Arithmetic coding with folds and unfolds

Richard Bird and Jeremy Gibbons
with Barney Stratford
University of Oxford
AFP, August 2002

1. Introduction

- Arithmetic coding (AC) is a method for data compression
- AC can be more *effective* than Huffman coding and other rival schemes
- AC is well-suited to *adaptive* encoding
- Substantial effort has gone into making AC as *efficient* as competing methods
- AC was popularized in 1987 by Witten et al, who provided an 'accessible implementation' of 300 lines of C

Goals

- Our aim is to develop a 30-line Haskell program for the basic encoding and decoding algorithms
- The development makes use of the algebra of folds and unfolds
- We start with a naive executable program for encoding, then refine it
- Along the way, we will present some of the theory of folds and unfolds
- Be warned: there is a fair amount of arithmetic in arithmetic coding!

1.1. Arithmetic coding, naively

- Break source message into *symbols*, where a symbol is either a character or some logical grouping of characters
- Associate each distinct symbol with a semi-open *interval* of the unit interval [0..1)
- Successively *narrow* the unit interval by the intervals associated with each symbol of the message in turn
- Represent the final interval by choosing some *fraction* within it

1.2. Intervals

$$\begin{array}{lcl} \textbf{type}\ Fraction & = & Ratio\ Integer \\ \textbf{type}\ Interval & = & (Fraction, Fraction) \\ unit & :: & Interval \\ unit & = & (0,1) \\ within & :: & Fraction \rightarrow Interval \rightarrow Bool \\ within\ x\ (l,r) & = & l \leq x\ \wedge\ x < r \\ pick & :: & Interval \rightarrow Fraction \\ pick\ (l,r) & = & (l+r)/2 \end{array}$$

We will assume that $0 \leq l < r \leq 1$ for every $(l, r) :: Interval$.

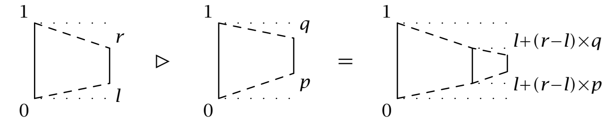
All we require of *pick* is that

$$pick\ int\ \underline{within}\ int$$

Narrowing

$$\begin{array}{lcl} (\triangleright) & :: & Interval \rightarrow Interval \rightarrow Interval \\ (l,r) \triangleright (p,q) & = & (l + (r{-}l) \times p,\ l + (r{-}l) \times q) \end{array}$$

Diagrammatically:



1.3. Models

Model is an abstract type representing a finite mapping from *Symbol*s to *Interval*s, with associated functions

$$\begin{array}{lcl} encodeSym & :: & Model \rightarrow Symbol \rightarrow Interval \\ decodeSym & :: & Model \rightarrow Fraction \rightarrow Symbol \end{array}$$

We assume that for any $m :: Model$ and $x :: Fraction$ such that $x\ \underline{within}\ unit$, there is a unique $s :: Symbol$ such that

$$x\ \underline{within}\ encodeSym\ m\ s$$

(so intervals in the model partition *unit*). Consequently,

$$s = decodeSym\ m\ x \quad \equiv \quad x\ \underline{within}\ encodeSym\ m\ s$$

Adaptive models

Rather than a single fixed model for a whole message, the model can *adapt* as the message is read.

Therefore we suppose also

$$newModel\ ::\ Model \rightarrow Symbol \rightarrow Model$$

As long as the decoder performs the same adaptations as the message is reconstructed, the message can be retrieved.

Crucially, *there is no need to transmit the model with the message.*

1.4. Encoding

$$
\begin{array}{lcl}
encode_0 & :: & Model \to [Symbol] \to Fraction \\
encode_0\ m & = & pick \cdot foldl\ (\triangleright)\ unit \cdot encodeSyms\ m \\
encodeSyms & :: & Model \to [Symbol] \to [Interval] \\
encodeSyms\ m\ ss & = & unfoldr\ nextInt\ (m, ss) \\
nextInt & :: & (Model, [Symbol]) \to \\
 & & \quad Maybe\ (Interval, (Mode, [Symbol])) \\
nextInt\ (m, []) & = & Nothing \\
nextInt\ (m, s : ss) & = & Just\ (encodeSym\ m\ s, (newModel\ m\ s, ss))
\end{array}
$$

1.5. Folds

foldl iterates over a list, from left to right:

$$
\begin{array}{lcl}
foldl & :: & (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
foldl\ f\ e\ [] & = & e \\
foldl\ f\ e\ (x : xs) & = & foldl\ f\ (f\ e\ x)\ xs
\end{array}
$$

Thus,

$$foldl\ (\oplus)\ e\ [x, y, z] \quad = \quad ((e \oplus x) \oplus y) \oplus z$$

Another fold: *foldr*

foldr iterates over a list, from right to left:

$$
\begin{array}{lcl}
foldr & :: & (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
foldr\ f\ e\ [] & = & e \\
foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs)
\end{array}
$$

Thus,

$$foldr\ (\oplus)\ e\ [x, y, z] \quad = \quad x \oplus (y \oplus (z \oplus e))$$

Crucial fact: universal property — for strict h,

$$h = foldr\ f\ e \quad \equiv \quad h\ [] = e \wedge h\ (x : xs) = f\ x\ (h\ xs)$$

A relationship between *foldl* and *foldr*

Theorem (First Duality Theorem). If $\oplus$ is associative with unit e, then

$$foldl\ f\ e\ xs \quad = \quad foldr\ f\ e\ xs$$

for finite lists xs.

Conversely,

Theorem Third Homomorphism Theorem. If $h = foldl\ f_1\ e = foldr\ f_2\ e$, then there is an associative f with unit e such that $h = foldr\ f\ e$

foldl vs *foldr*

From duality, we have

$$foldl\ (\triangleright)\ unit \quad = \quad foldr\ (\triangleright)\ unit$$

So why not use the (more familiar) *foldr*?

Lemma.

$$foldl\ (\triangleright)\ unit \cdot encodeSyms\ m \quad = \quad snd \cdot foldl\ step\ (m, unit)$$

where

$$step\ (m, int)\ s \quad = \quad (newModel\ m\ s, int \triangleright encodeSym\ m\ s)$$

... but there is no corresponding result for *foldr*.

1.6. *unfoldr* and *Maybe*

Recall the Haskell standard type *Maybe*:

$$\mathbf{data}\ Maybe\ \alpha \quad = \quad Just\ \alpha \mid Nothing$$

and the function *unfoldr*:

$$\begin{array}{lcl} unfoldr & :: & (\beta \to Maybe\ (\alpha, \beta)) \to \beta \to [\alpha] \\ unfoldr\ f\ b & = & \mathbf{case}\ f\ b\ \mathbf{of} \\ & & \quad Just\ (a, b') \to a : unfoldr\ f\ b' \\ & & \quad Nothing \to [\,] \end{array}$$

A relationship between *unfoldr* and *foldr*

The Haskell Library Report states:

> The *unfoldr* function undoes a *foldr* operation... :
>
> $$unfoldr\ f'\ (foldr\ f\ z\ xs) \quad = \quad xs$$
>
> if the following holds:
>
> $$\begin{array}{lcl} f'\ (f\ x\ y) & = & Just\ (x, y) \\ f'\ z & = & Nothing \end{array}$$

(and that's essentially all it says on unfolds!).

1.7. Decoding

$$decode_0 \quad :: \quad Model \to Fraction \to [Symbol]$$

specified by

$$ss\ \underline{begins}\ decode_0\ m\ (encode_0\ m\ ss)$$

for all *ss* (where $xs\ \underline{begins}\ ys$ if $ys = xs +\!\!+\ xs'$ for some xs').

So if

$$compress\ m\ ss \quad = \quad (length\ ss, encode_0\ m\ ss)$$

we can define

$$decompress\ m\ (n, x) \quad = \quad take\ n\ (decode_0\ m\ x)$$

1.8. Refinements

- We really want an encoding function that returns a list of bits rather than a fraction
- For efficiency, encoding should produce bits as soon as they are known — *incremental* transmission, or *streaming*
- Decoding should be implemented as a function that consumes bits and produces symbols
- For efficiency in both time and space, we should replace computations on fractions with computations on fixed-precision integers, although this will degrade effectiveness
- We have to choose a suitable representation of models

2. Bits and streaming

We replace

$$pick :: Interval \to Fraction$$

by two functions

```
type Bit = Int           -- 0 and 1 only
toBits   :: Interval → [Bit]
fromBits :: [Bit] → Fraction
```

such that

$$pick = fromBits \cdot toBits$$

or equivalently, for all intervals *int*,

$$fromBits\ (toBits\ int)\ \underline{within}\ int$$

2.1. Defining *toBits*

We define

$$toBits = unfoldr\ nextBit$$

where

```
nextBit            :: Interval → Maybe (Bit, Interval)
nextBit (l, r)
    | r ≤ 1/2      = Just (0, (2×l, 2×r))
    | 1/2 ≤ l      = Just (1, (2×l − 1, 2×r − 1))
    | otherwise    = Nothing
```


Length

Proposition.

$$length\ (toBits\ (l, r)) \leq -\log_2(r - l)$$

Proof (sketch).

$$0 \leq l < r \leq 1 \ \wedge\ 1/2 < r - l \ \Rightarrow\ l < 1/2 < r$$

Each iteration of *nextBit* doubles the width of the interval.

So if

$$1/2^{n+1} < r - l < 1/2^n$$

or equivalently

$$n \leq -\log_2(r - l) < n+1$$

then termination is guaranteed after at most n bits have been produced.

2.2. Defining *fromBits*

$$\begin{array}{lcl} fromBits & :: & [Bit] \to Fraction \\ fromBits & = & foldr\ pack\ (^1/_2) \\ pack & :: & Bit \to Fraction \to Fraction \\ pack\ b\ x & = & (b + x)/2 \end{array}$$

Proposition.

$$fromBits\ (toBits\ int)\quad \underline{within}\quad int$$

2.3. Hylomorphisms

Consider *hylomorphism*s, functions h of the form

$$h = foldr\ f\ e \cdot unfoldr\ g$$

Recall that

$$\begin{array}{lcl} unfoldr\ g\ z & = & \mathbf{case}\ g\ z\ \mathbf{of} \\ & & \quad Nothing \to [\,] \\ & & \quad Just\ (x, z') \to x : unfoldr\ g\ z' \\ foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \end{array}$$

The two loops can be *fused*, and the intermediate list *deforested*:

$$\begin{array}{lcl} h\ z & = & \mathbf{case}\ g\ z\ \mathbf{of} \\ & & \quad Nothing \to e \\ & & \quad Just\ (x, z') \to f\ x\ (h\ z') \end{array}$$

Two hylos

For $pick = fromBits \cdot toBits$, we get

$$\begin{array}{lcl} pick\ (l, r) & & \\ \quad \mid r \le {}^1/_2 & = & pick\ (2{\times}l, 2{\times}r)/2 \\ \quad \mid {}^1/_2 \le l & = & (1 + pick\ (2{\times}l - 1, 2{\times}r - 1))/2 \\ \quad \mid l < {}^1/_2 < r & = & {}^1/_2 \end{array}$$

For $size = foldr\ count\ 0 \cdot toBits$ where $count\ a\ n = 1 + n$, we get

$$\begin{array}{lcl} size\ (l, r) & & \\ \quad \mid r \le {}^1/_2 & = & 1 + size\ (2{\times}l, 2{\times}r) \\ \quad \mid {}^1/_2 \le l & = & 1 + size\ (2{\times}l - 1, 2{\times}r - 1) \\ \quad \mid l < {}^1/_2 < r & = & 0 \end{array}$$

Proof principle for hylos

For hylo

$$h \;=\; foldr\ f\ e \cdot unfoldr\ g$$

to show $P(x, h\ x)$ for every x for which $h\ x \ne \bot$, it suffices to perform a case analysis on $g\ x$ and show that

Case $g\ x = Nothing$**:** $P(x, e)$

Case $g\ x = Just\ (a, y)$**:** $P(y, z) \Rightarrow P(x, f\ a\ z)$

provided that f is strict in its second argument.

2.4. Summary of first refinement

We define

$$\begin{array}{lll} encode_1 & :: & Model \to [Symbol] \to [Bit] \\ encode_1\ m & = & toBits \cdot foldl\ (\triangleright)\ unit \cdot encodeSyms\ m \end{array}$$

Informally, execution of $encode_1$ consumes all its input before delivering any output.

Formally,

$$encode_1\ m\ ss \quad = \quad \bot$$

for all partial or infinite lists *ss*.

Can we do better?

2.5. Streaming

The function *stream* alternates between production and consumption:

$$\begin{array}{l} stream :: (\beta \to ([\alpha], \beta)) \to (\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to [\alpha] \\ stream\ f\ g\ z\ xs \quad = \quad ys +\!\!+ \ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad [\,] \quad \to \quad [\,] \\ \qquad\qquad\qquad\qquad\qquad\qquad (x : xs) \quad \to \quad stream\ f\ g\ (g\ z'\ x)\ xs \\ \qquad\qquad\qquad\quad \mathbf{where}\ (ys, z') = f\ z \end{array}$$

(Informally, γ is the input type, β the state, and α the output type.)

2.6. *evolve*: a variation on *unfoldr*

$$\begin{array}{lll} evolve & :: & (\beta \to Maybe\ (\alpha, \beta)) \to \beta \to ([\alpha], \beta) \\ evolve\ f\ z & = & \mathbf{case}\ f\ z\ \mathbf{of} \\ & & \quad Nothing \quad \to \quad ([\,], z) \\ & & \quad Just(y, z') \quad \to \quad (y : ys, z'') \\ & & \mathbf{where}\ (ys, z'') = evolve\ f\ z' \end{array}$$

Claim: *evolve* yields the unfold and the final state.

$$evolve\ f\ z \quad = \quad (unfoldr\ f\ z, loop\ (fmap\ snd \cdot f)\ z)$$

where the *fmap* here is the map for *Maybe*, and

$$\begin{array}{lll} loop & :: & (\beta \to Maybe\ \beta) \to \beta \to \beta \\ loop\ f\ z & = & \mathbf{case}\ f\ z\ \mathbf{of} \\ & & \quad Nothing \quad \to \quad z \\ & & \quad Just\ z' \quad \to \quad loop\ f\ z' \end{array}$$

2.7. The Streaming Theorem

Definition. The *streaming condition* for f and g is:

$$f\ z = Just\ (y, z') \quad \Rightarrow \quad f\ (g\ z\ x) = Just\ (y, g\ z'\ x)$$

for all z, y, z' and x.

Lemma. If the streaming condition holds for f and g, then

$$f\ z = Just\ (y, z') \quad \Rightarrow \quad f\ (foldl\ g\ z\ xs) = Just\ (y, foldl\ g\ z'\ xs)$$

for all z, y, z' and finite lists *xs*.

Theorem. If the streaming condition holds for f and g, then

$$unfoldr\ f\ (foldl\ g\ z\ xs) \quad = \quad stream\ (evolve\ f)\ g\ z\ xs$$

for all *finite* lists *xs*.

Proof of Lemma

By induction on xs:

Case $[]$**:** Immediate.

Case $x : xs$**:** Assume $f\ z = Just\ (y, z')$. So by streaming condition, $f\ (g\ z\ x) = Just\ (y, g\ z\ x')$. Now we have

$$
\begin{array}{ll}
 & f\ (foldl\ g\ z\ (x : xs)) \\
= & \quad \{\text{definition of } foldl\} \\
 & f\ (foldl\ g\ (g\ z\ x)\ xs) \\
= & \quad \{\text{induction}\} \\
 & Just\ (y, foldl\ g\ (g\ z'\ x)\ xs) \\
= & \quad \{\text{definition of } foldl\} \\
 & Just\ (y, foldl\ g\ z'\ (x : xs))
\end{array}
$$

Approx lemma

Recall *approx lemma* (IFPH §9.3):

Lemma. Let

$$
\begin{array}{lcl}
approx :: Integer \to [\alpha] \to [\alpha] & & \\
approx\ (n+1)\ [] & = & [] \\
approx\ (n+1)\ (x : xs) & = & x : approx\ n\ xs
\end{array}
$$

Then two lists xs and ys are equal iff

$$(\forall n : 0 \leq n : approx\ n\ xs = approx\ n\ ys)$$

Proof of Streaming Theorem

A *double induction* on xs and n to show that

$$
\begin{array}{ll}
 & approx\ n\ (unfoldr\ f\ (foldl\ g\ z\ xs)) \\
= & approx\ n\ (stream\ (evolve\ f)\ g\ z\ xs)
\end{array}
$$

if the streaming condition holds for f and g.

Case $xs = []$**:** Then

$$
\begin{array}{lcl}
\text{lhs} & = & approx\ n\ (unfoldr\ f\ z) \\
\text{rhs} & = & approx\ n\ (fst\ (evolve\ f\ z))
\end{array}
$$

Case $xs = x : xs'$**:** We perform a case analysis on $f\ z$.

Subcase $f\ z = Nothing$**:** Then

$$
\begin{array}{lcl}
\text{lhs} & = & approx\ n\ (unfoldr\ f\ (foldl\ g\ (g\ z\ x)\ xs')) \\
 & = & \quad \{\text{induction}\} \\
 & & approx\ n\ (stream\ (evolve\ f)\ g\ (g\ z\ x)\ xs') \quad = \ \text{rhs}
\end{array}
$$

Subcase $f\ z = Just\ (y, z')$**:** We perform an induction on n.

Subsubcase $n = 0$**:** Trivial, since $approx\ 0\ xs = \bot$ for any xs.

Subsubcase $n = n' + 1$**:** Then

$$
\begin{array}{lcl}
\text{lhs} & = & approx\ (n' + 1)\ (unfoldr\ f\ (foldl\ g\ z\ (x : xs'))) \\
 & = & \quad \{\text{by lemma}\} \\
 & & approx\ (n' + 1)\ (y : unfoldr\ f\ (foldl\ g\ z'\ (x : xs'))) \\
 & = & \quad \{approx\} \\
 & & y : approx\ n'\ (unfoldr\ f\ (foldl\ g\ z'\ (x : xs'))) \\
 & = & \quad \{\text{induction}\} \\
 & & y : approx\ n'\ (stream\ (evolve\ f)\ g\ z'\ (x : xs')) \\
 & = & \quad \{approx\} \\
 & & approx\ (n' + 1)\ (y : stream\ (evolve\ f)\ g\ z'\ (x : xs')) \\
 & = & \quad \{stream, evolve\} \\
 & & approx\ (n' + 1)\ (stream\ (evolve\ f)\ g\ z\ (x : xs')) \quad = \ \text{rhs}
\end{array}
$$

2.8. Streaming condition for *nextBit* and $\triangleright$

Recall

$$\begin{array}{lll} nextBit & :: & Interval \to Maybe\ (Bit, Interval) \\ nextBit\ (l, r) & & \\ \quad \mid r \le {}^1\!/_2 & = & Just\ (0, (0,2) \triangleright (l, r)) \\ \quad \mid {}^1\!/_2 \le l & = & Just\ (1, (-1,1) \triangleright (l, r)) \\ \quad \mid l < {}^1\!/_2 < r & = & Nothing \end{array}$$

We leave it as an exercise to show that the streaming condition follows from associativity of $\triangleright$, and the fact that $int_1 \triangleright int_2$ is contained within int_1.

2.9. Summary of second refinement

Previously,

$$\begin{array}{lll} encode_1 & :: & Model \to [Symbol] \to [Bit] \\ encode_1\ m & = & unfoldr\ nextBit \cdot foldl\ (\triangleright)\ unit \cdot encodeSyms\ m \end{array}$$

Now,

$$\begin{array}{lll} encode_2 & :: & Model \to [Symbol] \to [Bit] \\ encode_2\ m & = & stream\ (evolve\ nextBit)\ (\triangleright)\ unit \cdot encodeSyms\ m \end{array}$$

Note that $encode_1\ m \neq encode_2\ m$, but they *are* equal on all finite symbol sequences.

3. Decoding and interval expansion

The function $decode_2 :: Model \to [Bit] \to [Symbol]$ is specified by

$$ss\ \underline{begins}\ decode_2\ m\ (encode_2\ m\ ss)$$

for all finite sequences of symbols *ss*, where

$$\begin{array}{lll} encode_2 & :: & Model \to [Symbol] \to [Bit] \\ encode_2\ m & = & stream\ (evolve\ nextBit)\ (\triangleright)\ unit \cdot encodeSyms\ m \end{array}$$

How can we implement $decode_2$?

3.1. The Stream Inversion Theorem

Theorem. Let $process\ z = foldr\ (\oplus)\ e \cdot stream\ f\ g\ z$.

Suppose $\ominus$ and *extract* can be found so that

1. $(y \oplus w) \ominus y = w$, for all y and w.
2. $w = process\ z\ (x : xs) \Rightarrow extract\ (w, z) = x$, for all w, z, x and xs.

Then $xs\ \underline{begins}\ unfoldr\ nextx\ (process\ z\ xs, z)$, where

$$\begin{array}{lll} nextx\ (w, z) & = & Just\ (x, (foldl\ (\ominus)\ w\ ys, g\ z'\ x)) \\ & & \quad \textbf{where}\ x = extract\ (w, z) \\ & & \quad \phantom{\textbf{where}\ } (ys, z') = f\ z \end{array}$$

3.2. Proof of the Inversion Theorem

The proof is by induction on *xs*.

Case []: Observe firstly that *unfoldr nextx* (*process z xs*, *z*) evaluates to an infinite list (of possibly undefined elements). Since [] begins every finite or infinite list, the case is established.

Case $x : xs$: Let $w = \mathit{process}\ z\ (x : xs)$, so $x = \mathit{extract}\ (w, z)$ and

$$\mathit{unfoldr\ nextx}\ (w, z) \quad = \quad x : \mathit{unfoldr\ nextx}\ (\mathit{foldl}\ (\ominus)\ w\ ys, g\ z'\ x)$$

where $(ys, z') = f\ z$. We therefore have to show

$$xs \quad \underline{\mathit{begins}} \quad \mathit{unfoldr\ nextx}\ (\mathit{foldl}\ (\ominus)\ w\ ys, g\ z'\ x)$$

Proof, continued

By induction, $xs\ \underline{\mathit{begins}}\ \mathit{unfoldr\ nextx}\ (\mathit{foldl}\ (\ominus)\ w\ ys, g\ z'\ x)$ if

$$\mathit{foldl}\ (\ominus)\ w\ ys \quad = \quad \mathit{process}\ (g\ z'\ x)\ xs$$

By definition of *stream* and $f\ z = (ys, z')$ we have

$$w \quad = \quad \mathit{foldr}\ (\oplus)\ e\ (ys +\!\!+ \mathit{stream}\ f\ g\ (g\ z'\ x)\ xs)$$

Furthermore,

$$\mathit{foldl}\ (\ominus)\ (\mathit{foldr}\ (\oplus)\ e\ (ys +\!\!+ zs))\ ys \quad = \quad \mathit{foldr}\ (\oplus)\ e\ zs$$

for all *ys* and *zs*. The proof is left as an exercise.

Proof, continued

Hence

$$\begin{array}{cl} & \mathit{foldl}\ (\ominus)\ w\ ys \\ = & \quad \{\text{above expression for } w\} \\ & \mathit{foldl}\ (\ominus)\ (\mathit{foldr}\ (\oplus)\ e\ (ys +\!\!+ \mathit{stream}\ f\ g\ (g\ z'\ x)\ xs)) \\ = & \quad \{\text{above identity}\} \\ & \mathit{foldr}\ (\oplus)\ e\ (\mathit{stream}\ f\ g\ (g\ z'\ x)\ xs) \\ = & \quad \{\text{definition of } \mathit{process}\} \\ & \mathit{process}\ (g\ z'\ x)\ xs \end{array}$$

Summarising, $\mathit{foldl}\ (\ominus)\ w\ ys = \mathit{process}\ (g\ z'\ x)\ xs$, establishing the case and the theorem.

3.3. Applying the Inversion Theorem: a preliminary step

Recall that in the first lecture we showed

$$\mathit{foldl}\ (\triangleright)\ \mathit{unit} \cdot \mathit{encodeSyms}\ m \quad = \quad \mathit{snd} \cdot \mathit{foldl\ step}\ (m, \mathit{unit})$$

where

$$\mathit{step}\ (m, \mathit{int})\ s \quad = \quad (\mathit{newModel}\ m\ s, \mathit{int} \triangleright \mathit{encodeSym}\ m\ s)$$

This identity allows us to fuse *encodeSyms* into the narrowing process:

$$\mathit{encode}_1\ m \quad = \quad \mathit{unfoldr\ nextBitM} \cdot \mathit{foldl\ step}\ (m, \mathit{unit})$$

where *nextBitM* is identical to *nextBit* except that it propagates the model as an additional argument. The Streaming Theorem is again applicable and we obtain

$$\mathit{encode}_2\ m \quad = \quad \mathit{stream}\ (\mathit{evolve\ nextBitM})\ \mathit{step}\ (m, \mathit{unit})$$

3.4. Applying the Inversion Theorem: $\ominus$ and *extract*

Let $process\ (m, int) = fromBits \cdot stream\ (evolve\ nextBitM)\ step\ (m, int)$.

Recall that $fromBits = foldr\ (\oplus)\ ({}^{1}/_{2})$ where $b \oplus x = {}^{b+x}/_{2}$. Defining $\ominus$ by $x \ominus b = 2 \times x - b$, we therefore have $(b \oplus x) \ominus b = x$.

Let $x = process\ (m, int)\ (s : ss)$, so $x\ \underline{within}\ (int \triangleright encodeSym\ m\ s)$. If $int = (l, r)$ and $encodeSym\ m\ s = (p, q)$, then

$$
\begin{array}{ll}
 & l + (r - l) \times p \le x < l + (r - l) \times q \\
\equiv & \quad \{\text{arithmetic}\} \\
 & p \le {}^{x-l}/_{r-l} < q \\
\equiv & \quad \{\text{definition of } decodeSym\} \\
 & s = decodeSym\ m\ ({}^{x-l}/_{r-l})
\end{array}
$$

Hence $extract\ (x, (m, (l, r))) = decodeSym\ m\ ({}^{x-l}/_{r-l})$.

3.5. Definition of $decode_2$

We therefore obtain

$$
\begin{array}{lll}
decode_2\ m\ bs & = & unfoldr\ nextSym\ (fromBits\ bs, (m, unit)) \\
nextSym\ (x, (m, int)) & & \\
 & = & Just\ (s, (x', (newModel\ m\ s, \\
 & & \qquad int' \triangleright (encodeSym\ m\ s)))) \\
 & & \textbf{where}\ s = decodeSym\ m\ (resize\ x\ int) \\
 & & \qquad x' = foldl\ (\lambda\ b\ x \rightarrow 2 \times x - b)\ x\ bs \\
 & & \qquad (bs, int') = evolve\ nextBit\ int \\
resize\ x\ (l, r) & = & {}^{x-l}/_{r-l}
\end{array}
$$

Note, however, that $decode_2$ inspects all of bs before producing any output, so is not an incremental algorithm.

3.6. Interval expansion

In the final lecture we are going to replace fractional arithmetic by arithmetic with limited-precision integers. Before doing so we need a preparatory step: interval expansion.

Quoting from Howard and Vitter:

> The idea is to prevent the current interval from narrowing too much when the endpoints are close to $^{1}/_{2}$ but straddle $^{1}/_{2}$. In that case we do not yet know the next output bit, but we do know that whatever it is, the *following* bit will have the opposite value; we merely keep track of that fact, and expand the current interval about $^{1}/_{2}$. This follow-on procedure may be repeated any number of times, so the current interval is always strictly longer than $^{1}/_{4}$.

Interval expansion, formally

Formally, interval expansion is a data refinement in which an interval (l, r) is represented by a triple of the form $(n, (l', r'))$ satisfying

$$l' = scale\ (n, l) \quad \text{and} \quad r' = scale\ (n, r)$$

where

$$scale\ (n, x) \quad = \quad 2^{n} \times (x - {}^{1}/_{2}) + {}^{1}/_{2}$$

and subject to $0 \le l' < r' \le 1$. In particular, $(0, (l, r))$ is one possible representation of (l, r).

A *fully-expanded* interval is a triple $(n, (l', r'))$ in which n is as large as possible. Intervals will be fully-expanded immediately before narrowing.

The remainder of this lecture is devoted to installing this data refinement.

Definition of *expand*

Since

$$0 \le 2 \times (l - {}^1\!/_2) + {}^1\!/_2 \;\equiv\; {}^1\!/_4 \le l$$
$$2 \times (r - {}^1\!/_2) + {}^1\!/_2 \le 1 \;\equiv\; r \le {}^3\!/_4$$

we can further expand $(n, (l, r))$ if ${}^1\!/_4 \le l$ and $r \le {}^3\!/_4$. Hence we define

$$\begin{array}{lll} \textit{expand}\ (n,(l,r)) & & \\ \quad \mid {}^1\!/_4 \le l \wedge r \le {}^3\!/_4 & = & \textit{expand}\ (n{+}1, (2{\times}l - {}^1\!/_2, 2{\times}r - {}^1\!/_2)) \\ \quad \mid \textbf{otherwise} & = & (n,(l,r)) \end{array}$$

In particular, *expand* $(0, (l, r))$ returns a fully-expanded interval for (l, r).

3.7. Interval contraction

The converse of *expand* is given by

$$\textit{contract}\ (n,(l,r)) \;=\; (\textit{rescale}\ (n,l), \textit{rescale}\ (n,r))$$

where $\textit{rescale}\ (n,x) = (x - {}^1\!/_2)/2^n + {}^1\!/_2$.

We leave it as an exercise to verify that

$$\begin{array}{lll} \textit{contract} \cdot \textit{expand} & = & \textit{contract} \\ \textit{contract}\ (n, \textit{int}1 \triangleright \textit{int}2) & = & \textit{contract}\ (n, \textit{int}1) \triangleright \textit{int}2 \end{array}$$

Consequently, defining *enarrow* (“expand and narrow”) by

$$\begin{array}{lll} \textit{enarrow}\ \textit{ei}\ \textit{int}2 & = & (n, \textit{int}1 \triangleright \textit{int}2) \\ & & \quad \textbf{where}\ (n, \textit{int}1) = \textit{expand}\ \textit{ei} \end{array}$$

we have $\textit{contract}\ (\textit{enarrow}\ \textit{ei}\ \textit{int}) = \textit{contract}\ \textit{ei} \triangleright \textit{int}$.

3.8. Aim

Our aim is to show

$$\begin{array}{ll} & \textit{stream}\ (\textit{evolve}\ \textit{nextBit})\ (\triangleright)\ (\textit{contract}\ \textit{ei}) \\ = & \textit{concat} \cdot \textit{stream}\ (\textit{evolve}\ \textit{nextBits})\ \textit{enarrow}\ \textit{ei} \end{array}$$

for the following definition of *nextBits*:

$$\begin{array}{lll} \textit{nextBits}\ (n,(l,r)) & & \\ \quad \mid r \le {}^1\!/_2 & = & \textit{Just}\ (\textit{bits}\ n\ 0, (0, (2{\times}l, 2{\times}r))) \\ \quad \mid {}^1\!/_2 \le l & = & \textit{Just}\ (\textit{bits}\ n\ 1, (0, (2{\times}l - 1, 2{\times}r - 1))) \\ \quad \mid \textbf{otherwise} & = & \textit{Nothing} \end{array}$$

where $\textit{bits}\ n\ b = b : \textit{replicate}\ n\ (1-b)$ returns a b followed by a sequence of n copies of $1-b$.

3.9. Stream fusion

Say that $k :: [\alpha] \to [\beta]$ is *distributive* if

$$\begin{array}{lll} k\ (xs \mathbin{+\!\!+} ys) & = & k\ xs \mathbin{+\!\!+} k\ ys \\ k\ [\,] & = & [\,] \end{array}$$

Examples of distributive functions are *id*, *map f*, and *concat*.

Theorem. Suppose k is distributive. Then

$$k \cdot \textit{stream}\ f\ g\ z \;=\; \textit{stream}\ f'\ g'\ (h\ z)$$

if the following two conditions hold for all z, z', ys and x:

$$\begin{array}{lll} f\ z = (ys, z') & \Rightarrow & f'\ (h\ z) = (k\ ys, h\ z') \\ h\ (g\ z\ x) & = & g'\ (h\ z)\ x \end{array}$$

We leave the proof as an exercise in fixpoint induction.

Applying stream fusion

We have that *concat* is distributive, and that

$$contract\ (enarrow\ ei\ int) \quad = \quad (contract\ ei) \triangleright int$$

It remains to verify the remaining fusion condition:

$$\begin{array}{ll} & evolve\ nextBits\ ei1 = (bss, ei2) \\ \Rightarrow & evolve\ nextBit\ (contract\ ei1) = (concat\ bss, contract\ ei2) \end{array}$$

Again, we relegate the proof to the exercises.

3.10. Conclusion

We have derived

$$\begin{array}{lll} encode_3\ m & = & concat \cdot stream\ nextBits\ enarrow\ (0, unit) \cdot \\ & & encodeSyms\ m \end{array}$$

In the new version we expand intervals (l, r) satisfying $l < {}^1\!/_2 < r$ immediately before narrowing. Consequently, narrowing is applied only when $l < {}^1\!/_4$ and ${}^1\!/_2 < r$, or $l < {}^1\!/_2$ and ${}^3\!/_4 < r$.

In either case, ${}^1\!/_4 < r - l$.

This fact will be crucial in the final lecture.

4. From fractions to integers

We now want to replace fractional arithmetic by arithmetic with limited-precision integers.

In the final version intervals take the form (l, r), where l and r are integers in the range $0 \le l < r \le w$ and w is a *fixed* power of two. This pair represents the interval $({}^l\!/_w, {}^r\!/_w)$.

Intervals in each model m take the form (p, q, d), where p and q are integers in the range $0 \le p < q \le d$ and d is an integer which is fixed for m and called the *denominator* for m. This triple represents the interval $({}^p\!/_d, {}^q\!/_d)$.

4.1. Integer narrowing

The narrowing function is redefined as follows:

$$(l, r) \blacktriangleright (p, q, d) \quad = \quad (l + \lfloor (r-l) \times {}^p\!/_d \rfloor, l + \lfloor (r-l) \times {}^q\!/_d \rfloor)$$

Equivalently

$$(l, r) \blacktriangleright (p, q, d) \quad = \quad (l + ((r-l) \times p)\ \underline{div}\ d, l + ((r-l) \times q)\ \underline{div}\ d)$$

There are various problems with this idea, and a number of obstacles that have to be overcome.

Problems

- The revised definition of narrowing completely changes the specification: encoding will now produce different outputs than before. In general, the effectiveness of compression will be reduced.
- Worse, $\blacktriangleright$ is not associative and none of the foregoing development applies.
- Unless we take steps to avoid it, intervals can *collapse* to the empty interval when $\lfloor (r-l) \times {}^{p}/_{d} \rfloor = \lfloor (r-l) \times {}^{q}/_{d} \rfloor$.

4.2. Change of specification

Fortunately, we can recover all of the previous development. Observe that

$$(l, r) \blacktriangleright (p, q, d) \quad = \quad ({}^{l}/_{w}, {}^{r}/_{w}) \triangleright ({}^{p'}/_{d}, {}^{q'}/_{d})$$

where

$$p' \;=\; {}^{d}/_{r-l} \times \lfloor (r-l) \times {}^{p}/_{d} \rfloor$$
$$q' \;=\; {}^{d}/_{r-l} \times \lfloor (r-l) \times {}^{q}/_{d} \rfloor$$

Hence, provided $p' < q'$, integer narrowing can be viewed as fractional narrowing applied in an *adjusted* model. The adjustment to the model depends on the current interval (l, r) but that doesn't matter. What does follow is that our development remains valid because we have allowed for changing models in the specification.

4.3. When intervals collapse

It is left as an exercise to show that

$$(\forall p, q : 0 \le p < q \le d : \lfloor (r-l) \times {}^{p}/_{d} \rfloor < \lfloor (r-l) \times {}^{q}/_{d} \rfloor)$$

if and only if $d \le r - l$.

Hence we have to ensure that the width of each interval is at least d before narrowing. But interval expansion guarantees that the width of each (expanded) interval is greater than ${}^{w}/_{4}$ before narrowing, so interval collapse is avoided if ${}^{w}/_{4} \ge d$.

Since $w \times d \le w \times {}^{w}/_{4} = 2^{2\times e-2}$ if $w = 2^{e}$, we have to ensure that our limited-precision arithmetic is accurate to $2 \times e - 2$ bits.

4.4. Final version of encode

$$\begin{array}{lll}
\mathit{encode}\ m & = & \mathit{concat} \cdot \mathit{stream}\ (\mathit{evolve}\ \mathit{nextBits})\ \mathit{enarrow}\ \mathit{unit} \cdot \\
 & & \mathit{encodeSyms}\ m
\end{array}$$

$$\begin{array}{lll}
\mathit{unit} & = & (0, (0, w)) \\
\mathit{enarrow}\ \mathit{ei}\ \mathit{int}2 & = & (n, \mathit{int}1 \blacktriangleright \mathit{int}2) \\
 & & \quad \textbf{where}\ \ (n, \mathit{int}1) = \mathit{expand}\ \mathit{ei} \\
\mathit{expand}\ (n, (l, r)) & & \\
\quad \mid {}^{w}/_{4} \le l \wedge r \le 3 \times {}^{w}/_{4} & = & \mathit{expand}\ (n{+}1, (2{\times}l - {}^{w}/_{2}, 2{\times}r - {}^{w}/_{2})) \\
\quad \mid \textbf{otherwise} & = & (n, (l, r)) \\
\mathit{nextBits}\ (n, (l, r)) & & \\
\quad \mid r \le {}^{w}/_{2} & = & \mathit{Just}\ (\mathit{bits}\ n\ 0, (0, (2{\times}l, 2{\times}r))) \\
\quad \mid {}^{w}/_{2} \le l & = & \mathit{Just}\ (\mathit{bits}\ n\ 1, (0, (2{\times}l - w, 2{\times}r - w))) \\
\quad \mid \textbf{otherwise} & = & \mathit{Nothing}
\end{array}$$

4.5. A final problem

Notwithstanding everything that has gone before, encoding is *not* guaranteed to work with any form of limited-precision arithmetic!!

Can you see why?

4.6. Decoding in the integer version

To see how to perform decoding with limited-precision arithmetic, we need some preliminary investigation.

Suppose *bs* is the output of *encode* when started off with an initial interval *ei* on a symbol sequence $s : ss$.

Setting $x = w \times \mathit{fromBits}\ bs$, we know that x is a fractional number satisfying

$$x\ \underline{\mathit{within}}\ \mathit{contract}\ (\mathit{enarrow}\ ei\ (\mathit{encodeSym}\ m\ s))$$

How can we calculate s given x, m and ei? We need to do this in order to define *extract*.

BUG!

4.7. Another Stream Inversion Theorem

Theorem. Let $\mathit{process}\ z = \mathit{foldr}\ (\oplus)\ e \cdot \mathit{stream}\ (\mathit{evolve}\ f)\ g\ z$.

Suppose $\ominus$ and *extract* can be found so that

1. $(y \oplus w) \ominus y = w$, for all y and w.
2. $w = \mathit{process}\ z\ (x : xs) \wedge f\ z = \mathit{Nothing} \Rightarrow \mathit{extract}\ (w, z) = x$, for all w, z, x and xs.

Then $xs\ \underline{\mathit{begins}}\ \mathit{unfoldr}\ \mathit{nextx}\ (\mathit{process}\ z\ xs, z)$, where

$$\begin{array}{lcl} \mathit{nextx}\ (w, z) & = & \mathit{Just}\ (x, (w', z')) \\ & & \quad \textbf{where}\ x = \mathit{extract}\ (w, z) \\ & & \quad \quad (ys, z') = \mathit{evolve}\ f\ (g\ z\ x) \\ & & \quad \quad w' = \mathit{foldl}\ (\ominus)\ w\ ys \end{array}$$

Floor calculations

The following property characterizes floors: for all integers n and fractions x,

$$n \le \lfloor x \rfloor \quad \equiv \quad n \le x$$

Ignorance of this simple rule has marred every published paper on arithmetic coding that we have read.

The calculation

$$\begin{array}{ll}
 & x\ \underline{within}\ contract\ (enarrow\ ei\ (encodeSym\ m\ s)) \\
\equiv & \quad \{\text{setting } (n,(l,r)) = expand\ ei\ \} \\
 & x\ \underline{within}\ (contract\ (n,(l,r) \blacktriangleright encodeSym\ m\ s)) \\
\equiv & \quad \{\text{setting } y = scale\ (n,x)\ \} \\
 & y\ \underline{within}\ ((l,r) \blacktriangleright encodeSym\ m\ s) \\
\equiv & \quad \{\text{setting } (p,q,d) = encodeSym\ m\ s\ \} \\
 & l + \lfloor (r-l) \times p/d \rfloor \le y < l + \lfloor (r-l) \times q/d \rfloor \\
\equiv & \quad \{\text{arithmetic}\} \\
 & \lfloor (r-l) \times p/d \rfloor \le y - l < \lfloor (r-l) \times q/d \rfloor \\
\equiv & \quad \{\text{rule of floors, setting } k = \lfloor y \rfloor\} \\
 & \lfloor (r-l) \times p/d \rfloor \le k - l < \lfloor (r-l) \times q/d \rfloor
\end{array}$$

The calculation, continued

$$\begin{array}{ll}
 & \lfloor (r-l) \times p/d \rfloor \le k - l < \lfloor (r-l) \times q/d \rfloor \\
\equiv & \quad \{\text{arithmetic}\} \\
 & \lfloor (r-l) \times p/d \rfloor < k - l + 1 \le \lfloor (r-l) \times q/d \rfloor \\
\equiv & \quad \{\text{rule of floors}\} \\
 & (r-l) \times p/d < k - l + 1 \le (r-l) \times q/d \\
\equiv & \quad \{\text{arithmetic}\} \\
 & p \le ((k-l+1) \times d - 1) / (r-l) < q \\
\equiv & \quad \{\text{rule of floors}\} \\
 & p \le \lfloor ((k-l+1) \times d - 1) / (r-l) \rfloor < q
\end{array}$$

Defining *extract*

Hence, redefining *decodeSym* to have type $Model \to Int \to Symbol$, we have

$$\begin{array}{lll}
\multicolumn{3}{l}{extract\ (bs,(m,ei)) = decodeSym\ m\ t} \\
\quad \textbf{where}\ t & = & ((k-l+1) \times d - 1)\ \underline{div}\ (r-l) \\
\quad\quad d & = & denom\ m \\
\quad\quad k & = & \lfloor scale\ (n, w \times fromBits\ bs) \rfloor \\
\quad\quad (n,(l,r)) & = & expand\ ei
\end{array}$$

But we do not need to inspect all of *bs* to compute *k*; we only need the first $n + e$ bits, where $w = 2^e$.

This is the clue to making *decode* incremental.

4.8. A final data refinement

Suppose we represent bs by a pair (x, ds) where $x :: Int$ and $ds :: [Bit]$ are defined by

$$\begin{aligned} x &= foldl\ (\lambda\ x\ b \rightarrow 2 \times x + b)\ 0\ cs \\ (cs, ds) &= splitAt\ e\ (bs +\!\!+ 1 : replicate\ (e-1)\ 0) \end{aligned}$$

Recalling that $w = 2^e$ and

$$fromBits\ bs = toFrac\ 0\ (bs +\!\!+ 1 : replicate\ n\ 0)$$

for any n, we have

$$w \times fromBits\ bs = x + toFrac\ 0\ ds$$

In particular, $x = \lfloor w \times fromBits\ bs \rfloor$.

Redefinition of *extract*

Recalling that $scale\ (n, x) = 2^n \times (x - {}^w\!/_2) + {}^w\!/_2$, we can compute $\lfloor scale\ (n, w \times fromBits\ bs) \rfloor$ by computing $fscale\ (n, (x, ds))$, where

$$\begin{aligned} &fscale\ (n, (x, ds)) = \\ &\quad foldl\ (\lambda\ x\ b \rightarrow 2 \times x + b - {}^w\!/_2)\ x\ (take\ n\ ds) \end{aligned}$$

Hence

$$\begin{aligned} extract\ ((x, ds), (m, ei)) &= decodeSym\ m\ t \\ \textbf{where}\ t &= ((k - l + 1) \times d - 1)\ \underline{div}\ (r - l) \\ d &= denom\ m \\ k &= fscale\ (n, (x, ds)) \\ (n, (l, r)) &= expand\ ei \end{aligned}$$

4.9. An alternative

The representation of bs by (x, ds) can be computed another way, though it involves inspecting all the bs:

$$\begin{aligned} (x, ds) &= foldr\ step\ ({}^w\!/_2, [\,])\ bs \\ step\ b\ (x, ds) &= (y, d : ds) \\ &\quad \textbf{where}\ (y, d) = (x + b \times w)\ \underline{divMod}\ 2 \end{aligned}$$

The point of this alternative is that we have

$$foldr\ step\ ({}^w\!/_2, [\,]) \cdot concat = foldr\ (\oplus)\ ({}^w\!/_2, [\,])$$

where $bs \oplus (x, ds) = foldr\ step\ (x, ds)\ bs$.

We leave the proof as an exercise.

Preparing for inversion

Moreover, we can invert $\oplus$. Define $\ominus$ by

$$\begin{aligned} (x, ds) \ominus bs &= foldl\ unstep\ (x, ds)\ bs \\ unstep\ (x, ds)\ b &= (2 \times x - b \times w + head\ ds, tail\ ds) \end{aligned}$$

Then we have

$$\begin{aligned} &(bs \oplus (x, ds)) \ominus bs \\ = &\quad \{\text{definitions of } \oplus \text{ and } \ominus\} \\ &foldl\ unstep\ (foldr\ step\ (x, ds)\ bs)\ bs \\ = &\quad \{\text{since } unstep\ (step\ b\ (x, ds))\ b = (x, ds)\} \\ &(x, ds) \end{aligned}$$

Having defined *extract* and $\ominus$ we can now define *decode*.

4.10. Defining *decode*

$$
\begin{aligned}
& \mathit{decode}\ m\ bs = \\
& \quad \mathit{unfoldr}\ \mathit{nextSym}\ ((x, ds), (m, \mathit{unit})) \\
& \quad \textbf{where}\ x = \mathit{foldl}\ (\lambda\ x\ b \rightarrow 2 \times x + b)\ 0\ cs \\
& \qquad\quad (cs, ds) = \mathit{splitAt}\ e\ (bs \mathbin{+\!\!+} 1 : \mathit{replicate}\ (e-1)\ 0)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{nextSym}\ ((x, ds), (m, ei)) = \\
& \quad \mathit{Just}\ (s, ((x', ds'), (\mathit{newModel}\ m\ s, ei'))) \\
& \quad \textbf{where}\ s = \mathit{extract}\ ((x, ds), (m, ei)) \\
& \qquad\quad (x', ds') = \mathit{foldl}\ (\ominus)\ (x, ds)\ bss \\
& \qquad\quad (bss, ei') = \mathit{evolve}\ \mathit{nextBits}\ (\mathit{enarrow}\ ei\ (\mathit{encodeSym}\ m\ s))
\end{aligned}
$$

4.11. Final conclusions

- There is a lot of arithmetic in Arithmetic Coding — not only the arithmetic of fractions and integers, but also the arithmetic of folds and unfolds.
- To the best of our knowledge, no previous description of Arithmetic Coding has ever tackled the formal basis for why the method works.
- Functional programming provides a convenient vehicle for expressing and reasoning about the various algorithms.
- Structured recursion operators, such as *foldl*, *unfoldr* and *stream*, are ubiquitous.

Selected references

- P. G. Howard and J. S. Vitter. Arithmetic coding for data compression. *Proc. IEEE*, Vol 82, 6, 857–865, 1994.
- J. Jiang. Novel design of arithmetic coding for data compression. *IEE Proc. Comput. Dig. Tech.*, Vol 142, 6 (Nov) 419–424, 1995.
- A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. on Inf. Systems* Vol 16, No 3, 256–294, July 1998.
- I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *C. ACM*, Vol 30, No 6, 520–540, June 1987.