Origami Programming for Fun and Profit

Jeremy Gibbons

ECOOP Summer School, July 2019

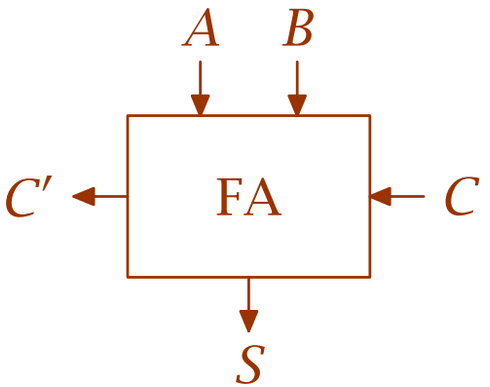
1. Pop quiz

What do the following have in common?

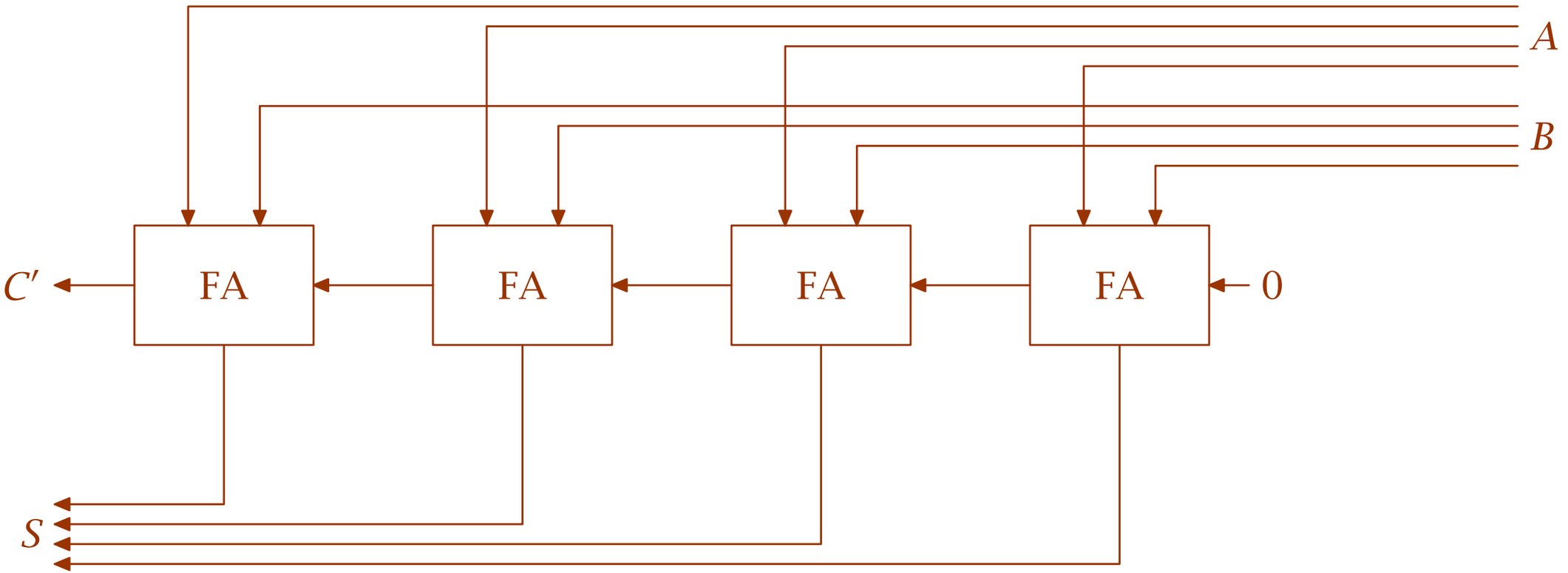
- *carry-lookahead adders*
- *fast language recognizers*
- *streaming number conversions*

Full 1-bit adder

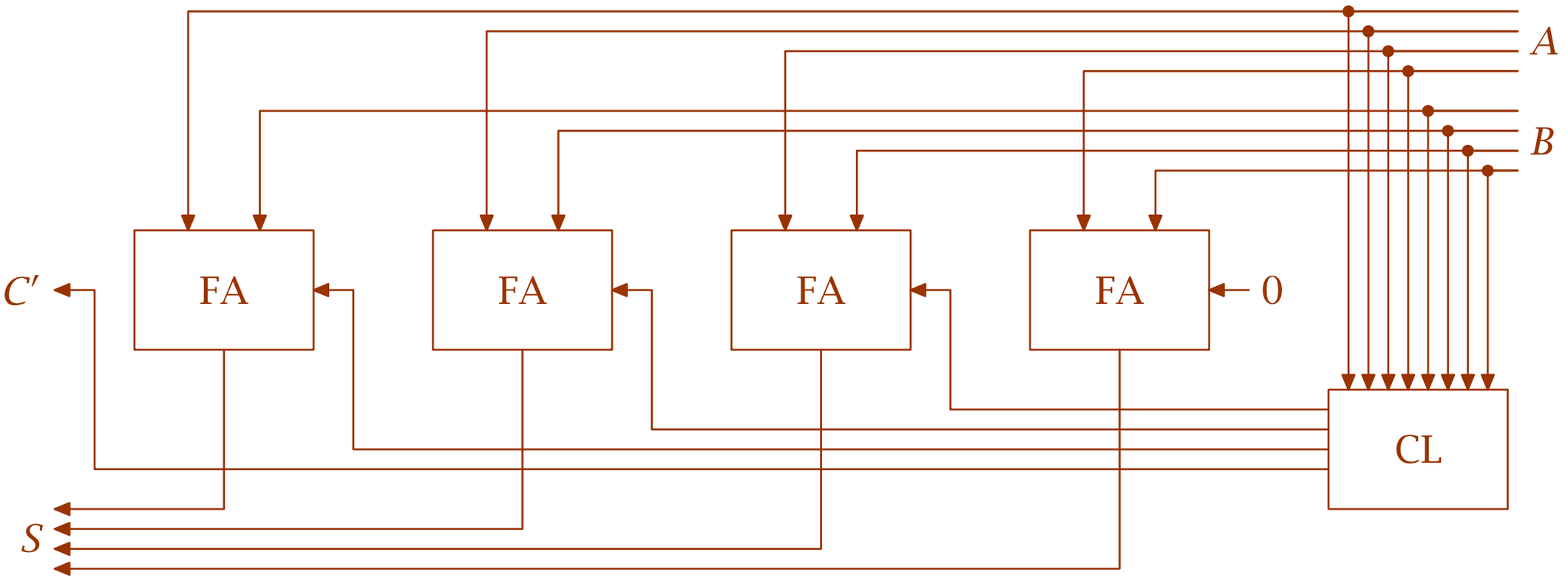
C	A	B	C'	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Ripple-carry adder



Carry-lookahead



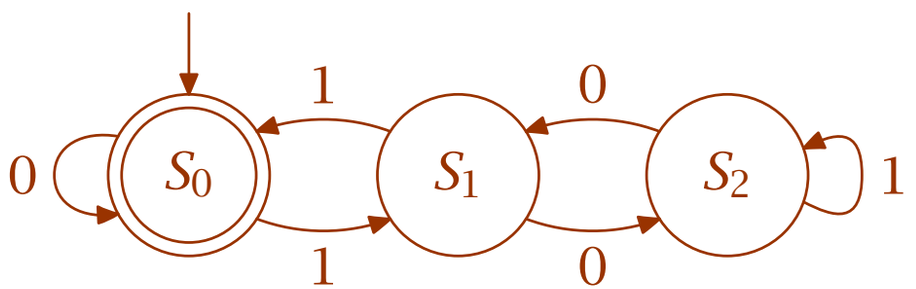
Each pair of bits will *generate* $(1,1)$, *kill* $(0,0)$, or *propagate* $(0,1$ or $1,0)$ a carry.
G, K, P closed under composition: $G \oplus f = G$, $K \oplus f = K$, $P \oplus f = f$.
Each carry-in determined by composition of preceding carry-transformers.
Compute all such compositions in one go as a *scan*, apply to initial carry-in 0.

Fast language recognizer

Regular language of multiples of 3 in binary (0, 11, 110, 0110, 1001...)

$$(0 \mid (1(01^*(00)^*0)^*1)^*)^*$$

recognized by nondeterministic finite automaton:



	0	1
S_0	S_0	S_1
S_1	S_2	S_0
S_2	S_1	S_2

State transition function $Q \times \Sigma \to Q$ is equivalently $\Sigma \to (Q \to Q)$.

So translate each input symbol to a $Q \to Q$ function (ie a matrix).
Aggregate in parallel, apply to initial state S_0.

Streaming number conversion

Evaluate finite decimal representation from the right:

$$0.125_{10} = {}^{1}/_{10} \times (1 + {}^{1}/_{10} \times (2 + {}^{1}/_{10} \times (5 + 0))) = {}^{1}/_{8}$$

Can then continue processing, eg to write out in binary.

But what about *infinite* or *unbounded* decimals? eg $0.123\ldots_{10}$.
Should be able to *stream* conversion to binary.

Bound the possible result:

$$0.00011111101_2 < 0.123_{10} \leqslant 0.123\ldots_{10} < 0.124_{10} < 0.00011111111_2$$

So first 3 input digits determine first 8 output bits.

Each digit yields a function $\lambda x \to {}^{1}/_{10} \times (d + x)$.
Closed under composition as $\lambda x \to v \times (u + x)$. Identity is $\lambda x \to 1 \times (0 + x)$.
Finite representation as a pair (v, u) of rationals.

2. Folds on lists

Many functions look very different at first glance,
but have profound similarities on closer inspection.

It is worth identifying and studying those common features:

- understanding
- investment
- reuse
- generalization

(The same motivations as for *design patterns*.)

Fold-right

Aggregate a list of elements:

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

so

$$
\mathit{foldr}\ f \quad e\ [a, b, c] = f\ a\ (f\ b\ (f\ c\ e))
$$

Fold-right

Aggregate a list of elements, written with infix operator:

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

so

$$\mathit{foldr}\ (\oplus)\ e\ [\,a, b, c\,] = a \oplus (b \oplus (c \oplus e))$$

For example, $\mathit{decimal}\ [\,1, 2, 3\,] = {}^{123}/_{1000}$ where

$$
\begin{aligned}
&\mathit{decimal} :: [\,\mathit{Int}\,] \to \mathit{Rational} \\
&\mathit{decimal} = \mathit{foldr}\ \mathit{digit}\ 0\ \textbf{where}\ \mathit{digit}\ d\ x = (\mathit{fromIntegral}\ d + x)\ /\ 10
\end{aligned}
$$

Many other examples too: *sum*, *maximum*, *and*, *all*, *filter*, *concat*...

Fold-right

Aggregate a list of elements, written with infix operator:

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

so

$$\mathit{foldr}\ (\oplus)\ e\ [a, b, c] = a \oplus (b \oplus (c \oplus e))$$

For example, $\mathit{decimal}\ [1, 2, 3] = {}^{123}/_{1000}$ where

$$
\begin{aligned}
&\mathit{decimal} :: [\mathit{Int}] \to \mathit{Rational} \\
&\mathit{decimal} = \mathit{foldr}\ \mathit{digit}\ 0\ \textbf{where}\ \mathit{digit}\ d\ x = (\mathit{fromIntegral}\ d + x)\ /\ 10
\end{aligned}
$$

Many other examples too: *sum*, *maximum*, *and*, *all*, *filter*, *concat*...

Fold-left

In a *foldr*, the parentheses collect at the right.
Dually, in a *foldl* they collect at the left:

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

with an *accumulating parameter*, so

$$\mathit{foldl}\ (\otimes)\ e\ [\,a, b, c\,] = ((e \otimes a) \otimes b) \otimes c$$

For example, $\mathit{natural}\ [\,1, 2, 3\,] = 123$ where

$$
\begin{aligned}
&\mathit{natural} :: [\,\mathit{Int}\,] \to \mathit{Integer} \\
&\mathit{natural} = \mathit{foldl}\ \mathit{digit}\ 0\ \textbf{where}\ \mathit{digit}\ n\ d = 10 \times n + d
\end{aligned}
$$

Not quite so many examples as *foldr*, but still useful.

1. Duality theorems

First Duality Theorem: If $\oplus$ is associative with neutral element e, then

$$foldr\ (\oplus)\ e\ x = foldl\ (\oplus)\ e\ x$$

for finite as. Eg

$$foldr\ (+)\ 0\ x = foldl\ (+)\ 0\ x = sum\ x$$

Third Duality Theorem: For any $\oplus$ and e,

$$foldr\ (\oplus)\ e\ (reverse\ x) = foldl\ (flip\ (\oplus))\ e\ x$$

for all x, where $flip\ f\ a\ b = f\ b\ a$. Eg

$$natural = foldr\ (flip\ digit)\ 0 \circ reverse$$

2. Associative fold

A *list homomorphism* is a fold after a map in a monoid:

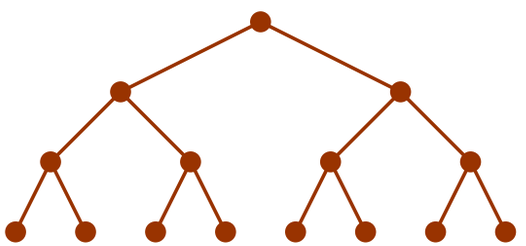
$$
\begin{aligned}
&\mathit{foldMap} :: (b \to b \to b) \to b \to (a \to b) \to [a] \to b \\
&\mathit{foldMap}\ (\odot)\ e\ f\ [\,] &&= e \\
&\mathit{foldMap}\ (\odot)\ e\ f\ [a] &&= f\ a \\
&\mathit{foldMap}\ (\odot)\ e\ f\ (x +\!\!+ y) &&= \mathit{foldMap}\ (\odot)\ e\ f\ x\ \odot\ \mathit{foldMap}\ (\odot)\ e\ f\ y
\end{aligned}
$$

provided that $\odot$ is associative with neutral element e.

For example,

$$\mathit{all}\ p = \mathit{foldMap}\ (\wedge)\ \mathit{True}\ p$$

can be executed in parallel in log time
(on a suitable list representation).



Sequentialization and parallelization

First Homomorphism Theorem:

$$foldMap\ (\odot)\ e\ f = foldMap\ (\odot)\ e\ id \circ map\ f$$

Second Homomorphism Theorem: for associative $\odot$ with neutral element e,

$$foldMap\ (\odot)\ e\ f = foldr\ (\lambda a\ b \to f\ a \odot b)\ e = foldl\ (\lambda a\ b \to a \odot f\ b)\ e$$

Third Homomorphism Theorem—conversely: if

$$h = foldr\ (\oplus)\ e = foldl\ (\otimes)\ e$$

then there exists f and associative $\odot$ with neutral element e such that

$$h = foldMap\ (\odot)\ e\ f$$

Eg $sort = foldr\ ins\ [\,] = foldl\ (flip\ ins)\ [\,]$ (insertion sort),
hence $sort\ (x +\!\!+ y) = sort\ x \odot sort\ y$ for some $\odot$...

3. The Trick

Any fold-right or fold-left can be implemented as
an associative fold (followed by an application)!

$$\mathit{foldr}\ (\oplus)\ e\ x = \mathit{foldMap}\ (\circ)\ \mathit{id}\ (\oplus)\ x\ e$$
$$\mathit{foldl}\ (\otimes)\ e\ x\ = \mathit{foldMap}\ (\mathit{flip}\ (\circ))\ \mathit{id}\ (\mathit{flip}\ (\otimes))\ x\ e$$

because function application and composition are related:

$$a \oplus (b \oplus (c \oplus e))\ =\ (a\oplus)\ \$\ (b\oplus)\ \$\ (c\oplus)\ \$\ e\ =\ ((a\oplus) \circ (b\oplus) \circ (c\oplus))\ e$$

And of course, $(\circ)$ is associative with neutral element id.

- *carry-lookahead*: carry-transformers as functions G, K, P
- *language recognition*: state transitions as functions $Q \to Q$
- *streaming conversion*: digits as functions $\lambda x \to v \times (u + x)$

All have *finite representations*. First two compute fold or scan in *log time in parallel*.

4. Unfolds

Folds *consume* data structures; unfolds *produce* them.

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

For example, $\mathit{digits}\ (^1/_8) = [\,1, 2, 5\,]$, where

$$
\begin{aligned}
&\mathit{digits} :: \mathit{Rational} \to [\,\mathit{Int}\,] \quad \text{-- input assumed in } [0, 1)\\
&\mathit{digits} = \mathit{unfoldr}\ \mathit{next}\ \textbf{where}\\
&\quad \mathit{next}\ x = \textbf{if}\ x == 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\\
&\qquad\qquad \textbf{let}\ d = \lfloor 10 \times x \rfloor\ \textbf{in}\ \mathit{Just}\ (d, 10 \times x - d)
\end{aligned}
$$

Base conversion

Generalize to arbitrary radix r:

$$
\begin{aligned}
&\mathit{rational} :: \mathit{Int} \to [\,\mathit{Int}\,] \to \mathit{Rational} \\
&\mathit{rational}\ r = \mathit{foldr}\ \mathit{inject}_r\ 0\ \textbf{where} \\
&\quad \mathit{inject}_r\ d\ x = (d + x)\ /\ r \\
&\mathit{digits} :: \mathit{Int} \to \mathit{Rational} \to [\,\mathit{Int}\,] \\
&\mathit{digits}\ r = \mathit{unfoldr}\ (\mathit{next}\ r)\ \textbf{where} \\
&\quad \mathit{next}\ r\ x = \textbf{if}\ x == 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else} \\
&\qquad\qquad \textbf{let}\ d = \lfloor r \times x \rfloor\ \textbf{in}\ \mathit{Just}\ (d, \mathit{extract}_r\ d\ x) \\
&\quad \mathit{extract}_r\ d\ y = r \times y - d
\end{aligned}
$$

Then $0.125_{10} = {}^{1}/_{8} = 0.010101\ldots_3$:

$$\mathit{digits}\ 3\ (\mathit{rational}\ 10\ [1,2,5]) = [0,1,0,1,0,1\ldots]$$

But *rational* starts at right, so only works for finite inputs. What about infinite?

Use the Trick!

$$
\begin{aligned}
&\mathit{rational} :: \mathit{Int} \to [\,\mathit{Int}\,] \to \mathit{Rational} \\
&\mathit{rational}\ r\ ds = \mathit{foldl}\ \mathit{step}'_r\ \mathit{id}\ ds\ 0\ \textbf{where} \\
&\quad \mathit{step}'_r\ f\ d = f \circ \mathit{inject}_r\ d
\end{aligned}
$$

(ie $\mathit{step}'_r\ f\ d\ x = f\ ((d + x)\ /\ r)$).

Now *rational* starts from the left; uses functions of the form $\lambda x \to v \times (u + x)$, which have finite representations (v, u):

$$
\begin{aligned}
&\mathit{rational} :: \mathit{Int} \to [\,\mathit{Int}\,] \to \mathit{Rational} \\
&\mathit{rational}\ r\ ds = \mathit{reify}\ (\mathit{foldl}\ \mathit{step}''_r\ (1, 0)\ ds)\ \textbf{where} \\
&\quad \mathit{step}''_r\ (v, u)\ d = (v\ /\ r, u \times r + d) \\
&\quad \mathit{reify}\ (v, u) = v \times u
\end{aligned}
$$

But still only works for finite inputs (because of *foldl*).

5. Streaming

```
stream :: (b → Maybe (c, b)) → (b → a → b) → b → [a] → [c]
stream g f b x =
  case g b of Just (c, b') → c : stream g f b' x
              Nothing      → case x of a : x' → stream g f (f b a) x'
                                       []     → []
```

Alternates between *production* (when it can) and *consumption* (when it can't). Not limited to finite input.

For example, a streaming version of concatenation:

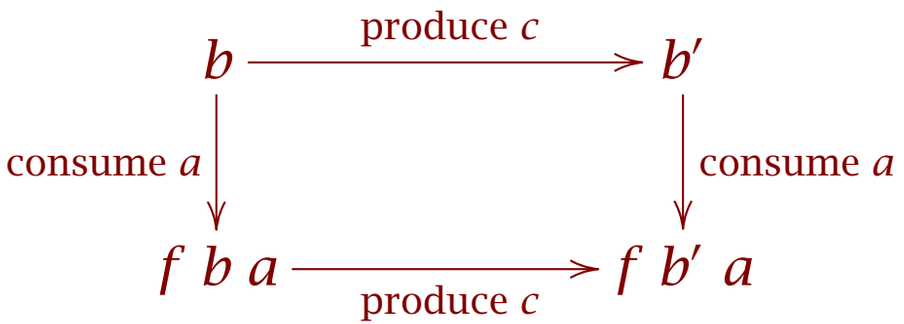
```
concat :: [[a]] → [a]
concat = stream uncons (++) []
  where uncons (a : x) = Just (a, x)
        uncons []      = Nothing
```

The streaming condition

The *streaming condition* for g and f is that

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

Informally,



From any productive state b, consuming a new input a preserves *productivity*, *output*, and *future behaviour*.

If streaming condition holds for g and f, then

$$\mathit{stream}\ g\ f\ b\ x = \mathit{unfoldr}\ g\ (\mathit{foldl}\ f\ b\ x)$$

on finite lists x. Eg $\mathit{stream}\ \mathit{uncons}\ (+\!\!+)\ [\,] = \mathit{unfoldr}\ \mathit{uncons} \circ \mathit{foldl}\ (+\!\!+)\ [\,]$.

Streaming base conversion?

Sadly, streaming condition does not hold. Eg decimal to ternary:

$$0.32\ldots_{10} = 0.0\ldots_3 \quad \text{-- less than } {}^1/_3$$
$$0.34\ldots_{10} = 0.1\ldots_3 \quad \text{-- greater than } {}^1/_3$$

Not safe to commit to output $0.1\ldots_3$ after seeing only input $0.3\ldots_{10}$.

Only safe to commit when *stable under all future inputs*:

$$\begin{aligned} next'_r\ f = \textbf{if } d_0 == d_1 \textbf{ then } &\mathit{Just}\ (d_0, \mathit{extract}_r\ d_0 \circ f) \\ \textbf{else } &\mathit{Nothing} \\ \textbf{where } d_0 = \lfloor r \times f\ 0 \rfloor &\quad \text{-- lower bound on future inputs} \\ d_1 = \lfloor r \times f\ 1 \rfloor &\quad \text{-- upper bound on future inputs} \end{aligned}$$

May then need to *flush buffer* once input is exhausted.

Flushing streams

```
fstream :: (b → Maybe (c, b)) → (b → [c]) → (b → a → b) → b → [a] → [c]
fstream g h f b x =
  case g b of Just (c, b') → c : fstream g h f b' x
              Nothing      → case x of a : x' → fstream g h f (f b a) x'
                                       []    → h b   -- flush buffer
```

eg

```
group :: Int → [[a]] → [[a]]
group n = fstream (split n) wrap (++) [] where
  split n x = if length x ⩾ n then Just (splitAt n x) else Nothing
  wrap x = if null x then [] else [x]
```

If streaming condition holds for g and f, then

```
fstream g h f b x = apo g h (foldl f b x)
```

for finite lists x, where *apo* is *primitive corecursion*...

Apomorphism (primitive corecursion)

Recall unfold:

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

Apomorphism (primitive corecursion)

Apomorphism has ‘finalization’ operator h too:

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

eg insertion into sorted list:

$$
\begin{aligned}
&\mathit{insert}\ b\ \mathit{as} = \mathit{apo}\ \mathit{next}\ \mathit{snd}\ (\mathit{Just}\ b, \mathit{as})\ \mathbf{where} \\
&\quad \mathit{next}\ (\mathit{Just}\ b, a : x) \mid b > a = \mathit{Just}\ (a, (\mathit{Just}\ b, x)) \\
&\qquad\qquad\qquad\qquad\quad \mid b \leqslant a = \mathit{Just}\ (b, (\mathit{Nothing}, a : x)) \\
&\quad \mathit{next}\ (\mathit{Just}\ b, [\,]) \qquad\qquad\quad = \mathit{Just}\ (b, (\mathit{Nothing}, [\,])) \\
&\quad \mathit{next}\ (\mathit{Nothing}, x) \qquad\qquad\quad = \mathit{Nothing}
\end{aligned}
$$

Back to streaming base conversion

Streaming condition *does* hold for $next'_s$ and $step'_r$:

$$
\begin{aligned}
&convert :: Int \to Int \to [Int] \to [Int] \\
&convert\ r\ s = fstream\ next'_s\ flush'_s\ step'_r\ id\ \textbf{where} \\
&\quad next'_r\ f \qquad = \textbf{if}\ d_0 == d_1\ \textbf{then}\ Just\ (d_0, extract_r\ d_0 \circ f) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ Nothing \\
&\qquad \textbf{where}\ (d_0, d_1) = (\lfloor r \times f\ 0 \rfloor, \lfloor r \times f\ 1 \rfloor) \\
&\quad flush'_r\ f \qquad = unfoldr\ (next\ r)\ (f\ 0) \\
&\qquad \textbf{where}\ next\ r\ x = \textbf{if}\ x == 0\ \textbf{then}\ Nothing\ \textbf{else} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ d = \lfloor r \times x \rfloor\ \textbf{in}\ Just\ (d, extract_r\ d\ x) \\
&\quad step'_r\ f\ d \qquad = f \circ inject_r\ d
\end{aligned}
$$

works (almost*) to convert numeral from one radix to another:

$$convert\ 10\ 3\ [1, 2, 5] = [0, 1, 0, 1, 0, 1 \ldots]$$

Back to streaming base conversion

Streaming condition also holds for $next''_s$ and $step''_r$ (finite representation):

$$
\begin{aligned}
&convert :: Int \to Int \to [Int] \to [Int] \\
&convert\ r\ s = fstream\ next''_s\ flush''_s\ step''_r\ (1,0)\ \textbf{where} \\
&\quad next''_r\ (v,u) \quad = \textbf{if}\ d_0 == d_1\ \textbf{then}\ Just\ (d_0, (v \times r, u - d_0\ /\ (v \times r))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ Nothing \\
&\qquad \textbf{where}\ (d_0, d_1) = (\lfloor r \times v \times u \rfloor, \lfloor r \times v \times (u+1) \rfloor) \\
&\quad flush''_r\ (v,u) \quad = unfoldr\ (next\ r)\ (v \times u) \\
&\qquad \textbf{where}\ next\ r\ x = \textbf{if}\ x == 0\ \textbf{then}\ Nothing\ \textbf{else} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ d = \lfloor r \times x \rfloor\ \textbf{in}\ Just\ (d, extract_r\ d\ x) \\
&\quad step''_r\ (v,u)\ d = (v\ /\ r, u \times r + d)
\end{aligned}
$$

works (almost*) to convert numeral from one radix to another:

$$convert\ 10\ 3\ [1,2,5] = [0,1,0,1,0,1 \ldots]$$

4. Digits of pi

Expansion

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2 2^{i+1}}{(2i+1)!} = 2 + {}^{1}/_{3} \times (2 + {}^{2}/_{5} \times (2 + {}^{3}/_{7} \times (\cdots (2 + {}^{i}/_{2i+1} \times \cdots))))$$

$$= 3 + {}^{1}/_{10} \times (1 + {}^{1}/_{10} \times (4 + {}^{1}/_{10} \times \cdots))$$

Computing the digits of π is conversion from *mixed radix* base to *fixed radix*:

```
pi = stream next comp (1,0,1) lfts where
  lfts                  = [(i, 4*i+2, 2*i+1) | i <- [1..]]
  next z                = if m == n then Just (n, comp (10, -10*n, 1) z)
                                    else Nothing
    where (m, n) = (⌊extr z 3⌋, ⌊extr z 4⌋)
  comp (q,r,t) (q',r',t') = (q*q', q*r' + r*t', t*t')
  extr (q,r,t) x        = (q*x + r) / t
```

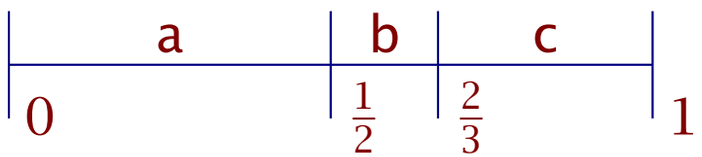
using upper-triangular matrices $\left(\begin{smallmatrix} q & r \\ 0 & t \end{smallmatrix}\right)$ of integers (no need to flush).

5. Arithmetic coding

Data compression, of a text to a bit sequence:

- *distribute alphabet* across unit interval
- *narrow* unit interval, character by character
- output *shortest binary fraction* in final interval

For example, with $\mathsf{a} \mapsto [0, \frac{1}{2})$, $\mathsf{b} \mapsto [\frac{1}{2}, \frac{2}{3})$, $\mathsf{c} \mapsto [\frac{2}{3}, 1)$



and text abacab:

$$[0,1) \xrightarrow{\mathsf{a}} [0,\tfrac{1}{2}) \xrightarrow{\mathsf{b}} [\tfrac{1}{4},\tfrac{1}{3}) \xrightarrow{\mathsf{a}} [\tfrac{1}{4},\tfrac{7}{24}) \xrightarrow{\mathsf{c}} [\tfrac{5}{18},\tfrac{7}{24}) \xrightarrow{\mathsf{a}} [\tfrac{5}{18},\tfrac{41}{144}) \xrightarrow{\mathsf{b}} [\tfrac{9}{32},\tfrac{61}{216})$$

Final interval $[\frac{9}{32}, \frac{61}{216})$ contains 0.01001 and no shorter binary fraction.

For efficiency, we wish to *stream* the process.

6. Further reading

- "Unbounded Spigot Algorithms for the Digits of π", Jeremy Gibbons, *American Mathematical Monthly*, **113**:318–328, 2006
- "Metamorphisms: Streaming Representation-Changers", Jeremy Gibbons, *Science of Computer Programming*, **65**:108–139, 2007
- "Arithmetic Coding with Folds and Unfolds", Richard Bird and Jeremy Gibbons, *Advanced Functional Programming*, LNCS **2638**:1–26, 2003

See also:


www.cs.ox.ac.uk/jeremy.gibbons


patternsinfp.wordpress.com


@jer_gib