7. Lazy evaluation and infinite lists

**Summary**: This lecture introduces lazy evaluation and infinite lists in functional languages.

**cs123 notes**: Lecture 19
Reduction order

- Consider the following expression

\[
\text{fst (sqr 4, sqr 2)}
\]

- There are two ways in which this might be evaluated

  - *Arguments first* or **innermost reduction**:
    - also known as **applicative-order reduction** or **eager evaluation** or **call-by-value**

\[
\begin{align*}
\text{fst (sqr 4, sqr 2)} & \Rightarrow \text{fst (4 * 4, sqr 2)} \\
& \Rightarrow \text{fst (16, sqr 2)} \\
& \Rightarrow \text{fst (16, 2 * 2)} \\
& \Rightarrow \text{fst (16, 4)} \\
& \Rightarrow 16 
\end{align*}
\]

  - *Function first* or **outermost reduction**:
    - also known as **normal-order reduction** or **lazy evaluation**

\[
\begin{align*}
\text{fst (sqr 4, sqr 2)} & \Rightarrow \text{sqr 4} \\
& \Rightarrow 4 * 4 \\
& \Rightarrow 16 
\end{align*}
\]

- Note that the two schemes will **never** give different answers
  - although sometimes outermost gives an answer when innermost doesn’t
Lazy evaluation

- Most programming languages use eager evaluation
- Haskell uses lazy evaluation
- The principle of lazy evaluation is

  “evaluate an expression
  only if you know you have to”

- e.g. \(\text{sqr}\ 2\) above never gets evaluated

- This has several consequences
  - fewer reductions
  - avoids non-termination
    - e.g. \(\text{fst}\ (\text{sqr}\ 4, \ [1\ ..\])\)
  - allows a more flexible style of programming
  - although lazy evaluation is usually somewhat slower due to the
    bookkeeping required to keep track of which expressions have been
    evaluated
  - side-effects like printing and modifying variables don't fit well,
    because the sequence of operations depends on how an expression is
    used
    [eager functional languages usually do include such side-effects, but
    they require laziness to be manually programmed]
  - effects in Haskell are instead provided via monads (next topic)
An example: Boolean functions

- From the Prelude:

```
(&&) :: Bool -> Bool -> Bool
-- z && x returns True iff
-- both z and x are True
False && x = False
True && x = x
```

- Note that `&&` checks its left argument first, then checks its right argument **only if the left is True**
  - so if `z = last [True | x <- [1 ..]]`

    `z && False = <infinite loop>`

    - but

    `False && z = False`

- So `&&` isn’t commutative!
- This “Boolean left-laziness” is common to many programming languages
An example: Boolean list functions

• Lazy evaluation generalises this idea to every definition in a program
  – for example

    and :: [Bool] -> Bool
    -- and xs returns True iff
    -- every element on xs is True
    and [] = True
    and (x : xs) = x && and xs

• and checks its list in order from the first element onwards, and stops as soon as any element is False
  – so

    and [True | x <- [1 ..]] = <infinite loop>

  – but

    and (False : [True | x <- [1 ..]])

    ⇒ False && and [True | x <- [1 ..]]

    ⇒ False

• So here again lazy evaluation returns a result where eager evaluation wouldn't
Infinite lists

- The expression \([m \ldots]\) denotes an infinite list

- What can we do with this?

- We can’t examine the whole list
  - that would take an infinite amount of time!

- We can examine any finite part of the list
  - for example

\[
\text{head } [m \ldots] = m
\]

\[
[m \ldots] !! k = m + k
\]

\[
\text{take } k [m \ldots] = [m \ldots m + k - 1]
\]

\[
\text{zip } [m \ldots] "abc"
\]
\[
= [(m, 'a'), (m+1, 'b'), (m+2, 'c')]
\]

- The system “knows” how far it needs to evaluate the list
  - remember the principle of lazy evaluation
Infinite lists

• But care is needed
  – for example

  \[4 \ `\text{elem}` \ [0, 2 ..] = \text{True}\]

  \[3 \ `\text{elem}` \ [0, 2 ..] = \text{<infinite loop>}\]

  \[\text{filter} (< 10) [x ^ 2 \mid x \leftarrow [1 ..]]\]
  \[= \text{[1, 4, 9 <infinite loop>}\]

• The system doesn’t know that all of the subsequent numbers will fail the tests

• We need to incorporate the information that the numbers on the infinite lists are always increasing
  – one way is to use \text{takeWhile}

  \[3 \ `\text{elem` \text{takeWhile} (<= 3) \ [0, 2 ..]\]
  \[= \text{False}\]

  \[\text{takeWhile} (< 10) [x ^ 2 \mid x \leftarrow [1 ..]]\]
  \[= \text{[1, 4, 9]}\]
An example: prime numbers

• Consider the following functions.

```haskell
factors :: Int -> [Int]
-- factors n returns a list
-- containing the factors of n
factors n = [k | k <- [1 .. n], n `mod` k == 0]

isPrime :: Int -> Bool
-- isPrime n returns True if n is prime
isPrime n = factors n == [1, n]

primesUpto :: Int -> [Int]
-- primesUpto n returns a list containing
-- the primes up to n
primesUpto n = [k | k <- [2 .. n], isPrime k]
```

• But suppose we want the first \( n \) primes?
  
  − use an infinite list

```haskell
n'primes :: Int -> [Int]
-- n'primes n returns a list containing
-- the first \( n \) primes
n'primes n = take n [k | k <- [2 ..], isPrime k]
```
iterate

- The function \texttt{iterate} takes a function \( f \) and generates an infinite list of repeated applications of \( f \) to a value \( x \)

\[
\text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]
\]

-- iterate \( f \) \( x \) returns an infinite list of
-- repeated applications of \( f \) to \( x \)

\[
\text{iterate} \ f \ x = x : \text{iterate} \ f \ (f \ x)
\]

- note: recursion with no base case!

- For example

\[
\text{iterate} \ (+1) \ 1 = [1, 2, 3, 4, 5, ...]
\]

\[
\text{iterate} \ (*2) \ 1 = [1, 2, 4, 8, 16, ...]
\]

\[
\text{iterate} \ (\text{`div` 10}) \ 271
\]

\[
= [271, 27, 2, 0, 0, 0, 0, ...]
\]

- \texttt{iterate} captures a common pattern used in infinite lists
Generate-and-test algorithms

- `iterate` is often used in **generate-and-test algorithms**
- Consider the function `remainder` that takes two numbers \( x \) and \( y \) and returns the remainder of \( x \) divided by \( y \)

```haskell
remainder :: Int -> Int -> Int
-- pre: \( x \geq 0 \) \&\& \( y > 0 \)
-- remainder \( x y \) returns the remainder
-- of \( x \) divided by \( y \)
remainder x y = head (dropWhile (>= y) (iterate f x))
  where f u = u - y
```

- Consider the function `digits` that takes a number \( x \) and returns a list containing the digits in \( x \)

```haskell
digits :: Int -> [Int]
-- pre: \( x > 0 \)
-- digits \( x \) returns the digits in \( x \)
digits = reverse .
  map (\`mod\` 10) .
  takeWhile (> 0) .
  iterate (\`div\` 10)
```

- These two functions work in similar ways
  - `iterate` generates candidate elements for the result
  - the other functions test the candidates to see which ones we actually want
- The use of an infinite list allows us to completely separate the generation of candidates from the testing
- In an eager language, we would have to explicitly interleave these operations
  - generate a candidate, test it, generate another candidate, test it, …
  - a much less “modular” definition
An example: the Sieve of Eratosthenes

• The Sieve was the first algorithm for generating prime numbers efficiently
  – Eratosthenes was a Greek mathematician

1. Write down the numbers 2, 3, 4, 5, …
2. Mark the first element p as prime
3. Remove all multiples of p from the list
4. Return to Step 2

• It operates like this:

\[
\begin{align*}
2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 & \quad 12 & \quad 13 & \quad 14 & \quad 15 & \quad 16 & \quad 17 & \ldots \\
\text{– so 2 is prime} & \\
3 & \quad 5 & \quad 7 & \quad 9 & \quad 11 & \quad 13 & \quad 15 & \quad 17 & \quad 19 & \quad 21 & \quad 23 & \quad 25 & \quad 27 & \quad 29 & \quad 31 & \ldots \\
& \text{(no multiples of 2)} \\
\text{– so 3 is prime} & \\
5 & \quad 7 & \quad 11 & \quad 13 & \quad 17 & \quad 19 & \quad 23 & \quad 25 & \quad 29 & \quad 31 & \quad 35 & \quad 37 & \quad 41 & \quad 43 & \ldots \\
& \text{(no multiples of 2 or 3)} \\
\text{– so 5 is prime} & \\
7 & \quad 11 & \quad 13 & \quad 17 & \quad 19 & \quad 23 & \quad 29 & \quad 31 & \quad 37 & \quad 41 & \quad 43 & \quad 47 & \quad 49 & \quad 53 & \ldots \\
& \text{(no multiples of 2, 3 or 5)}
\end{align*}
\]

etc.
The Sieve of Eratosthenes

• The Sieve is implemented by the function `sieve`

```haskell
sieve :: [Int] -> [Int]
-- sieve xs returns p and sieves xs
sieve (p : xs)
    = p : sieve [x | x <- xs, x `mod` p /= 0]
```

```haskell
all'primes :: [Int]
-- all'primes returns a list
-- containing all primes
all'primes = sieve [2 ..]
```

• `all'primes` can then be used in other functions

```haskell
-- primesUpto n returns a list containing
-- the primes up to n
primesUpto n = takeWhile (<= n) all'primes

-- n'primes n returns a list containing
-- the first n primes
n'primes n = take n all'primes
```

− the generation of primes and the selection of elements for the result
list are again separate operations

• Compare the performance of these definitions with the previous
definitions