Arithmetic coding with folds and unfolds

Richard Bird and Jeremy Gibbons

Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

1 Introduction

Arithmetic coding is a method for data compression. Although the idea was developed in the 1970's, it wasn't until the publication of an "accessible implementation" [14] that it achieved the popularity it has today. Over the past ten years arithmetic coding has been refined and its advantages and disadvantages over rival compression schemes, particularly Huffman [9] and Shannon-Fano [5] coding, have been elucidated. Arithmetic coding produces a theoretically optimal compression under much weaker assumptions than Huffman and Shannon-Fano, and can compress within one bit of the limit imposed by Shannon's Noiseless Coding Theorem [13]. Additionally, arithmetic coding is well suited to adaptive coding schemes, both character and word based. For recent perspectives on the subject, see [10, 12].

The "accessible implementation" of [14] consisted of a 300 line C program, and much of the paper was a blow-by-blow description of the workings of the code. There was little in the way of proof of why the various steps in the process were correct, particularly when it came to the specification of precisely what problem the implementation solved, and the details of why the inverse operation of decoding was correct. This reluctance to commit to specifications and correctness proofs seems to be a common feature of most papers devoted to the topic. Perhaps this is not surprising, because the plain fact is that arithmetic coding is tricky. Nevertheless, our aim in these lectures is to provide a formal derivation of basic algorithms for coding and decoding.

Our development of arithmetic coding makes heavy use of the algebraic laws of folds and unfolds. Although much of the general theory of folds and unfolds is well-known, see [3, 6], we will need one or two novel results. One concerns a new pattern of computation, which we call *streaming*. In streaming, elements of an output list are produced as soon as they are determined. This may sound like lazy evaluation but it is actually quite different.

2 Arithmetic coding, informally

Arithmetic coding is simple in theory but, as we said above, tricky to implement in practice. The basic idea is to:

1. Break the source message into *symbols*, where a symbol is some logical grouping of characters (or perhaps just a single character).

2. Associate each distinct symbol with a semi-open *interval* of the unit interval [0..1).
3. Successively *narrow* the unit interval by an amount determined by the interval associated with each symbol in the message.
4. Represent the final interval by choosing some *fraction* within it.

We can capture the basic datatypes and operations in Haskell by defining

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

Except where otherwise stated, we assume throughout that $0 \leq l < r \leq 1$ for every $(l, r) :: \mathit{Interval}$, so all intervals are subintervals of the unit interval. The code above gives a concrete implementation of *pick*, but all we really require is that

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

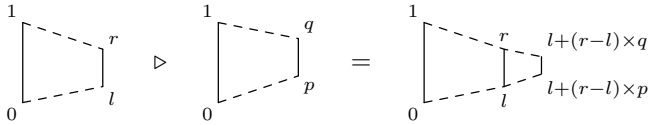
(We use underlining to turn a prefix function into an infix binary operator; this would be written `` `within` `` in Haskell.)

2.1 Narrowing

The operation of *narrowing* takes two intervals i and j and returns a subinterval k of i such that k is in the same relationship to i as j is to the unit interval:

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

Diagrammatically, we have:



Exercise 1. Prove that $x\ \underline{\mathit{within}}\ (\mathit{int}_1 \triangleright \mathit{int}_2) \Rightarrow x\ \underline{\mathit{within}}\ \mathit{int}_1$.

Exercise 2. Show that $\triangleright$ is associative with identity *unit*. Is $\triangleright$ commutative?

Exercise 3. Define an inverse $\triangleleft$ ('widen') of $\triangleright$ such that $(\mathit{int}_1 \triangleright \mathit{int}_2) \triangleleft \mathit{int}_1 = \mathit{int}_2$.

Exercise 4. Define the notion of the *reciprocal* i^{-1} of an interval i, such that

$$i \triangleright i^{-1} = \mathit{unit} = i^{-1} \triangleright i$$

(The reciprocal of a sub-unit interval will in general not itself be a sub-unit.) Redefine widening in terms of narrowing and reciprocal.

2.2 Models

In order to encode a message, each symbol has to be associated with a given interval. For our purposes, *Model* is an abstract type representing a finite mapping from *Symbol*s to *Interval*s with associated functions:

$$\begin{aligned}&\mathit{encodeSym} :: \mathit{Model} \to \mathit{Symbol} \to \mathit{Interval}\\&\mathit{decodeSym} :: \mathit{Model} \to \mathit{Fraction} \to \mathit{Symbol}\end{aligned}$$

We assume that the intervals associated with symbols do not overlap: for any $m :: \mathit{Model}$ and $x :: \mathit{Fraction}$,

$$s = \mathit{decodeSym}\ m\ x \equiv x\ \underline{\mathit{within}}\ (\mathit{encodeSym}\ m\ s)$$

Rather than having a single fixed model for the whole message, we allow the possibility that the model can change as the message is read; such a scheme is called *adaptive*. For instance, one can begin with a simple model in which symbols are associated via some standard mapping with intervals of the same size, and then let the model adapt depending on the actual symbols read. Therefore we also assume the existence of a function

$$\mathit{newModel} :: \mathit{Model} \to \mathit{Symbol} \to \mathit{Model}$$

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*. The idea of an adaptive model is not just a useful refinement on the basic scheme, but also an essential component in the derivation of the final algorithms.

Exercise 5. Specify the stronger condition that the intervals associated with symbols *partition* the unit interval.

2.3 Encoding

Having defined the relevant datatypes and auxiliary operations we can now define arithmetic encoding, which is to compute $\mathit{encode}_0\ m\ \mathit{unit}$, where

$$\begin{aligned}&\mathit{encode}_0 && :: \mathit{Model} \to \mathit{Interval} \to [\mathit{Symbol}] \to \mathit{Fraction}\\&\mathit{encode}_0\ m\ \mathit{int} && = \mathit{pick} \cdot \mathit{foldl}\ (\triangleright)\ \mathit{int} \cdot \mathit{encodeSyms}\ m\\&\mathit{encodeSyms} && :: \mathit{Model} \to [\mathit{Symbol}] \to [\mathit{Interval}]\\&\mathit{encodeSyms}\ m\ ss && = \mathit{unfoldr}\ \mathit{nextInt}\ (m, ss)\\&\mathit{nextInt} && :: (\mathit{Model}, [\mathit{Symbol}]) \to\\&&& \quad \mathit{Maybe}\ (\mathit{Interval}, (\mathit{Model}, [\mathit{Symbol}]))\\&\mathit{nextInt}\ (m, [\,]) && = \mathit{Nothing}\\&\mathit{nextInt}\ (m, s : ss) && = \mathit{Just}\ (\mathit{encodeSym}\ m\ s, (\mathit{newModel}\ m\ s, ss))\end{aligned}$$

The function $\mathit{encodeSyms}\ m$ uses the initial model m to encode the symbols of the message as intervals. These intervals are then used to narrow the unit interval to some final interval from which some number is chosen. The code makes use of the standard Haskell higher-order operators *foldl* and *unfoldr*, which are discussed in more detail in the following section.

2.4 Decoding

What remains is the question of how to perform the inverse operation of arithmetic decoding. Rather than give a program, we will give a non-executable specification. The function $decode_0 :: Model \to Interval \to Fraction \to [Symbol]$ is specified by

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

for all ss, where $xs \ \underline{begins} \ ys$ if $ys = xs +\!\!+ xs'$ for some xs'. Thus $decode_0$ is inverse to $encode_0$ in the sense that it is required to produce the sequence of symbols that $encode_0$ encodes but is not required to stop after producing them. Termination is handled separately. Provided we record the number of symbols in the message, or ensure that it ends with a special end-of-message symbol that occurs nowhere else, we can stop the decoding process at the right point.

Exercise 6. The Haskell definition of $begins :: Eq \ \alpha \to [\alpha] \to [\alpha] \to Bool$ is

$$\begin{array}{ll} [\,] \ \underline{begins} \ ys & = True \\ (x : xs) \ \underline{begins} \ [\,] & = False \\ (x : xs) \ \underline{begins} \ (y : ys) & = (x == y \wedge xs \ \underline{begins} \ ys) \end{array}$$

What is the value of $[\,] \ \underline{begins} \ \bot$?.

Exercise 7. What are the advantages and disadvantages of the two schemes (returning the length of the message, or making use of a special end-of-message symbol) for determining when to stop decoding?

2.5 Remaining refinements

Simple though $encode_0$ is, it will not suffice in a viable implementation and this is where the complexities of arithmetic coding begin to emerge. Specifically:

- we really want an encoding function that returns a list of bits (or bytes) rather than a number, not least because —
- for efficiency both in time and space, encoding should produce bits as soon as they are known (this is known as *incremental* transmission, or *streaming*);
- consequently, decoding should be implemented as a function that consumes bits and produces symbols, again in as incremental a manner as possible;
- for efficiency both in time and space, we should replace computations on fractions (pairs of arbitrary precision integers) with computations on fixed-precision integers, accepting that the consequent loss of accuracy will degrade the effectiveness of compression;
- we have to choose a suitable representation of models.

All of the above, except the last, will be addressed in what follows. We warn the reader now that there is a lot of arithmetic in arithmetic coding, not just the arithmetic of numbers, but also of folds and unfolds.

3 Folds and unfolds

Let us now digress a little to recall some of the theory of folds and unfolds. We will return to and augment our understanding of these operators in subsequent sections.

The higher-order operator *foldl* iterates over a list, from left to right:

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

Thus, writing f as an infix operator $\oplus$, we have

$$\mathit{foldl}\ (\oplus)\ e\ [x, y, z] = ((e \oplus x) \oplus y) \oplus z$$

Dually, the higher-order operator *foldr* iterates over a list, from right to left:

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

Thus, $\mathit{foldr}\ (\oplus)\ e\ [x, y, z] = x \oplus (y \oplus (z \oplus e))$. The crucial fact about *foldr* is the following *universal property*: for a strict function h we have

$$h = \mathit{foldr}\ f\ e \equiv h\ [\,] = e \wedge h\ (x : xs) = f\ x\ (h\ xs)$$

There is a close relationship between *foldl* and *foldr*, captured in part by the following two theorems. As the names of the theorems suggest, we are not telling the whole story here.

Theorem 8 (First Duality Theorem [3]). *If f is associative with unit e, then $\mathit{foldl}\ f\ e\ xs = \mathit{foldr}\ f\ e\ xs$ for all finite lists xs.*

Theorem 9 (Third Homomorphism Theorem [7]). *If both $h = \mathit{foldl}\ f_1\ e$ and $h = \mathit{foldr}\ f_2\ e$, then there is an associative f with unit e such that $h = \mathit{foldr}\ f\ e$.*

From Theorem 8 and Exercise 2, we have

$$\mathit{foldl}\ (\triangleright)\ \mathit{unit} = \mathit{foldr}\ (\triangleright)\ \mathit{unit}$$

So why don't we use the arguably more familiar *foldr* to express arithmetic coding? The answer lies in the the following lemma, which turns out to be an essential step in obtaining a program for decoding:

Lemma 10.

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

where

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

This lemma shows how two computations, namely turning the sequence of symbols into a sequence of intervals and then combining that sequence of intervals into a single interval, can be *fused* into one. Fusion is perhaps the single most important general idea for achieving efficient computations. There is no equivalent lemma if we replace *foldl* by *foldr*.

Exercise 11. Using the universal property, prove the fusion theorem for *foldr*: provided h is a strict function, $h\ e = e'$ and $h\ (f\ x\ z) = f'\ x\ (h\ x)$ for every x and z, we have $h \cdot foldr\ f\ e = foldr\ f'\ e'$.

Exercise 12. By defining *map* as an instance of *foldr*, prove *map fusion*:

$$foldr\ f\ e \cdot map\ g = foldr\ (f \cdot g)\ e$$

Exercise 13. Why don't the universal property and the fusion theorem for *foldr* hold for non-strict h? Does the First Duality Theorem hold for infinite or partial lists?

Exercise 14. Suppose that $(x \oplus y) \ominus x = y$ for all x and y. Prove that

$$foldl\ (\ominus)\ (foldr\ (\oplus)\ x\ ys)\ ys = x$$

for all x and finite lists ys.

Exercise 15. 'Parallel loops' may also be fused into one: if

$$h\ xs = (foldr\ f_1\ e_1\ xs, foldr\ f_2\ e_2\ xs)$$

then $h = foldr\ f\ (e_1, e_2)$, where $f\ x\ (z_1, z_2) = (f_1\ x\ z_1, f_2\ x\ z_2)$. For example,

$$average = uncurry\ div \cdot sumlength$$

where $sumlength\ xs = (sum\ xs, length\ xs)$, and *sumlength* can be written with a single *foldr*. Parallel loop fusion is sometimes known as the 'Banana Split Theorem' (because, in days of old, folds were often written using "banana" brackets; see, for example, [4]). Prove the theorem, again using the universal property of *foldr*.

Exercise 16. The function *foldl* can be expressed in terms of *foldr*:

$$foldl\ f = flip\ (foldr\ (comp\ f)\ id)\ \textbf{where}\ comp\ f\ x\ u = u \cdot flip\ f\ x$$

Verify this claim, and hence (from the universal property of *foldr*) derive the following universal property of *foldl*: for h strict in its second argument,

$$h = foldl\ f\ \equiv\ h\ e\ [\,] = e \wedge h\ e\ (x : xs) = h\ (f\ e\ x)\ xs$$

3.1 Unfolds

To describe unfolds first recall the Haskell standard type *Maybe*:

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Just}\ \alpha \mid \mathit{Nothing}$$

The function *unfoldr* is defined by

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

For example, the standard Haskell prelude function *enumFromTo* is very nearly given by *curry* (*unfoldr next*), where

$$\mathit{next}\ (a, b) = \textbf{if}\ a \leq b\ \textbf{then}\ \mathit{Just}\ (a, (\mathit{succ}\ a, b))\ \textbf{else}\ \mathit{Nothing}$$

(Only 'very nearly' because membership of the type class *Enum* does not actually imply membership of *Ord* in Haskell; the comparison is done instead by using *fromEnum* and comparing the integers.)

The Haskell Library Report [2] states:

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

That's essentially all the Report says on unfolds! We will have more to say about them later on.

3.2 Hylomorphisms

One well-known pattern involving folds and unfolds is that of a *hylomorphism* [11], namely a function h whose definition takes the form

$$h = \mathit{foldr}\ f\ \ e \cdot \mathit{unfoldr}\ \ g$$

The two component computations have complementary structures and they can be fused into one:

$$\begin{aligned}&h\ \ z = \textbf{case}\ g\ \ z\ \textbf{of}\\ &\qquad\qquad \mathit{Nothing} \quad\ \to e\\ &\qquad\qquad \mathit{Just}\ (x, z') \to f\ \ x\ \ (h\ \ z')\end{aligned}$$

This particular rule is known as *deforestation* because the intermediate data structure (in this case a list, but in a more general form of hylomorphism it could be a tree) is removed.

4 Producing bits

Let us now return to arithmetic coding. As we noted above, we would like encoding to return a list of bits rather than a number. To achieve this aim we replace the function $pick :: Interval \to Fraction$ by two functions

$$\begin{array}{l} \textbf{type}\ Bit = Int \qquad -- \quad 0 \text{ and } 1 \text{ only} \\ toBits \quad :: Interval \to [Bit] \\ fromBits \ :: [Bit] \to Fraction \end{array}$$

such that $pick = fromBits \cdot toBits$. Equivalently, for all intervals int, we require

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

The 'obvious' choices here are to let $toBits\ (l, r)$ return the *shortest* binary fraction x satisfying $l \le x < r$, and $fromBits$ return the value of the binary fraction. Thus, $fromBits = foldr\ pack\ 0$, where $pack\ b\ x = (b + x)/2$. However, as Exercises 25 and 26 explain, we reject the obvious definitions and take instead

$$\begin{array}{l} fromBits = foldr\ pack\ (1/2) \\ toBits \quad = unfoldr\ nextBit \end{array}$$

where

$$\begin{array}{l} nextBit \qquad\qquad :: Interval \to Maybe\ (Bit, Interval) \\ nextBit\ (l, r) \\ \quad \mid r \le 1/2 \qquad = Just\ (0, (2 \times l, 2 \times r)) \\ \quad \mid 1/2 \le l \qquad = Just\ (1, (2 \times l - 1, 2 \times r - 1)) \\ \quad \mid \textbf{otherwise} = Nothing \end{array}$$

Exercise 17. Give an equivalent definition of *nextBit* in terms of narrowing by non-sub-unit intervals.

We leave it as an exercise to show

$$foldr\ pack\ (1/2)\ bs = foldr\ pack\ 0\ (bs +\!\!+ [1])$$

Thus $fromBits\ bs$ returns the binary fraction obtained by adding a final 1 to the end of bs. The definition of *toBits* has a simple reading: if $r \le 1/2$, then the binary expansion of any fraction x such that $l < x < r$ begins with 0; and if $1/2 \le l$, the expansion of x begins with 1. In the remaining case $l < 1/2 < r$ the empty sequence is returned.

Proposition 18. $\quad length\ (toBits\ (l, r)) \le -\log_2(r - l)$

In particular, *toBits* always yields a finite list given a non-empty interval.

Proof. The function *toBits* applied to an interval of width greater than a half yields the empty sequence of bits:

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

Moreover, each iteration of *nextBit* doubles the width of the interval. So if $1/2^{n+1} < r - l \le 1/2^n$ or, equivalently, $n \le -\log_2(r - l) < n + 1$, then termination is guaranteed after at most n bits have been produced.

Proposition 19. *fromBits* (*toBits int*) *within int*

Proof. The function $pick = fromBits \cdot toBits$ is a hylomorphism, so we obtain

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

The proof now follows by appeal to fixpoint induction.

Exercise 20. Show that $foldr\ pack\ (1/2)\ bs = foldr\ pack\ 0\ (bs +\!\!+ [1])$.

Exercise 21. Show that

$$\begin{array}{ll} (2\times l, 2\times r) & = (0, 1/2) \triangleleft (l,r) \\ (2\times l - 1, 2\times r - 1) & = (1/2, 1) \triangleleft (l,r) \end{array}$$

Exercise 22. Show that

$$\begin{array}{l} fromBits\ bs = mean\ (foldr\ pack\ 0\ bs, foldr\ pack\ 1\ bs) \\ \quad \textbf{where}\ mean\ (x,y) = (x+y)/2 \end{array}$$

Exercise 23. Show that

$$\begin{array}{l} (foldr\ pack\ 0\ bs, foldr\ pack\ 1\ bs) = foldl\ (\triangleright)\ unit\ (map\ encodeBit\ bs) \\ \quad \textbf{where}\ encodeBit\ b = (b/2, (b{+}1)/2) \end{array}$$

Exercise 24. One might expect *toBits* (l,r) to yield the *shortest* binary fraction within $[l..r)$, but in fact it does not. What definition does?

Exercise 25. The reason we do not use the shortest binary fraction as the definition of *toBits* is that the *streaming* condition of Section 5.1 fails to hold with this definition. After studying that section, justify this remark.

Exercise 26. Since we are using intervals that are closed on the left, one might expect that guard in the second clause of *nextBit* would be $1/2 < l$. However, with this definition of *fromBits*, the result of Exercise 42 in Section 7 fails to hold. After studying that section, justify this remark.

4.1 Summary of first refinement

Drawing together the results of this section, we define

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

The new version of encoding yields a bit sequence rather than a fraction. However, execution of $encode_1$ still consumes all its input before delivering any output. Formally, $encode_1\ m\ ss = \bot$ for all partial or infinite lists ss. Can we do better?

5 Streaming

The function $encode_1$ consists of an *unfoldr* after a *foldl*. Even under lazy evaluation, the *foldl* consumes all its input before the *unfoldr* can start producing output. For efficiency, we would prefer a definition that is capable of yielding some output as soon as possible.

To this end, we introduce a new higher-order operator *stream*, which alternates between production and consumption. This function has type

$$\begin{aligned} &stream :: (state \to Maybe\ (output, state)) \to \\ &\qquad (state \to input \to state) \to \\ &\qquad state \to [input] \to [output] \end{aligned}$$

and is defined by

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

The function *stream* describes a process that alternates between producing output and consuming input. Starting in state z, control is initially passed to the producer function f, which delivers output until no more can be produced. Control is then passed to the consumer process g, which consumes the next input x and delivers a new state. The cycle then continues until the input is exhausted.

Exercise 27. Define a variant *stream* that alternates between production and consumption but hands control to the consumer process first.

5.1 The Streaming Theorem

The relationship between *stream* and folds and unfolds hinges on the following definition:

Definition 28. *The* streaming condition *for f and g is*

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

for all z, y, z' and x.

The streaming condition states very roughly that f is invariant under g. By induction we can then conclude that f is invariant under repeated applications of g; this is the content of the following lemma:

Lemma 29. *If the streaming condition holds for f and g, then*

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

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

Proof. The proof is by induction on xs:

Case $[\,]$**:** Immediate.
Case $x : xs$**:** Assume $f\ z = \mathit{Just}\ (y, z')$, so by the streaming condition we have $f\ (g\ z\ x) = \mathit{Just}\ (y, g\ z'\ x)$. Now we reason

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

Now we come to the crunch.

Theorem 30. *If the streaming condition holds for f and g, then*

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

for all z and all finite *lists xs.*

The proof of Theorem 30 uses the following lemma, which states how to prove that two potentially infinite lists are equal (see [3, §9.3]).

Lemma 31. *Define approx by*

$$\begin{aligned}
& \mathit{approx} :: \mathit{Integer} \to [\alpha] \to [\alpha] \\
& \mathit{approx}\ (n+1)\ [\,] \quad\ = [\,] \\
& \mathit{approx}\ (n+1)\ (x : xs) = x : \mathit{approx}\ n\ xs
\end{aligned}$$

Then two arbitrary lists xs and ys are equal iff approx n xs = approx n ys for all n.

Proof (of Theorem 30). We use a double induction on n and xs to show that, provided that the streaming condition holds for f and g,

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

for all n, z and finite lists xs. The first step is case analysis on n.

Case 0**:** Immediate since $\mathit{approx}\ 0\ xs = \bot$ for any xs.
Case $n+1$**:** In this case we perform an analysis on $f\ z$:
Subcase $f\ z = \mathit{Just}\ (y, z')$**:** We reason

$$\begin{aligned}
& \mathit{approx}\ (n+1)\ (\mathit{unfoldr}\ f\ (\mathit{foldl}\ g\ z\ xs)) \\
= \quad & \{\text{applying Lemma 29}\} \\
& \mathit{approx}\ (n+1)\ (y : \mathit{unfoldr}\ f\ (\mathit{foldl}\ g\ z'\ xs)) \\
= \quad & \{\text{definition of } \mathit{approx}\} \\
& y : \mathit{approx}\ n\ (\mathit{unfoldr}\ f\ (\mathit{foldl}\ g\ z'\ xs)) \\
= \quad & \{\text{induction}\} \\
& y : \mathit{approx}\ n\ (\mathit{stream}\ f\ g\ z'\ xs) \\
= \quad & \{\text{definition of } \mathit{approx}\} \\
& \mathit{approx}\ (n+1)\ (y : \mathit{stream}\ f\ g\ z'\ xs) \\
= \quad & \{\text{definition of } \mathit{stream}\} \\
& \mathit{approx}\ (n+1)\ (\mathit{stream}\ f\ g\ z\ xs)
\end{aligned}$$

Subcase $f\ z = Nothing$**:** Now we need a case analysis on xs. The case of the empty list is immediate since both sides reduce to $[\,]$. In the remaining case we reason

$$\begin{array}{ll} & approx\ (n+1)\ (unfoldr\ f\ (foldl\ g\ z\ (x : xs))) \\ = & \quad \{\text{definition of } foldl\} \\ & approx\ (n+1)\ (unfoldr\ f\ (foldl\ g\ (g\ z\ x)\ xs)) \\ = & \quad \{\text{induction}\} \\ & approx\ (n+1)\ (stream\ f\ g\ (g\ z\ x)\ xs) \\ = & \quad \{\text{definition of } stream\} \\ & approx\ (n+1)\ (stream\ f\ g\ z\ (x : xs)) \end{array}$$

This completes the induction and the proof.

Exercise 32. Show that the streaming condition holds for *unCons* and *snoc*, where

$$\begin{array}{lll} unCons\ [\,] & = & Nothing \\ unCons\ (x : xs) & = & Just\ (x, xs) \\ snoc\ x\ xs & = & xs +\!\!+ [x] \end{array}$$

Exercise 33. What happens to the streaming theorem for partial or infinite lists?

Exercise 34. Recall that

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

Show that streaming condition for *nextBit* and $\triangleright$ follows from associativity of $\triangleright$ (Exercise 2) and the fact that $int_1 \triangleright int_2$ is contained in int_1 (Exercise 1).

5.2 Summary of second refinement

At the end of Section 4.1, we had

$$\begin{array}{l} encode_1 \quad :: Model \to Interval \to [Symbol] \to [Bit] \\ encode_1\ m\ int = unfoldr\ nextBit \cdot foldl\ (\triangleright)\ int \cdot encodeSyms\ m \end{array}$$

Since Exercise 34 established the streaming condition for *nextBit* and $\triangleright$, we can define

$$\begin{array}{l} encode_2 \quad :: Model \to Interval \to [Symbol] \to [Bit] \\ encode_2\ m\ int = stream\ nextBit\ (\triangleright)\ int \cdot encodeSyms\ m \end{array}$$

Although $encode_1 \neq encode_2$, the two functions *are* equal on all finite symbol sequences, which is all we require.

6 Decoding and stream inversion

The function $decode_2 :: Model \to Interval \to [Bit] \to [Symbol]$ corresponding to $encode_2$ is specified by

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

for all finite sequences of symbols ss.

To implement $decode_2$ we have somehow to invert streams. We will make use of a function *destream* with type

$$\begin{aligned} destream :: & (state \to Maybe \ (output, state)) \to \\ & (state \to input \to state) \to \\ & (state \to [output] \to input) \to \\ & state \to [output] \to [input] \end{aligned}$$

The definition of *destream* is

$$\begin{aligned} & destream \ f \ g \ h \ z \ ys = \\ & \ \mathbf{case} \ f \ z \ \mathbf{of} \\ & \qquad Just \ (y, z') \to destream \ f \ g \ h \ z' \ (ys \ \underline{after} \ y) \\ & \qquad Nothing \quad \to x : destream \ f \ g \ h \ (g \ z \ x) \ ys \\ & \ \mathbf{where} \ x = h \ z \ ys \end{aligned}$$

The operator $\underline{after}$ is partial:

$$ys \ \underline{after} \ y = \mathbf{if} \ head \ ys = y \ \mathbf{then} \ tail \ ys \ \mathbf{else} \ \bot$$

The function *destream* is dual to *stream*: when $f \ z$ produces something, an element of the input is consumed; when $f \ z$ produces nothing, an element of the output is produced using the helper function h. Note that *destream* always produces a partial or infinite list, never a finite one.

The relationship between *stream* and *destream* is given by the following theorem:

Theorem 35. *Suppose the following implication holds for all z, x and xs:*

$$f \ z = Nothing \Rightarrow h \ z \ (stream \ f \ g \ z \ (x : xs)) = x$$

Then, provided stream $f \ g \ z \ xs$ returns a finite list, we have

$$xs \ \underline{begins} \ destream \ f \ g \ h \ z \ (stream \ f \ g \ z \ xs)$$

Proof. The proof is by a double induction on xs and n, where n is the length of $stream \ f \ g \ z \ xs$.

Case $[\,]$**:** Immediate since $[\,]$ begins every list.

Case $x : xs$**:** We first consider the subcase $f\ z = Nothing$ (which includes the case $n = 0$):

$$\begin{array}{ll} & destream\ f\ g\ h\ z\ (stream\ f\ g\ z\ (x : xs)) \\ = & \quad \{\text{definition of } destream \text{ and } h\ z\ (stream\ f\ g\ (x : xs)) = x\} \\ & x : destream\ f\ g\ h\ (g\ z\ x)\ (stream\ f\ g\ z\ (x : xs)) \\ = & \quad \{\text{definition of } stream\} \\ & x : destream\ f\ g\ h\ (g\ z\ x)\ (stream\ f\ g\ (g\ z\ x)\ xs) \end{array}$$

Since $(x : xs)\ \underline{begins}\ (x : xs')$ if and only if $xs\ \underline{begins}\ xs'$, an appeal to induction establishes the case.
In the case $f\ z = Just\ (y, z')$, we have $n \neq 0$, and so $stream\ f\ g\ z'\ (x : xs)$ has length $n - 1$. We reason

$$\begin{array}{ll} & destream\ f\ g\ h\ z\ (stream\ f\ g\ z\ (x : xs)) \\ = & \quad \{\text{definition of } stream\} \\ & destream\ f\ g\ h\ z\ (y : stream\ f\ g\ z'\ (x : xs)) \\ = & \quad \{\text{definition of } destream\} \\ & destream\ f\ g\ h\ z'\ (stream\ f\ g\ z'\ (x : xs)) \end{array}$$

An appeal to induction establishes the case, completing the proof.

6.1 Applying the theorem

In order to apply the stream inversion theorem, recall Lemma 10 which states that $foldl\ (\triangleright)\ int \cdot encodeSyms\ m = snd \cdot foldl\ step\ (m, int)$ where

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

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

$$encode_2\ m\ int = unfoldr\ nextBitM \cdot foldl\ step\ (m, int)$$

where *nextBitM* is identical to *nextBit* except that it propagates the model as an additional argument:

$$\begin{array}{ll} nextBitM & :: (Model, Interval) \to Maybe\ (Bit, (Model, Interval)) \\ nextBitM\ (m, (l, r)) & \\ \quad \mid r \leq 1/2 & = Just\ (0, (m, (2 \times l, 2 \times r))) \\ \quad \mid 1/2 \leq l & = Just\ (1, (m, (2 \times l - 1, 2 \times r - 1))) \\ \quad \mid \textbf{otherwise} & = Nothing \end{array}$$

Theorem 30 is again applicable and we obtain the following alternative definition of $encode_2$:

$$encode_2\ m\ int = stream\ nextBitM\ step\ (m, int)$$

Now we are ready for stream inversion. Observe that $encode_2\ m\ int$ returns a finite bit sequence on all finite symbol sequences, so it remains to determine h.

Let $bs = encode_2\ m\ int\ (s : ss)$ and $x = fromBits\ bs$, so that

$$x\ \underline{within}\ (int \triangleright encodeSym\ m\ s)$$

We can now reason:

$$\begin{array}{ll} & x\ \underline{within}\ (int \triangleright encodeSym\ m\ s) \\ \equiv & \quad \{\text{with } int = (l, r) \text{ and } encodeSym\ m\ s = (p, q)\} \\ & l + (r - l) \times p \leq x < l + (r - l) \times q \\ \equiv & \quad \{\text{arithmetic}\} \\ & p \leq (x - l)/(r - l) < q \\ \equiv & \quad \{\text{definition of } decodeSym\} \\ & s = decodeSym\ m\ ((x - l)/(r - l)) \end{array}$$

Hence we can take

$$h\ (m, (l, r))\ bs = decodeSym\ m\ ((fromBits\ bs - l)/(r - l))$$

Putting these pieces together, we therefore obtain

$$\begin{array}{ll} decode_2\ m\ int & = destream\ nextBitM\ step\ nextSym\ (m, int) \\ nextSym\ (m, (l, r))\ bs & = decodeSym\ m\ ((fromBits\ bs - l)/(r - l)) \\ step\ (m, int)\ s & = (newModel\ m\ s, int \triangleright encodeSym\ m\ s) \end{array}$$

where *nextBitM* was defined above.

This is not a very efficient way to compute $decode_2$. Each computation of *fromBits bs* requires that the bit sequence *bs* is traversed in its entirety. Worse, this happens each time an output symbol is produced. Better is to fuse the computation of *fromBits* into *destream* so that the bit sequence is processed only once. We can do this fusion with a somewhat more complicated version of *destream*.

6.2 A better stream inversion theorem

Replace the previous function *destream* with a more general one, called *unstream*, with type

$$\begin{array}{l} unstream :: (state \to Maybe\ (output, state)) \to \\ \qquad\qquad (state \to input \to state) \to \\ \qquad\qquad (state \to result \to input) \to \\ \qquad\qquad (result \to output \to result) \to \\ \qquad\qquad state \to result \to [input] \end{array}$$

With six arguments this seems a complicated function, which is why we didn't give it earlier. The definition of *unstream* is

$$\begin{array}{l} unstream\ f\ g\ h\ k\ z\ w = \\ \quad \textbf{case}\ f\ z\ \textbf{of} \\ \qquad Just\ (y, z') \to unstream\ f\ g\ h\ k\ z'\ (k\ w\ y) \\ \qquad Nothing \quad\ \to x : unstream\ f\ g\ h\ k\ (g\ z\ x)\ w \\ \quad \textbf{where}\ x = h\ z\ w \end{array}$$

This more complicated definition is a generalisation, since *destream f g h z* is equivalent to *unstream f g h after z*. The relationship between *stream* and *unstream* is given by the following theorem, a generalisation of Theorem 35:

Theorem 36. *Let process z = foldr* $(\oplus)$ *w* $\cdot$ *stream f g z. Suppose that*

$$f\ z = Nothing \Rightarrow h\ z\ (process\ z\ (x : xs)) = x$$

for all z, x and xs. Furthermore, suppose that $\ominus$ *satisfies* $(y \oplus w) \ominus y = w$ *for all y and w. Then, provided stream f g z xs returns a finite list, we have*

$$xs\ \underline{begins}\ unstream\ f\ g\ h\ (\ominus)\ z\ (process\ z\ xs)$$

The proof is so similar to the earlier one that we can leave details as an exercise. The point of the new version is that, since $fromBits = foldr\ pack\ (1/2)$ where $pack\ b\ x = (b + x)/2$, we can define $\ominus = unpack$, where $unpack\ x\ b = 2 \times x - b$. As a consequence, we obtain

$$\begin{aligned} &decode_2\ m\ int\ bs = \\ &\quad unstream\ nextBitM\ step\ nextSym\ unpack\ (m, int)\ (fromBits\ bs) \end{aligned}$$

In this version the bit sequence *bs* is traversed only once. Nevertheless, $decode_2$ is not an incremental algorithm since all of *bs* has to be inspected before any output is produced.

Exercise 37. Following the steps of the proof of the first version of stream inversion, prove the second version of stream inversion.

Exercise 38. What substitutions for $\oplus$ and w in Theorem 36 yield Theorem 35?

7 Interval expansion

The major problem with $encode_2$ and $decode_2$ is that they make use of fractional arithmetic. In Section 8 we are going to replace fractional arithmetic by arithmetic with limited-precision integers. In order to do so we need a preparatory step: interval expansion. Quoting from Howard and Vitter [8]:

> 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 $1/4$.

For the moment we will just accept the fact that ensuring the width of the current interval is greater than $1/4$ before narrowing is an important step on the path to limited precision.

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

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

where $\mathit{scale}\ (n, x) = 2^n \times (x - 1/2) + 1/2$, subject to $0 \leq l' < r' \leq 1$. In particular, $(0, (l, r))$ is one possible representation of (l, r).

A *fully-expanded* interval for (l, r) is a triple $(n, (l', r'))$ in which n is as large as possible. Intervals straddling $1/2$ will be fully-expanded immediately before narrowing. The remainder of this section is devoted to installing this data refinement. More precisely, with *ei* denoting an expanded interval and *contract ei* the corresponding un-expanded interval, our aim is to provide suitable definitions that justify the following calculation:

$$\begin{array}{ll} & \mathit{toBits} \cdot \mathit{foldl}\ (\triangleright)\ \mathit{int} \\ = & \quad \{\text{assuming } \mathit{int} = \mathit{contract}\ \mathit{ei}\} \\ & \mathit{toBits} \cdot \mathit{foldl}\ (\triangleright)\ (\mathit{contract}\ \mathit{ei}) \\ = & \quad \{\text{fold-fusion (in reverse) for some function } \mathit{enarrow}\} \\ & \mathit{toBits} \cdot \mathit{contract} \cdot \mathit{foldl}\ \mathit{enarrow}\ \mathit{ei} \\ = & \quad \{\text{definition of } \mathit{toBits}\} \\ & \mathit{unfoldr}\ \mathit{nextBit} \cdot \mathit{contract} \cdot \mathit{foldl}\ \mathit{enarrow}\ \mathit{ei} \\ = & \quad \{\text{for some suitable definition of } \mathit{nextBits}\} \\ & \mathit{concat} \cdot \mathit{unfoldr}\ \mathit{nextBits} \cdot \mathit{foldl}\ \mathit{enarrow}\ \mathit{ei} \\ = & \quad \{\text{streaming}\} \\ & \mathit{concat} \cdot \mathit{stream}\ \mathit{nextBits}\ \mathit{enarrow}\ \mathit{ei} \end{array}$$

The function *enarrow* connotes "expand and narrow" and is an operation that first expands an interval before narrowing it. Given this motivating calculation, we can then define

$$\mathit{encode}_3\ m\ \mathit{ei} = \mathit{concat} \cdot \mathit{stream}\ \mathit{nextBits}\ \mathit{enarrow}\ \mathit{ei} \cdot \mathit{encodeSyms}\ m$$

Arithmetic coding is then implemented by the call $\mathit{encode}_3\ m\ (0, (0, 1))$. Note that composing *concat* with *stream* still gives incremental transmission because of laziness: the argument to *concat* does not have to be evaluated fully before results are produced.

7.1 Defining expand and contract

First, we give a definition of the function *expand* that expands intervals. Observe that

$$\begin{array}{l} 0 \leq 2 \times (l - 1/2) + 1/2 \ \equiv\ 1/4 \leq l \\ 2 \times (r - 1/2) + 1/2 \leq 1 \ \equiv\ r \leq 3/4 \end{array}$$

Hence we can further expand $(n, (l, r))$ if $1/4 \leq l$ and $r \leq 3/4$. This leads to the definition

$$\begin{array}{l} \mathit{expand}\ (n, (l, r)) \\ \quad \mid 1/4 \leq l \wedge r \leq 3/4 = \mathit{expand}\ (n{+}1, (2 \times l - 1/2, 2 \times r - 1/2)) \\ \quad \mid \textbf{otherwise} \qquad = (n, (l, r)) \end{array}$$

The function *nextBits*, to be defined in a short while, will return *Nothing* on intervals that straddle $\frac{1}{2}$. Consequently, in $encode_3$ we expand intervals (l, r) satisfying $l < \frac{1}{2} < r$ immediately before narrowing. It follows that narrowing is applied only when $l < \frac{1}{4}$ and $\frac{1}{2} < r$, or $l < \frac{1}{2}$ and $\frac{3}{4} < r$; in either case, $\frac{1}{4} < r - l$, which is the key inequality.

The converse of *expand* is given by

$$contract\ (n, (l, r)) = (rescale\ (n, l), rescale\ (n, r))$$

where $rescale\ (n, x) = (x - \frac{1}{2})/2^n + \frac{1}{2}$. We leave it as exercises to verify that

$$\begin{aligned} contract \cdot expand &= contract \\ contract\ (n, int1 \triangleright int2) &= contract\ (n, int1) \triangleright int2 \end{aligned}$$

Consequently, defining *enarrow* by

$$\begin{aligned} enarrow\ ei\ int2 = &(n, int1 \triangleright int2) \\ &\textbf{where}\ (n, int1) = expand\ ei \end{aligned}$$

we have $contract\ (enarrow\ ei\ int) = contract\ ei \triangleright int$. An appeal to fold-fusion therefore gives

$$contract \cdot foldl\ enarrow\ ei = foldl\ (\triangleright)\ (contract\ ei)$$

This identity was used in the motivating calculation above. The remaining step is to find some suitable definition of *nextBits* so that

$$toBits \cdot contract = concat \cdot unfoldr\ nextBits$$

and also that *nextBits* and *enarrow* satisfy the streaming condition.

The definition of *nextBits* turns out to be

$$\begin{aligned} &nextBits\ (n, (l, r)) \\ &\quad \mid r \leq \tfrac{1}{2} &&= Just\ (bits\ n\ 0, (0, (2 \times l, 2 \times r))) \\ &\quad \mid \tfrac{1}{2} \leq l &&= Just\ (bits\ n\ 1, (0, (2 \times l - 1, 2 \times r - 1))) \\ &\quad \mid \textbf{otherwise} &&= Nothing \end{aligned}$$

where $bits\ n\ b = b : replicate\ n\ (1-b)$ returns a b followed by a sequence of n copies of $1-b$. The proof that this definition satisfies all our requirements is left as an exercise.

Exercise 39. Verify that

$$\begin{aligned} contract \cdot expand &= contract \\ contract\ (n, int1 \triangleright int2) &= contract\ (n, int1) \triangleright int2 \end{aligned}$$

Why don't we have $contract \cdot expand = id$?

Exercise 40. Prove that

$$\begin{aligned} rescale\ (n, x) \leq \tfrac{1}{2} &\equiv x \leq \tfrac{1}{2} \\ rescale\ (n, x) \geq \tfrac{1}{2} &\equiv x \geq \tfrac{1}{2} \end{aligned}$$

Hence $contract\ (n, (l, r))$ straddles $\frac{1}{2}$ iff (l, r) does.

Exercise 41. Prove that

$$\begin{aligned} 2 \times \mathit{rescale}\ (n+1, x) &= \mathit{rescale}\ (n, x) + {}^1\!/_2 \\ 2 \times \mathit{rescale}\ (n+1, x) - 1 &= \mathit{rescale}\ (n, x) - {}^1\!/_2 \end{aligned}$$

Exercise 42. Prove by induction on n that

$$\begin{aligned} &\mathit{toBits}\ (2 \times \mathit{rescale}\ (n, l), 2 \times \mathit{rescale}\ (n, r)) = \\ &\quad = \mathit{replicate}\ n\ 1 +\!\!+ \mathit{toBits}\ (2 \times l, 2 \times r) \\ &\mathit{toBits}\ (2 \times \mathit{rescale}\ (n, l) - 1, 2 \times \mathit{rescale}\ (n, r) - 1) = \\ &\quad = \mathit{replicate}\ n\ 0 +\!\!+ \mathit{toBits}\ (2 \times l - 1, 2 \times r - 1) \end{aligned}$$

Exercise 43. Prove that if $l < {}^1\!/_2 < r$ then

$$\mathit{toBits}\ (\mathit{contract}\ (n, (l, r))) = \mathit{concat}\ (\mathit{unfoldr}\ \mathit{nextBits}\ (n, (l, r)))$$

Exercise 44. Prove that if $r \leq {}^1\!/_2$ then

$$\mathit{toBits}\ (\mathit{contract}\ (n, (l, r))) = \mathit{bits}\ n\ 0 +\!\!+ \mathit{toBits}\ (2 \times l, 2 \times r)$$

Similarly, prove that if ${}^1\!/_2 \leq l$ then

$$\mathit{toBits}\ (\mathit{contract}\ (n, (l, r))) = \mathit{bits}\ n\ 1 +\!\!+ \mathit{toBits}\ (2 \times l - 1, 2 \times r - 1)$$

Hence complete the proof of $\mathit{toBits} \cdot \mathit{contract} = \mathit{concat} \cdot \mathit{unfoldr}\ \mathit{nextBits}$.

Exercise 45. Verify that the streaming condition holds for *nextBits* and *enarrow*.

8 From fractions to integers

We now want to replace fractional arithmetic by arithmetic with limited-precision integers. In the final version of arithmetic coding, intervals take the form (l, r), where l and r are integers in the range $0 \leq l < r \leq w$ 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 $0 \leq p < q \leq d$ and d is an integer which is fixed for m and called the *denominator* for m. This triple represents the interval $({}^p\!/_d, {}^q\!/_d)$.

8.1 Integer narrowing

The narrowing function is redefined as follows:

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

Equivalently,

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

A reasonable step, you might think, but there are a number of problems with it:

- the revised definition of narrowing completely changes the specification: encoding will now produce different outputs than before and, 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 $\lfloor (r-l) \times {}^{p}/_{d} \rfloor = \lfloor (r-l) \times {}^{q}/_{d} \rfloor$.

The middle point seems the most damaging one, and is perhaps the reason that writers on arithmetic coding do not attempt to specify what problem arithmetic coding solves.

8.2 Change of specification

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

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

where

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

Hence, provided $p' < q'$, integer narrowing of an interval (l, r) by another interval (p, q) drawn from a model m can be viewed as fractional narrowing of (l, r) by the corresponding interval (p', q') drawn from an *adjusted* model *adjust* (l, r) m. Note that $p' \leq p$ and $q' \leq q$, so the effect of this adjustment is that some of the intervals shuffle down a little, leaving a little headroom at the top (see below for an example). We do not need to implement *adjust*; the important point is by invoking it at every step all of the previous development remains valid.

It is instructive to illustrate the adjustments made to the model. Consider Figure 1 in which $w = 64$ and $d = 10$. The columns on the left show a given sequence of models that might arise after processing symbols in the string ABAC. For example, the first row shows a model in which A is associated with the interval [0.0..0.3), B is associated with [0.3..0.6), and C with [0.6..1.0). The columns on the right show the corresponding adjusted intervals to three decimal places. The current intervals immediately before processing the next symbol are shown in the middle. The output of the integer implementation is 0010010, while that of the real implementation is 00100, so there is a deterioration in compression effectiveness even for this short string.

8.3 When intervals collapse

It is left as an exercise to show that

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

if and only if $d \leq r - l$. Hence we have to ensure that the width of each interval is at least d before narrowing. But interval expansion guarantees that the width

models	A	B	C	adjustments	A	B	C
initial model:	0.0	0.3	0.6	*adjust* $(0, 64)$:	0.0	0.297	0.594
after A:	0.0	0.4	0.7	*adjust* $(0, 38)$:	0.0	0.395	0.684
after B:	0.0	0.4	0.8	*adjust* $(30, 52)$:	0.0	0.364	0.773
after A:	0.0	0.4	0.8	*adjust* $(24, 56)$:	0.0	0.375	0.781
after C:	0.0	0.5	0.7	*adjust* $(8, 64)$:	0.0	0.500	0.696

Fig. 1. Model adjustment

of each (expanded) interval is greater than $w/4$ before narrowing, so interval collapse is avoided if $w/4 \geq d$. That was the whole point of making use of interval expansion.

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

Exercise 46. Prove that

$$(\forall p, q : 0 \leq p < q \leq d : \lfloor (r-l) \times p/d \rfloor < \lfloor (r-l) \times q/d \rfloor)$$

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

Exercise 47. According to the Haskell Report [1], the finite-precision type *Int* covers at least the range $[-2^{29}, 2^{29} - 1]$. What are suitable choices for w and d?

8.4 Final version of encode

Gathering together the ingredients of this data refinement, we can now give the final version of encode:

$$encode\ m\ ei = concat \cdot stream\ nextBits\ enarrow\ ei \cdot encodeSyms\ m$$

where

$$\begin{aligned}
&enarrow\ ei\ int2 && = (n, int1 \blacktriangleright int2)\\
&&&\quad \textbf{where}\ \ (n, int1) = expand\ ei\\
&expand\ (n, (l, r))\\
&\quad \mid w/4 \leq l \wedge r \leq 3 \times w/4 && = expand\ (n{+}1, (2 \times l - w/2, 2 \times r - w/2))\\
&\quad \mid \textbf{otherwise} && = (n, (l, r))\\
&nextBits\ (n, (l, r))\\
&\quad \mid r \leq w/2 && = Just\ (bits\ n\ 0, (0, (2 \times l, 2 \times r)))\\
&\quad \mid w/2 \leq l && = Just\ (bits\ n\ 1, (0, (2 \times l - w, 2 \times r - w)))\\
&\quad \mid \textbf{otherwise} && = Nothing
\end{aligned}$$

Arithmetic coding is now implemented by $encode\ m\ (0, (0, w))$.

Exercise 48. Instead of using semi-open intervals $[l..r)$ we could use a closed interval $[l..r - 1]$. What modifications are required to the definitions of *encode* and *decode*, and why should such a representation have an advantage over the semi-open one?

Exercise 49. Notwithstanding everything that has gone before, encoding is *not* guaranteed to work with any form of limited-precision arithmetic! Why not?

Exercise 50. Imagine a static model of three equiprobable symbols A, B and C, so that B is assigned the range $[{}^1/_3..{}^2/_3)$. Suppose a message of a billion B's is to be encoded. What is the output? How big does n get in the definition of *expand*? What does this relationship reveal about the answer to the previous exercise?

8.5 Decoding in the integer version

Decoding with limited-precision arithmetic is again implemented by appeal to stream inversion, just as in the previous version. Let us start by showing how to compute the symbol s from $bs = encode\ m\ ei\ (s : ss)$ under the assumption that $nextBits\ ei = Nothing$, so that ei straddles ${}^1/_2$ and $expand\ ei$ delivers an integer that will not collapse to the empty interval on narrowing. Setting $wfromBits = (w\times) \cdot fromBits$, we know that $x = wfromBits\ bs$ is a fraction in the interval $[0..w)$ satisfying

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

How can we compute s given x, m, and ei? We need to be able to do this in order to define the helper function *nextSym* for *unstream*.

To determine s, we make use of the following property of floors: for all integers n and fractions x, we have $n \le \lfloor x \rfloor \equiv n \le x$. Ignorance of this simple rule has marred practically every published paper on arithmetic coding that we have read.

We now reason:

$$\begin{array}{ll}
& x\ \underline{within}\ (contract\ (enarrow\ ei\ (encodeSym\ m\ s))) \\
\equiv & \quad \{\text{setting } (n,(l,r)) = expand\ ei\ \} \\
& x\ \underline{within}\ (contract\ (n,(l,r) \blacktriangleright encodeSym\ m\ s)) \\
\equiv & \quad \{\text{setting } y = scale\ (n,x)\ \} \\
& y\ \underline{within}\ ((l,r) \blacktriangleright encodeSym\ m\ s) \\
\equiv & \quad \{\text{setting } (p,q,d) = encodeSym\ m\ s\ \} \\
& l + \lfloor (r-l) \times {}^p/_d \rfloor \le y < l + \lfloor (r-l) \times {}^q/_d \rfloor \\
\equiv & \quad \{\text{arithmetic}\} \\
& \lfloor (r-l) \times {}^p/_d \rfloor \le y - l < \lfloor (r-l) \times {}^q/_d \rfloor \\
\equiv & \quad \{\text{rule of floors, setting } k = \lfloor y \rfloor\} \\
& \lfloor (r-l) \times {}^p/_d \rfloor \le k - l < \lfloor (r-l) \times {}^q/_d \rfloor \\
\equiv & \quad \{\text{arithmetic}\} \\
& \lfloor (r-l) \times {}^p/_d \rfloor < k - l + 1 \le \lfloor (r-l) \times {}^q/_d \rfloor \\
\equiv & \quad \{\text{rule of floors}\} \\
& (r-l) \times {}^p/_d < k - l + 1 \le (r-l) \times {}^q/_d \\
\equiv & \quad \{\text{arithmetic}\} \\
& p \le ((k-l+1) \times d - 1)/(r-l) < q \\
\equiv & \quad \{\text{rule of floors}\} \\
& p \le \lfloor ((k-l+1) \times d - 1)/(r-l) \rfloor < q
\end{array}$$

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

$$\begin{array}{l} nextSym\ (m, ei)\ x = decodeSym\ m\ t \\ \quad \textbf{where}\ t \quad\quad = ((k - l + 1) \times denom\ m - 1)\ \underline{div}\ (r - l) \\ \qquad\quad k \quad\quad = \lfloor scale\ (n, x) \rfloor \\ \qquad\quad (n, (l, r)) = expand\ ei \end{array}$$

Armed with this result, we can now tackle the task of inverting *encode*. First, as before, we rewrite *encode* in the form

$$encode\ m\ ei = concat \cdot stream\ nextBitsM\ step\ (m, ei)$$

where $step\ (m, ei)\ s = (newModel\ m\ s, enarrow\ ei\ (encodeSym\ m\ s))$ and *nextBitsM* carries the model as an extra argument:

$$\begin{array}{l} nextBitsM\ (m, (n, (l, r))) \\ \quad \mid r \leq {}^{w}\!/_{2} \quad\quad\quad\quad = Just\ (bits\ n\ 0, (m, (0, (2 \times l, 2 \times r)))) \\ \quad \mid {}^{w}\!/_{2} \leq l \quad\quad\quad\quad = Just\ (bits\ n\ 1, (m, (0, (2 \times l - w, 2 \times r - w)))) \\ \quad \mid \textbf{otherwise} \quad\quad\quad\quad = Nothing \end{array}$$

Now set $x = wfromBits\ (concat\ (stream\ nextBitM\ step\ (m, ei)\ (s : ss)))$. An appeal to fold-fusion gives

$$\begin{array}{l} wfromBits = foldr\ pack\ (w/2) \\ \qquad\qquad \textbf{where}\ pack\ b\ x = (w \times b + x)/2 \end{array}$$

A second appeal to fold-fusion gives

$$wfromBits \cdot concat = foldr\ (\oplus)\ (w/2)$$

where $bs \oplus x = foldr\ pack\ x\ bs$. Moreover, defining

$$x \ominus bs = foldl\ unpack\ x\ bs$$

where $unpack\ x\ b = 2 \times x - w \times b$, we have $(bs \oplus x) \ominus bs = x$ by Exercise 14.

All the ingredients for destreaming are now in place, and we can define

$$\begin{array}{l} decode\ m\ ei\ bs = \\ \quad unstream\ nextBitsM\ step\ nextSym\ (\ominus)\ (m, ei)\ (wfromBits\ bs) \end{array}$$

where

$$\begin{array}{l} nextSym\ (m, ei)\ x = decodeSym\ m\ t \\ \quad \textbf{where}\ t \quad\quad = ((k - l + 1) \times denom\ m - 1)\ \underline{div}\ (r - l) \\ \qquad\quad k \quad\quad = \lfloor scale\ (n, x) \rfloor \\ \qquad\quad (n, (l, r)) = expand\ ei \end{array}$$

and

$$\begin{array}{l} x \ominus bs = foldl\ unpack\ x\ bs \\ \qquad\quad \textbf{where}\ unpack\ x\ b = 2 \times x - w \times b \end{array}$$

The one remaining fly in the ointment is that *decode* is not incremental, as all elements of *bs* are inspected in order to compute *wfromBits bs*.

8.6 A final data refinement

Consider the first invocation of *nextSym* in the computation *decode* m ei bs. We have to compute

$$k = \lfloor 2^n \times (wfromBits\ bs - {}^{w}/_{2}) + {}^{w}/_{2} \rfloor$$

This can be done without inspecting all of bs. We only need to compute the first $e + n$ bits, where $w = 2^e$. This is the clue to making *decode* incremental.

Suppose we represent bs not by $x = wfromBits\ bs$ but by a pair (z, rs) where z is the *binary integer* formed from $take\ e\ bs$ (assuming bs contains at least e bits) and $rs = drop\ e\ bs$. Then $z = \lfloor wfromBits\ bs \rfloor$. If bs contains fewer than e bits, then we can always append a 1 to bs followed by a sufficent number of 0s. To justify this, recall Exercise 20. Let us call this computation *buffering* and write $(z, rs) = buffer\ bs$.

Given $(z, rs) = buffer\ bs$ we can now compute $k = fscale\ (n, (z, rs))$, where

$$fscale\ (n, (z, rs)) = foldl\ (\lambda\ x\ b \to 2 \times x + b - {}^{w}/_{2})\ z\ (take\ n\ rs)$$

The proof is left as an exercise. Hence k can be computed by inspecting only the first $e + n$ bits of bs.

To install this final refinement we need to show how to compute *buffer*. There are two ways to do it and we will need both. The first is to define

$$\begin{aligned} buffer\ bs &= (foldl\ (\lambda\ x\ b \to 2 \times x + b)\ 0\ cs, rs) \\ &\quad \textbf{where}\ (cs, rs) = splitAt\ e\ (bs +\!\!+ 1 : replicate\ (e - 1)\ 0) \end{aligned}$$

The definition of z uses the standard method for converting a bit string into a binary integer. This method is used in the final version of *decode*.

But we also have to show how to maintain the representation (z, rs) during the destreaming process. We leave it as an exercise to show that *buffer* can also be computed by

$$\begin{aligned} buffer &= foldr\ op\ ({}^{w}/_{2}, [\,])\ bs \\ op\ b\ (z, rs) &= (y, r : rs) \\ &\quad \textbf{where}\ (y, r) = (w \times b + z)\ \underline{divMod}\ 2 \end{aligned}$$

The point of this alternative is that we have

$$foldr\ op\ ({}^{w}/_{2}, [\,]) \cdot concat = foldr\ (\oplus)\ ({}^{w}/_{2}, [\,])$$

where $bs \oplus (x, ds) = foldr\ op\ (x, ds)\ bs$. Moreover, we can invert $\oplus$ by defining $\ominus$ to be

$$\begin{aligned} (z, rs) \ominus bs &= foldl\ unop\ (z, rs)\ bs \\ unop\ (z, rs)\ b &= (2 \times z - w \times b + head\ rs, tail\ rs) \end{aligned}$$

Now all the ingredients for destreaming are once again in place.

Exercise 51. Show that $\lfloor scale\ (n, wfromBits\ bs) \rfloor = fscale\ (n, buffer\ bs)$, where

$$fscale\ (n, (z, rs)) = foldl\ (\lambda\ x\ b \to 2 \times x + b - {}^{w}/_{2})\ z\ (take\ n\ rs)$$

Exercise 52. Show that

$$\begin{aligned} &\mathit{buffer} &&= \mathit{foldr}\ \mathit{op}\ ({}^{w}\!/_{2}, [\,])\ \mathit{bs} \\ &\mathit{op}\ b\ (z, rs) &&= (y, r : rs) \\ &\qquad\qquad &&\textbf{where}\ (y, r) = (w \times b + z)\ \underline{\mathit{divMod}}\ 2 \end{aligned}$$

8.7 Final version of decode

Here is the final version of *decode*:

$$\begin{aligned} &\mathit{decode}\ m\ ei\ \mathit{bs} = \\ &\quad \mathit{unstream}\ \mathit{nextBitsM}\ \mathit{step}\ \mathit{nextSym}\ (\ominus)\ (m, ei)\ (\mathit{buffer}\ \mathit{bs}) \\ &\mathit{buffer}\ \mathit{bs} = (z, rs) \\ &\quad \textbf{where}\ z &&= \mathit{foldl}\ (\lambda\ x\ b \to 2 \times x + b)\ 0\ cs \\ &\qquad\quad (cs, rs) &&= \mathit{splitAt}\ e\ (\mathit{bs} +\!\!+ 1 : \mathit{replicate}\ (e - 1)\ 0) \\ &\mathit{nextSym}\ (m, ei)\ (z, rs) = \mathit{decodeSym}\ m\ t \\ &\quad \textbf{where}\ t &&= ((k - l + 1) \times \mathit{denom}\ m - 1)\ \underline{\mathit{div}}\ (r - l) \\ &\qquad\quad k &&= \mathit{fscale}\ (n, (z, rs)) \\ &\qquad\quad (n, (l, r)) &&= \mathit{expand}\ ei \\ &(z, rs) \ominus \mathit{bs} = \mathit{foldl}\ \mathit{unop}\ (z, rs)\ \mathit{bs} \\ &\quad \textbf{where}\ \mathit{unop}\ (z, rs)\ b = (2 \times z - w \times b + \mathit{head}\ rs, \mathit{tail}\ rs) \\ &\mathit{fscale}\ (n, (z, rs)) = \mathit{foldl}\ (\lambda\ x\ b \to 2 \times x + b - {}^{w}\!/_{2})\ z\ (\mathit{take}\ n\ rs) \end{aligned}$$

The remaining functions *nextBitsM*, *step*, and *expand* were defined previously.

9 Conclusions

The reader who has followed us up to now will appreciate that there is rather a lot of arithmetic in arithmetic coding, and that includes the arithmetic of folds and unfolds as well as numbers. As we said at the start, arithmetic coding is a simple idea but one that requires care to implement with limited-precision integer arithmetic. To the best of our knowledge, no previous description of arithmetic coding has ever tackled the formal basis for why the method works, let alone providing a formal development of the coding and decoding algorithms.

Perhaps not surprisingly we went through many iterations of the development, considering different ways of expressing the concepts of streaming and stream inversion. The final constructions given above differ markedly from the versions given in the Summer School in August, 2002. None of these iterations would have been possible without the availability of a functional perspective, whose smooth proof theory enabled us to formulate theorems, prove them, and perhaps discard them, quite quickly. Whether or not the reader has followed all the details, we hope we have demonstrated that functional programming and equational reasoning are essential tools of thought for expressing and proving properties of complicated algorithms, and that the ability to define structured recursion operators, such as *foldl*, *unfoldr*, *stream* and *destream*, is critical for formulating and understanding patterns of computation.

References

1. *Haskell 98: A Non-Strict, Purely Functional Language.* Available online at www.haskell.org/onlinereport.
2. *Standard Libraries for Haskell 98.* Available online at www.haskell.org/onlinelibrary.
3. R. S. Bird. *Introduction to Functional Programming using Haskell.* International Series in Computer Science. Prentice Hall, 1998.
4. R. S. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.
5. R. M. Fano. *Transmission of Information.* MIT Press, Cambridge MA, and Wiley, NY, 1961.
6. J. Gibbons. Origami programming. In *The Fun of Programming*, J. Gibbons and O. de Moor, eds, Palgrave, 2003.
7. J. Gibbons. The Third Homomorphism Theorem. *J. Functional Prog.*, Vol 6, No 4, 657–665, 1996.
8. P. G. Howard and J. S. Vitter. Arithmetic coding for data compression. *Proc. IEEE*, Vol 82, No 6, 857–865, 1994.
9. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.* Vol 40, No 9, 1098–1101, Sept. 1952.
10. J. Jiang. Novel design of arithmetic coding for data compression. *IEE Proc. Comput. Dig. Tech.*, Vol 142, 6 (Nov) 419–424, 1995.
11. E. Meijer, M. Fokkinga and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 523, 124–144, 1991.
12. 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.
13. C. E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.* Vol 27, 79–423, 1948.
14. 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.