pick} :: \mathit{Interval} \to \mathit{Rational}$ satisfies $\mathit{pick}\ i \in i$. Eg, with $\mathit{pick} = \mathit{fst}$:

$$
(0, 1) \xrightarrow{\text{'a'}} (0, {}^{1}\!/\!{}_{5}) \xrightarrow{\text{'b'}} ({}^{1}\!/\!{}_{25}, {}^{1}\!/\!{}_{10}) \xrightarrow{\text{'c'}} ({}^{7}\!/\!{}_{100}, {}^{1}\!/\!{}_{10}) \quad \rightsquigarrow \quad {}^{7}\!/\!{}_{100}
$$

13. Trading in bits

Let $pick = fromBits \circ toBits$, where

$$\begin{aligned} &toBits :: Interval \to [Bool] \\ &fromBits :: [Bool] \to Rational \end{aligned}$$

Obvious thing: let $toBits\ i$ pick shortest binary fraction in i, and $fromBits$ evaluate this fraction. But doesn't satisfy streaming.

Instead: $toBits\ i$ yields bit sequence bs such that $bs +\!\!+ [True]$ is shortest.

$$\begin{aligned} &toBits = unfoldr\ nextBit\ \mathbf{where} \\ &\quad nextBit\ (l, r) \mid r \leqslant {}^{1}/_{2} = Just\ (False, (0, {}^{1}/_{2})\ \text{`}widen\text{`}\ (l, r)) \\ &\quad\qquad\qquad\quad \mid {}^{1}/_{2} \leqslant l = Just\ (True, ({}^{1}/_{2}, 1)\ \text{`}widen\text{`}\ (l, r)) \\ &\quad\qquad\qquad\quad \mid otherwise = Nothing \end{aligned}$$

$$fromBits = foldr\ pack\ ({}^{1}/_{2})\ \mathbf{where}\ pack\ b\ x = ((\mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 0) + x)\ /\ 2$$

Now $pick$ is a hylomorphism. Also, $toBits$ yields a finite sequence.

14. Streaming encoding

Move *fromBits* from encoding to decoding:

$$
\begin{aligned}
&\mathit{encode}_{\mathit{Bits}} :: [\mathit{Symbol}] \to [\mathit{Bool}] \\
&\mathit{encode}_{\mathit{Bits}} = \mathit{toBits} \circ \mathit{foldl}\ \mathit{estep}_1\ \mathit{unit} \\
&\mathit{decode}_{\mathit{Bits}} :: [\mathit{Bool}] \to [\mathit{Symbol}] \\
&\mathit{decode}_{\mathit{Bits}} = \mathit{unfoldr}\ \mathit{dstep}_1 \circ \mathit{fromBits}
\end{aligned}
$$

Now streaming condition holds for *nextBit* and estep_1, so encoding can be streamed.

Also, *fromBits* can be written with a *foldl* (like *fromBase*3). The streaming condition doesn't hold immediately, but does with flushing. So decoding can be streamed too.

15. Towards ANS—fusion and fission

$$
\begin{array}{ll}
& \mathit{encode}_1 \\
= & \quad [\![\text{ definition; now let } \mathit{pick} = \mathit{fst} \quad]\!] \\
& \mathit{fst} \circ \mathit{foldl}\ \mathit{estep}_1\ \mathit{unit} \\
= & \quad [\![\text{ map fusion for } \mathit{foldl}\text{, backwards} \quad]\!] \\
& \mathit{fst} \circ \mathit{foldl}\ \mathit{narrow}\ \mathit{unit} \circ \mathit{map}\ \mathit{encodeSym} \\
= & \quad [\![\ \mathit{narrow} \text{ is associative} \quad]\!] \\
& \mathit{fst} \circ \mathit{foldr}\ \mathit{narrow}\ \mathit{unit} \circ \mathit{map}\ \mathit{encodeSym} \\
= & \quad [\![\text{ fusion for } \mathit{foldr} \quad]\!] \\
& \mathit{foldr}\ \mathit{weight}\ 0 \circ \mathit{map}\ \mathit{encodeSym} \\
= & \quad [\![\ \mathit{map} \text{ fusion; let } \mathit{estep}_2\ s\ x = \mathit{weight}\ (\mathit{encodeSym}\ s)\ x \quad]\!] \\
& \mathit{foldr}\ \mathit{estep}_2\ 0
\end{array}
$$

so let $\mathit{encode}_2 = \mathit{foldr}\ \mathit{estep}_2\ 0$.

16. Unfoldr–foldr theorem

Inverting a fold:

$$unfoldr\ g\ (foldr\ f\ e\ xs) = xs \Leftarrow g\ (f\ x\ z) = Just\ (x, z) \wedge g\ e = Nothing$$

Allowing junk:

$$(\exists ys\ .\ unfoldr\ g\ (foldr\ f\ e\ xs) = xs +\!\!\!+\ ys) \Leftarrow g\ (f\ x\ z) = Just\ (x, z)$$

With invariant:

$$\begin{aligned} unfoldr\ g\ (foldr\ f\ e\ xs) = xs \Leftarrow\ & ((g\ (f\ x\ z) = Just\ (x, z)) \Leftarrow p\ z) \wedge \\ & ((g\ e = Nothing) \Leftarrow p\ e) \end{aligned}$$

where invariant p of $foldr\ f\ e$ and $unfoldr\ g$ is such that

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

17. Correctness of decoding

$$
\begin{aligned}
& \mathit{dstep}_1\ (\mathit{estep}_2\ s\ z) \\
= \quad & [\![\ \mathit{estep}_2\ ]\!] \\
& \mathit{dstep}_1\ (\mathit{weight}\ (\mathit{encodeSym}\ s)\ z) \\
= \quad & [\![\ \mathit{dstep}_1;\ \text{let } s' = \mathit{decodeSym}\ (\mathit{weight}\ (\mathit{encodeSym}\ s)\ z)\ ]\!] \\
& \mathit{Just}\ (s', \mathit{scale}\ (\mathit{encodeSym}\ s')\ (\mathit{weight}\ (\mathit{encodeSym}\ s)\ z)) \\
= \quad & [\![\ s' = s \text{ (see below)}\ ]\!] \\
& \mathit{Just}\ (s, \mathit{scale}\ (\mathit{encodeSym}\ s)\ (\mathit{weight}\ (\mathit{encodeSym}\ s)\ z)) \\
= \quad & [\![\ \mathit{scale}\ i \circ \mathit{weight}\ i = \mathit{id}\ ]\!] \\
& \mathit{Just}\ (s, z)
\end{aligned}
$$

17. Correctness of decoding (continued)

Indeed, $s' = s$:

$$
\begin{array}{ll}
 & \mathit{decodeSym}\ (\mathit{weight}\ (\mathit{encodeSym}\ s)\ z) = s \\
\Longleftrightarrow & \quad [\![\ \text{central property}\ ]\!] \\
 & \mathit{weight}\ (\mathit{encodeSym}\ s)\ z \in \mathit{encodeSym}\ s \\
\Longleftrightarrow & \quad [\![\ \text{property of } \mathit{weight}\ ]\!] \\
 & z \in \mathit{unit}
\end{array}
$$

and $z \in \mathit{unit}$ is an invariant. Therefore

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

for all finite xs.

18. From fractions to integers

Where AC encodes longer messages as more precise fractions, ANS makes larger integers.

$$\begin{aligned} &\mathit{count} &&:: \mathit{Symbol} \rightarrow \mathit{Integer} \\ &\mathit{cumul} &&:: \mathit{Symbol} \rightarrow \mathit{Integer} \\ &\mathit{total} &&:: \mathit{Integer} \\ &\mathit{find} &&:: \mathit{Integer} \rightarrow \mathit{Symbol} \end{aligned}$$

such that

$$\mathit{find}\ z = s \Longleftrightarrow \mathit{cumul}\ s \leqslant z < \mathit{cumul}\ s + \mathit{count}\ s$$

for $0 \leqslant z < \mathit{total}$.

19. Asymmetric encoding: the idea

- text encoded as integer z, with $\log_2 z$ bits of information
- next symbol s has probability $p = \mathit{count}\ s \,/\, \mathit{total}$, so requires $\log_2 ({}^1/_p)$ bits
- so map z, s to $z' \simeq z \times \mathit{total} \,/\, \mathit{count}\ s$—but do so invertibly
- with $z' = (z \,`\mathit{div}`\, \mathit{count}\ s) \times \mathit{total}$, can undo the multiplication:

$$z \,`\mathit{div}`\, \mathit{count}\ s = z' \,`\mathit{div}`\, \mathit{total}$$

- what about s? with $z' = (z \,`\mathit{div}`\, \mathit{count}\ s) \times \mathit{total} + \mathit{cumul}\ s$,

$$s = \mathit{find}\ (\mathit{cumul}\ s) = \mathit{find}\ (z' \,`\mathit{mod}`\, \mathit{total})$$

- what about z? with $z' = (z \,`\mathit{div}`\, \mathit{count}\ s) \times \mathit{total} + \mathit{cumul}\ s + z \,`\mathit{mod}`\, \mathit{count}\ s$,

$$z \,`\mathit{mod}`\, \mathit{count}\ s = z' \,`\mathit{mod}`\, \mathit{total} - \mathit{cumul}\ s$$

20. ANS encoding and decoding

$$
\begin{aligned}
&\mathit{encode}_3 :: [\,\mathit{Symbol}\,] \to \mathit{Integer} \\
&\mathit{encode}_3 = \mathit{foldr}\ \mathit{estep}_3\ 0 \\
&\mathit{estep}_3 :: \mathit{Symbol} \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{estep}_3\ s\ z = \mathbf{let}\ (q, r) = z\,`\mathit{divMod}`\ \mathit{count}\ s\ \mathbf{in}\ q \times \mathit{total} + \mathit{cumul}\ s + r \\
&\mathit{decode}_3 :: \mathit{Integer} \to [\,\mathit{Symbol}\,] \\
&\mathit{decode}_3 = \mathit{unfoldr}\ \mathit{dstep}_3 \\
&\mathit{dstep}_3 :: \mathit{Integer} \to \mathit{Maybe}\ (\mathit{Symbol}, \mathit{Integer}) \\
&\mathit{dstep}_3\ z = \mathbf{let}\ (q, r) = z\,`\mathit{divMod}`\ \mathit{total} \\
&\qquad\qquad\qquad\ \ s = \mathit{find}\ r \\
&\qquad\qquad\ \ \mathbf{in}\ \mathit{Just}\ (s, \mathit{count}\ s \times q + r - \mathit{cumul}\ s)
\end{aligned}
$$

Correctness argument as before.

21. Variation

Correctness does not depend on starting value: can pick any l instead of 0.

Also, $estep_3$ strictly increasing on $z > 0$, and $dstep_3$ strictly decreasing, so we know when to stop:

$$
\begin{aligned}
&encode_4 :: [\,Symbol\,] \to Integer \\
&encode_4 = foldr\ estep_3\ l \\
&decode_4 :: Integer \to [\,Symbol\,] \\
&decode_4 = unfoldr\ dstep_4 \\
&dstep_4 :: Integer \to Maybe\ (Symbol, Integer) \\
&dstep_4\ z = \mathbf{if}\ z == l\ \mathbf{then}\ Nothing\ \mathbf{else}\ dstep_3\ z
\end{aligned}
$$

and we have

$$decode_4\ (encode_4\ xs) = xs$$

for all finite xs, without junk.

22. Bounded precision

Fix base b and lower bound l. Represent accumulator z as pair (x, ys) such that:

- *remainder* ys is a list of digits in base b
- *window* x satisfies $l \leqslant x < u$ for upper bound $u = l \times b$

under abstraction $z = \mathit{foldl}\ \mathit{inject}\ x\ ys$ where

$$\mathit{inject}\ x\ y = x \times b + y \qquad \text{and} \qquad \mathit{extract}\ x = x\ \text{`}\mathit{divMod}\text{`}\ b$$

Eg with $b = 10$ and $l = 100$, pair $(123, [4, 5, 6])$ represents 123456.

$$\mathbf{type}\ \mathit{Number} = (\mathit{Integer}, [\mathit{Integer}])$$

Note "you can't miss it" properties:

$$\begin{aligned} \mathit{inject}\ x\ y < u &\iff x < l \\ l \leqslant \mathit{fst}\ (\mathit{extract}\ x) &\iff u \leqslant x \end{aligned}$$

Want b, l powers of 2, u single-word. Also nice if $l\ \text{`}\mathit{mod}\text{`}\ \mathit{total} = 0$.

23. Encoding

Maintain window in range.

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

$$\begin{aligned}
&\mathit{estep}_5 :: \mathit{Symbol} \to \mathit{Number} \to \mathit{Number} \\
&\mathit{estep}_5\ s\ (x, ys) = \mathbf{let}\ (x', ys') = \mathit{enorm}_5\ s\ (x, ys)\ \mathbf{in}\ (\mathit{estep}_3\ s\ x', ys')
\end{aligned}$$

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

Eg with $b = 10, l = 100$:

$$(340, [\,3\,]) \xleftarrow{\text{‘a’}} (68, [\,3\,]) \xleftarrow{\text{norm}} (683, [\,]) \xleftarrow{\text{‘b’}} (205, [\,]) \xleftarrow{\text{‘c’}} (100, [\,])$$

24. Decoding

$$
\begin{array}{l}
\mathit{dproduce}_5 :: \mathit{Number} \rightarrow [\,\mathit{Symbol}\,] \\
\mathit{dproduce}_5 = \mathit{unfoldr}\ \mathit{dstep}_5
\end{array}
$$

$$
\begin{array}{l}
\mathit{dstep}_5 :: \mathit{Number} \rightarrow \mathit{Maybe}\ (\mathit{Symbol}, \mathit{Number}) \\
\mathit{dstep}_5\ (x, ys) = \ \ \mathbf{let}\ \mathit{Just}\ (s, x') = \mathit{dstep}_3\ x \\
\qquad\qquad\qquad\qquad\quad (x'', ys'') = \mathit{dnorm}_5\ (x', ys) \\
\qquad\qquad\qquad\quad\ \mathbf{in\ if}\ x'' \geqslant l\ \mathbf{then}\ \mathit{Just}\ (s, (x'', ys''))\ \mathbf{else}\ \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{l}
\mathit{dnorm}_5 :: \mathit{Number} \rightarrow \mathit{Number} \quad \text{-- } \mathit{dnorm}_5\ (\mathit{enorm}_5\ s\ (x, ys)) = (x, ys)\ \text{when}\ l \leqslant x < u \\
\mathit{dnorm}_5\ (x, y : ys) = \mathbf{if}\ x < l\ \mathbf{then}\ \mathit{dnorm}_5\ (\mathit{inject}\ x\ y, ys)\ \mathbf{else}\ (x, y : ys) \\
\mathit{dnorm}_5\ (x, [\,])\qquad = (x, [\,])
\end{array}
$$

Decoding is symmetric to encoding: renormalize after emitting a symbol.

$$
(340, [\,3\,]) \xrightarrow{\text{‘a’}} (68, [\,3\,]) \xrightarrow{\text{norm}} (683, [\,]) \xrightarrow{\text{‘b’}} (205, [\,]) \xrightarrow{\text{‘c’}} (100, [\,])
$$

Correctness again as before (no junk; invariant $l \leqslant x < u$).

25. Trading in sequences

$$\begin{aligned}
&\mathit{eflush}_5 :: \mathit{Number} \to [\,\mathit{Integer}\,]\\
&\mathit{eflush}_5\ (0, \mathit{ys}) = \mathit{ys}\\
&\mathit{eflush}_5\ (x, \mathit{ys}) = \mathbf{let}\ (x', y) = \mathit{extract}\ x\ \mathbf{in}\ \mathit{eflush}_5\ (x', y : \mathit{ys})
\end{aligned}$$

$$\begin{aligned}
&\mathit{encode}_5 :: [\,\mathit{Symbol}\,] \to [\,\mathit{Integer}\,]\\
&\mathit{encode}_5 = \mathit{eflush}_5 \circ \mathit{econsume}_5
\end{aligned}$$

$$\begin{aligned}
&\mathit{dstart}_5 :: [\,\mathit{Integer}\,] \to \mathit{Number}\\
&\mathit{dstart}_5\ \mathit{ys} = \mathit{dnorm}_5\ (0, \mathit{ys})
\end{aligned}$$

$$\begin{aligned}
&\mathit{decode}_5 :: [\,\mathit{Integer}\,] \to [\,\mathit{Symbol}\,]\\
&\mathit{decode}_5 = \mathit{dproduce}_5 \circ \mathit{dstart}_5
\end{aligned}$$

for which

$$\mathit{dstart}_5\ (\mathit{eflush}_5\ x) = x \Longleftarrow l \leqslant x < u$$

26. Streaming

Both $encode_5$ and $decode_5$ can be transformed into an unfold after a fold, albeit with some *reverse*s.

The *streaming condition* applies, so they can yield output before consuming all inputs. (Encoding needs a *flushing* phase too.)

But perhaps better not to take that route. In fact, $encode_5$ and $decode_5$ already correspond to fast imperative loops.

27. Fast loops

$$
\begin{array}{l}
\mathit{encode} :: [\mathit{Symbol}] \to [\mathit{Integer}] \\
\mathit{encode} = h_1\ l \circ \mathit{reverse}\ \mathbf{where} \\
\quad h_1\ x\ (s : ss) = \mathbf{let}\ x' = \mathit{estep}_3\ s\ x\ \mathbf{in}\ \mathbf{if}\ x' < u\ \mathbf{then}\ h_1\ x'\ ss\ \mathbf{else} \\
\qquad\qquad\qquad\quad \mathbf{let}\ (q, r) = \mathit{extract}\ x\ \mathbf{in}\ r : h_1\ q\ (s : ss) \\
\quad h_1\ x\ [\,] \qquad\quad = h_2\ x \\
\quad h_2\ x = \mathbf{if}\ x == 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ \mathbf{let}\ (x', y) = \mathit{extract}\ x\ \mathbf{in}\ y : h_2\ x'
\end{array}
$$

$$
\begin{array}{l}
\mathit{decode} :: [\mathit{Integer}] \to [\mathit{Symbol}] \\
\mathit{decode} = h_0\ 0 \circ \mathit{reverse}\ \mathbf{where} \\
\quad h_0\ x\ (y : ys) \quad \mid x < l = h_0\ (\mathit{inject}\ x\ y)\ ys \\
\quad h_0\ x\ ys \qquad\qquad\quad\ \ = h_1\ x\ ys \\
\quad h_1\ x\ ys \qquad\qquad\quad\ \ = \mathbf{let}\ \mathit{Just}\ (s, x') = \mathit{dstep}_3\ x\ \mathbf{in}\ h_2\ s\ x'\ ys \\
\quad h_2\ s\ x\ (y : ys) \mid x < l = h_2\ s\ (\mathit{inject}\ x\ y)\ ys \\
\quad h_2\ s\ x\ ys \qquad\qquad\ \ = \mathbf{if}\ x \geqslant l\ \mathbf{then}\ s : h_1\ x\ ys\ \mathbf{else}\ [\,]
\end{array}
$$