Jeremy Gibbons
University of Oxford

Introduction to FP
Part 5: Laziness and Infinite Data Structures

5.1 Different evaluation orders

- recall different evaluation orders from before:

$$
\begin{array}{ll}
 & \mathit{square}\ (3+4) \\
\Rightarrow & \quad \{ \text{ defn of } + \} \\
 & \mathit{square}\ 7 \\
\Rightarrow & \quad \{ \text{ defn of } \mathit{square} \} \\
 & 7 \times 7 \\
\Rightarrow & \quad \{ \text{ defn of } \times \} \\
 & 49
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{square}\ (3+4) \\
\Rightarrow & \quad \{ \text{ defn of } \mathit{square} \} \\
 & (3+4) \times (3+4) \\
\Rightarrow & \quad \{ \text{ defn of } + \} \\
 & 7 \times (3+4) \\
\Rightarrow & \quad \{ \text{ defn of } + \} \\
 & 7 \times 7 \\
\Rightarrow & \quad \{ \text{ defn of } \times \} \\
 & 49
\end{array}
$$

- not two different answers
- but sometimes no answer at all!
- which order to choose?

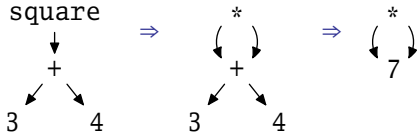
5.1 Applicative-order and normal-order evaluation

- applicative-order reduction of application $f\ e$:
  - first reduce e to normal form
  - then expand definition of f and continue reducing
- simple and obvious; easy to implement
- may not terminate when other evaluation orders would

- normal-order reduction of application $f\ e$:
  - expand definition of f, substituting e
  - reduce result of expansion
- avoids non-termination, if any evaluation order will
- may involve repeating work

5.1 A third way: lazy evaluation

- like normal-order evaluation, but instead of copying arguments we *share* them



- terms are directed graphs, not trees; *graph reduction*
- best of both worlds: evaluates argument only when needed, so terminating; but never evaluates argument more than once, so efficient

5.1 Lazy evaluation in Haskell

- pattern-matching forces reduction of arguments

  $head\ (enumFromTo\ 1\ 1000000) = ???$

- patterns matched top to bottom, left to right

  $False\ \&\&\ x = False$
  $True\ \&\&\ x = x$

- guards also trigger reduction (they desugar to **case** analysis and patterns)

  $f\ z \mid fst\ z > 0 \quad = fst\ z$
  $\quad\ \mid otherwise = snd\ z$

- local definitions not reduced until needed (**where**, **let** desugar to lambda)

  $g\ x = (x \neq 0\ \&\&\ y < 10)\ \textbf{where}\ y = 1\ /\ x$
  $\quad = (\lambda y \to x \neq 0\ \&\&\ y < 10)\ (1\ /\ x)$

5.2 Demand-driven programming

- lazy evaluation has useful implications for program design
- many computations can be thought of as *pipelines*
- with lazy evaluation, intermediate data structures need not exist all at once
- same effect requires major program surgery in most languages

> **Slogan:** lazy evaluation allows new and better means of modularizing programs

- nearest approximation in many languages is enumerators/iterators, *yield*...

5.2 A pipeline (approximately true)

$$
\begin{aligned}
& \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ [1..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ (1 : [2..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 0\ (1 : \mathit{map}\ \mathit{square}\ [2..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 1\ (\mathit{map}\ \mathit{square}\ [2..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 1\ (\mathit{map}\ \mathit{square}\ (2 : [3..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 1\ (4 : \mathit{map}\ \mathit{square}\ [3..100]) \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 5\ (\mathit{map}\ \mathit{square}\ [3..100]) \\
\Rightarrow\ & \ldots \\
\Rightarrow\ & \mathit{foldl}\ (+)\ 14\ (\mathit{map}\ \mathit{square}\ [4..100]) \\
\Rightarrow\ & \ldots \\
\Rightarrow\ & 338350
\end{aligned}
$$

5.2 Another pipeline

- insertion sort

$$
\begin{aligned}
&\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\
&\mathit{insertSort}\ [\,] \qquad = [\,] \\
&\mathit{insertSort}\ (x : xs) = \mathit{insert}\ x\ (\mathit{insertSort}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [a] \to [a] \\
&\mathit{insert}\ a\ [\,] \qquad = [a] \\
&\mathit{insert}\ a\ (b : xs) \\
&\quad \mid a \leqslant b \qquad = a : b : xs \\
&\quad \mid \mathit{otherwise} = b : \mathit{insert}\ a\ xs
\end{aligned}
$$

- minimum

$$
\begin{aligned}
&\mathit{minimum} :: (\mathit{Ord}\ a) \Rightarrow [a] \to a \\
&\mathit{minimum} = \mathit{head} \circ \mathit{insertSort}
\end{aligned}
$$

- complexity?

5.3 Infinite data structures

- demand-driven evaluation means that programs can manipulate *infinite* data structures
- whole structure is not evaluated at once (fortunately)
- because of laziness, finite result can be obtained from (finite prefix of) infinite data structure

5.3 Infinite lists

- $ones = 1 : ones$
- $[n..] = [n, n+1, n+2, \ldots$
- $[n, n+k..] = [n, n+k, n+2\times k, \ldots$
- $repeat\ n = n : repeat\ n$
- $iterate\ f\ x = x : iterate\ f\ (f\ x)$
- $fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$

fibs	0	1	1	2	3	5	8	...
tail fibs	1	1	2	3	5	8	13	...
zipWith (+)	1	2	3	5	8	13	21	...

5.3 No magic

- can apply functions to infinite data structures

  $$\mathit{filter}\ \mathit{even}\ [1..] = [2, 4, 6, 8, \ldots$$

- can even return finite results

  $$\mathit{takeWhile}\ (<10)\ [1..] = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

- note that these do not always behave like infinite sets in maths

  $$\mathit{filter}\ (<10)\ [1..] = [1, 2, 3, 4, 5, 6, 7, 8, 9$$

- to interrupt, ctrl-C

5.3 What does it mean?

- essential idea is that infinite data structure is *limit* of series of *approximations*
- eg infinite list

  $[1, 2, 3, 4, 5, \ldots$

  is limit of series of approximations

  $\bot$
  $1 : \bot$
  $1 : 2 : \bot$
  $1 : 2 : 3 : \bot$
  $\ldots$

- the same mathematics (limits, convergence) as in high school calculus

5.3 Primes

- recall bounded sequences of primes

$$primes\ m = [\, n \mid n \leftarrow [1\,..\,m], divisors\ n == [1, n]\,]$$
$$divisors\ n = [\, d \mid d \leftarrow [1\,..\,n], n\ `mod`\ d == 0\,]$$

- infinite sequence of primes

$$primes = [\, n \mid n \leftarrow [1\,..\,], divisors\ n == [1, n]\,]$$

- much more efficient version: (ersatz) *sieve of Eratosthenes*

$$primes = sieve\ [2\,..\,] \textbf{ where}$$
$$sieve\ (x : xs) = x : sieve\ [\, y \mid y \leftarrow xs, y\ `mod`\ x \neq 0\,]$$

(exercise: write *sieve* as an *unfold*)

- modularity: compatible with eg *take* and *takeWhile*

5.3 Pythagorean triples

- obvious definition

$$pyth = [\ (a,b,c) \mid a \leftarrow [1..], b \leftarrow [1..], c \leftarrow [1..], a \times a + b \times b == c \times c]$$

  doesn't work — why?

- instead

$$pyth = [\ (a,b,c) \mid c \leftarrow [1..], b \leftarrow [1..c-1], a \leftarrow [1..b-1], a \times a + b \times b == c \times c]$$

- then

$$pyth = [(3,4,5),(6,8,10),(5,12,13),\ldots$$

5.3 What's the point?

- better abstraction: some real-world entities are infinite
- better modularity: separation of concerns, reuse of components
- provides a model of interaction (but now superseded by monads)

- fun!

5.4 Unbounded spigot algorithm for the digits of π

Rabinowitz & Wagon's algorithm, obfuscated in C by Winter & Flammenkamp:

```
a[52514],b,c=52514,d,e,f=1e4,g,h;
main() {
  for(;b=c-=14;h=printf("%04d",e+d/f))
    for(e=d%=f;g=--b*2;d/=g)
      d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;
}
```

based on the expansion

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

A *spigot algorithm*: digits 'drip' out, one by one, with limited intermediate storage. (Here, four by four.)

5.4 Finite versus infinite sequences

R&W's algorithm is inherently *bounded*, committing initially to length:

> *"One cannot simply [keep going], because memory allocations must be made in advance."*

W&F's program operates on a *finite* array, generating just 15,000 digits.
In contrast, this program

$$
\begin{array}{l}
pi = g\ (1,0,1,1,3,3)\ \textbf{where} \\
\quad g\ (q,r,t,k,n,l) = \\
\quad\quad \textbf{if}\ 4\times q + r - t < n \times t \\
\quad\quad\quad \textbf{then}\ n : g\ (10\times q, 10\times (r - n\times t), t, k, \\
\quad\quad\quad\quad\quad\quad div\ (10\times (3\times q + r))\ t - 10\times n, l) \\
\quad\quad\quad \textbf{else}\quad\ g\ (q\times k, (2\times q + r)\times l, t\times l, k+1, \\
\quad\quad\quad\quad\quad\quad div\ (q\times (7\times k + 2) + r\times l)\ (t\times l), l+2)
\end{array}
$$

is based on *infinite* sequences, and generates digits without bound.

5.4 Number representations

Familiar representations use a *fixed-radix* base; think of

$$\pi = 3 + \frac{1}{10}\left(1 + \frac{1}{10}\left(4 + \frac{1}{10}\left(1 + \frac{1}{10}\left(5 + \cdots\right)\right)\right)\right)$$

as the number $(3;1,4,1,5,\ldots)$ in fixed-radix base $\mathcal{F}_{10} = (\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \ldots)$.
Similarly, think of expansion

$$\pi = 2 + \frac{1}{3}\left(2 + \frac{2}{5}\left(2 + \frac{3}{7}\left(\cdots\left(2 + \frac{i}{2i+1}\left(\cdots\right)\right)\right)\right)\right)$$

as the number $(2;2,2,2,\ldots)$ in *mixed-radix* base $\mathcal{B} = (\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \ldots)$.
Computing the digits of π is then radix conversion from base $\mathcal{B}$ to base $\mathcal{F}_{10}$.
Regular representations: digit i after the point is

- in $[0,9]$, and 'maximal fraction' is $(0;9,9,9\ldots) = 1$, for $\mathcal{F}_{10}$;
- in $[0,2i]$, and maximal fraction is $(0;2,4,6\ldots) = 2$, for $\mathcal{B}$.

5.4 Converting to fixed-radix base

Digits in base $\mathcal{F}_{10}$ of number x (assume $0 \leq x < 10$):

- first digit $d = \lfloor x \rfloor$
- remainder is $x - d$
- remaining digits obtained from $10 \times (x - d)$

In Haskell:

$$\begin{aligned} &\mathit{decimal}\ x = d : \mathit{decimal}\ (10 \times (x - \mathit{fromIntegral}\ d)) \\ &\quad \textbf{where}\ d = \mathit{floor}\ x \end{aligned}$$

Doesn't get very far with *Float* or *Double*: needs arbitrary precision.
We have to do this for a number x represented in $\mathcal{B}$.

5.4 Operations in mixed-radix base

For number $x = (a_0; a_1, a_2, a_3 \ldots)$ in $\mathcal{B}$,

- $\lfloor x \rfloor$ is either a_0 or $a_0 + 1$, depending on whether remainder $(0; a_1, a_2, a_3 \ldots)$ is in $[0, 1)$ or $[1, 2)$
- (remainder cannot be 2, for irrational x, like π)
- so need to buffer any 9s produced, in case of carries
- multiplying x by 10 can be achieved by multiplying each a_i by 10
- this typically yields an *irregular* representation
- for *finite* number, regularize from right to left, carrying leftwards

For *infinite* number, regularization needs to be left to right.
This can be done by *streaming*.

5.4 Streaming: the idea

Consider conversion of *infinite* representations from base $\mathcal{F}_m$ to $\mathcal{F}_n$. Key idea:

> *first few input digits determine first few output digits.*

So consume first few, produce first few, continue with remainder.
Maintain additional information, representing the function from the remaining inputs to the remaining outputs: with input

$$x = \frac{1}{m}\left(a_0 + \frac{1}{m}\left(a_1 + \cdots\right)\right)$$

after $a_0, a_1, \ldots, a_{i-1}$ have been consumed and $b_0, b_1, \ldots, b_{j-1}$ produced,

$$x = \frac{1}{n}\left(b_0 + \frac{1}{n}\left(b_1 + \cdots + \frac{1}{n}\left(b_{j-1} + v \times \left(u + \frac{1}{m}\left(a_i + \frac{1}{m}\left(a_{i+1} + \cdots\right)\right)\right)\right)\right)\right)$$

Represent function $\lambda x \to v \times (u + x)$ by the pair (u, v) of rationals.
Initially, $i = j = 0$ and $(u, v) = (0, 1)$.
Commit when $v \times (u + 0)$ and $v \times (u + 1)$ have same first digit in base n.

5.4 Streaming: an example

For example, $1/e = 0.100221\ldots$ in $\mathcal{F}_3$, and $0.240\ldots$ in $\mathcal{F}_7$.
First three input digits 100 determine first output digit 2:

$$0.2_7 < 0.100_3 < 0.101_3 < 0.25_7$$

So consume three input digits, produce an output digit; continue with remainder.
First few states of the conversion:

a_i		1		0		0				2		2			1				
u, v	$\frac{0}{1}, \frac{1}{1}$	$\frac{1}{1}, \frac{1}{3}$	$\frac{3}{1}, \frac{1}{9}$	$\frac{9}{1}, \frac{1}{27}$	$\frac{9}{7}, \frac{7}{27}$	$\frac{41}{7}, \frac{7}{81}$	$\frac{137}{7}, \frac{7}{243}$	$\frac{418}{7}, \frac{7}{729}$	$\frac{10}{49}, \frac{49}{729}$										
b_j										2								4	

First *safe* state is $(u, v) = (\frac{9}{1}, \frac{1}{27})$, the first for which we have:

$$\lfloor 7 \times v \times u \rfloor = \lfloor 7 \times \tfrac{1}{27} \times \tfrac{9}{1} \rfloor = 2 = \lfloor 7 \times \tfrac{1}{27} \times (\tfrac{9}{1} + 1) \rfloor = \lfloor 7 \times v \times (u + 1) \rfloor$$

5.4 Streaming: the pattern

```
stream :: (b→c) → (b→c→Bool) → (b→c→b) → (b→a→b) → b → [a] → [c]
stream next safe prod cons z (x : xs)
  = if safe z y then y : stream next safe prod cons (prod z y) (x : xs)
                else   stream next safe prod cons (cons z x) xs
    where y = next z
```

In particular,

```
convert :: (Integer, Integer) → [Integer] → [Integer]
convert (m, n) xs = stream next safe prod cons init xs where
  (m', n')     = (fromInteger m, fromInteger n)
  init         = (0 % 1, 1 % 1)
  next (u, v)  = floor (u × v × n')
  safe (u, v) y = (y == floor ((u + 1) × v × n'))
  prod (u, v) y = (u − fromInteger y / (v × n'), v × n')
  cons (u, v) x = (fromInteger x + u × m', v / m')
```

5.4 Back to π

Can use streaming to regularize an infinite representation.
But there is a more direct approach to computing the digits of π.

$$\begin{aligned}\pi &= 2 + \tfrac{1}{3}\Big(2 + \tfrac{2}{5}\Big(2 + \tfrac{3}{7}\Big(\cdots\Big(2 + \tfrac{i}{2i+1}\Big(\cdots\Big)\Big)\Big)\Big)\Big) \\ &= \Big(2 + \tfrac{1}{3}\times\Big)\Big(2 + \tfrac{2}{5}\times\Big)\Big(2 + \tfrac{3}{7}\times\Big)\cdots\Big(2 + \tfrac{i}{2i+1}\times\Big)\cdots\end{aligned}$$

—composition of infinite series of *linear fractional transformations* $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$.

- fixpoint of $(2 + \frac{1}{3}\times)$ is 3, fixpoint of $(2 + \frac{1}{2}\times)$ is 4
- so each of these LFTs maps interval $[3,4]$ onto a subinterval of itself
- each LFT shrinks by a constant factor (indeed, by at least $\frac{1}{2}$)
- so compositions of such LFTs converge to a point in $[3,4]$.

Finding that point is another *change of representation*,
from infinite sequences of LFTs to infinite sequences of decimal digits.

5.4 Streaming π

- represent each *input* LFT by a 2-by-2 matrix of integers

$$\left[\left(\begin{smallmatrix} 1 & 6 \\ 0 & 3 \end{smallmatrix}\right), \left(\begin{smallmatrix} 2 & 10 \\ 0 & 5 \end{smallmatrix}\right), \left(\begin{smallmatrix} 3 & 14 \\ 0 & 7 \end{smallmatrix}\right), \ldots \right]$$

- *state* is another LFT z
- *initial* state is identity LFT, $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$
- z is *safe* if image under z of $[3,4]$ all has same integer part, n

$$n \leq \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \times [3,4] = \left[\tfrac{3q+r}{3s+t}, \tfrac{4q+r}{4s+t}\right] < n+1$$

- then *produce* digit n, and multiply state by $\left(\begin{smallmatrix} 10 & -10n \\ 0 & 1 \end{smallmatrix}\right)$, inverse of the LFT $x \mapsto n + \frac{x}{10}$
- otherwise *consume* next LFT, by matrix multiplication

5.4 Program for π

$$
\begin{aligned}
&pi = stream\ next\ safe\ prod\ cons\ init\ lfts\ \textbf{where}\\
&\quad init \;\; = unit\\
&\quad lfts \;\; = [\,(k, 4\times k+2, 0, 2\times k+1) \mid k \leftarrow [1..]\,]\\
&\quad next\ z \;\; = floor\ (extr\ z\ 3)\\
&\quad safe\ z\ n \;\; = (n == floor\ (extr\ z\ 4))\\
&\quad prod\ z\ n = comp\ (10, -10\times n, 0, 1)\ z\\
&\quad cons\ z\ z' = comp\ z\ z'
\end{aligned}
$$

where *comp* is matrix multiplication, and *extr* extracts the LFT from a matrix $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$, taking x to $(q\times x + r)/(s\times x + t)$.
Obfuscated program obtained from this by inlining definitions, and observing that invariant $s = 0$ holds in all our LFTs $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$.

5.4 Reasoning about *stream*

For finite sequences, express change of representation by *abstraction*:

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

followed by *reification*:

$$
\begin{aligned}
&unfoldr :: (b \to Bool) \to (b \to c) \to (b \to b) \to b \to [c] \\
&unfoldr\ p\ f\ g\ z = \textbf{if}\ p\ z\ \textbf{then}\ f\ z : unfoldr\ p\ f\ g\ (g\ z)\ \textbf{else}\ [\,]
\end{aligned}
$$

and $convert\ p\ f\ g\ h\ z\ xs = unfoldr\ p\ f\ g\ (foldl\ h\ z\ xs)$.
Sometimes this process can be *streamed*: if state z satisfies

$$\exists y\ .\ \forall xs\ .\ convert\ p\ f\ g\ h\ z\ xs = y : \ldots$$

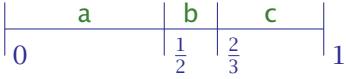
then it is safe to produce y from z before consuming any more of xs.

5.4 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:

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

and $[\frac{9}{32}, \frac{61}{216}]$ contains 0.01001 and no shorter binary fraction.
For efficiency, we wish to *stream* the output.
Lecturing on arithmetic coding led us to the streaming abstraction.

5.4 Further reading

- "A Spigot Algorithm for the Digits of π", Stanley Rabinowitz and Stan Wagon, *American Mathematical Monthly*, **102**:195–203, 1995
- "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

My papers are available from my webpage:

`http://www.cs.ox.ac.uk/jeremy.gibbons`