| galois |

Background: Sequence Comprehensions

What is a sequence comprehension?

Cryptol uses a sequence comprehension to manage state changes during a computation. A simple example of a sequence comprehension is this:

```
Cryptol> [x | x <- [1...] | y <- [1..10]]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The outer `[ ]` means that a sequence is being constructed. The leftmost `x`, which is to the left of the leftmost `|`, means that values of `x` are being appended to the sequence. Those values come from the construct `x <- [1...]` which means `x` values are taken from an infinite sequence. The variable `y` controls how many of those values `x` will take. Variable `y` gets a value every time `x` gets a value due to the rightmost `|`. Thus, when `x` gets 1, `y` gets 1, when `x` gets 2, `y` gets 2, and so on. But `y` gets only up to 10. At that point computation stops, but not before the output sequence grew from empty to `[1]` to `[1,2]`, to `[1,2,3]`, up to `[1,2,3,4,5,6,7,8,9,10]` due to the 10 `x` values that were appended to the sequence. Now replace the rightmost | with , and observe the output of this:

```
Cryptol>  [ (x,y) | x <- [10..12] , y <- [1..4]]
[(10, 1), (10, 2), (10, 3), (10, 4), (11, 1), (11, 2), (11, 3),
 (11, 4), (12, 1), (12, 2), (12, 3), (12, 4)]
```

The tuple `(x,y)` is appended to the sequence, per iteration, but `x` and `y` do not take values synchronously, rather as follows: values for `x` are taken from the sequence `[10,11,12]` but for each value of `x` all the values for `y` are taken from the sequence `[1,2,3,4]` and paired with a static `x` as many times as there are `y` values. A tuple `(x,y)` is appended for each such pairing of `x` and `y`. Another simple example:

```
Cryptol>  [ (x,y,z) | x <- [10,11] , y <- [20,21], z <- [30,31]]
[(10, 20, 30), (10, 20, 31), (10, 21, 30), (10, 21, 31),
 (11, 20, 30), (11, 20, 31), (11, 21, 30), (11, 21, 31)]
```

The rightmost `|` can be added to control the number of tuples is appended to the sequence.

```
Cryptol>  [ (x,y,z) | x <- [10,11] , y <- [20,21] | z <- [30,31,32]]
[(10, 20, 30), (10, 21, 31), (11, 20, 32)]
```

Now things get interesting. The elements that are appended to the sequence are states of computation. But next states typically depend on current or previous states. The following is a simple illustration of how this is managed with sequence comprehensions.

```
Cryptol> let s = [1]#[ x+y | x <-s | y <- [1]#s ]
Cryptol> take `{10} s
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Sequence `s` begins with just one element, namely `1`, due to `[1]#[ x+y |…]` and `[ x+y |…]` is initially empty. Next, the first `x+y` is appended to `s`. The first `x` is 1 which is the only element existing for `s` at this point. The first `y` comes from the sequence `[1]#s` which starts

off as `[1]#[1]` which is `[1,1]`. So, the first `y` is `1` and the first `x+y` is `2` which is appended to `s`. For the second `x+y`, s is now `[1,2]`. Variable `x` has already taken `1` from `s` so now it takes `2`. Variable `y` is now taking elements from `[1,1,2]` since `2` has been appended to `s`. Since `y` already took the first `1` it now takes the second `1`. Thus `x+y` is equal to `2` (from `x`) + 1 (from `y`) = `3` which is appended to `s`. On the next iteration `s = [1,2,3]`, `x` gets the `3`, `y` gets the `2` and the sum `3+2` is appended to `s`. On the next iteration `s = [1,2,3,5]`, `x` gets `5`, `y` gets `3` and `8` is appended to `s`. This is an infinite computation but no time is spent making the computation because the elements in `s` are computed only on demand. In this case, `take `{10} s` demands that the first 10 of the elements of `s` be computed and displayed. It should be evident that the above comprehension computes all Fibonacci numbers. To find, say, the 30th number in the sequence do this: `Cryptol> s@29` (the 0th element is in the first position of s so the 29th element of the sequence is in the 30th position).

Now consider a classic problem in logic: prove that if the number of pigeon holes is less than the number of pigeons, and pigeons are distributed over the holes, then at least one hole has more than one pigeon. First consider a representation of the problem. Let a sequence `ph` represent a collection of pigeon holes. Each element of `ph` will be a number of pigeons in the hole corresponding to that element. The total number of pigeons in holes can be calculated with this:

```
Cryptol> :s base=10
Cryptol> let ph = [1,1,1,0,1,2,2,1,1]   // Just for illustration
Cryptol> let s = [0]#[ x+y | x <- ph | y <- s ]
Cryptol> s
[0, 1, 2, 3, 3, 4, 6, 8, 9, 10]
Cryptol> let npigeons = s ! 0
Cryptol> npigeons
10
```

Now write a comprehension that returns True if and only if all elements of `ph` are no greater than 1:

```
Cryptol> :s base=2
Cryptol> let nh = [True]#[ (x <= 1) /\ y | x <- ph | y <- nh ]
Cryptol> nh
0b1111110000
Cryptol> let pigeonsOK = nh ! 0
Cryptol> pigeonsOK
False
```

The number of pigeon holes is:

```
Cryptol> let nholes = length ph
Cryptol> nholes
9
```

The property of interest is this:

```
Cryptol> if (npigeons > nholes) then pigeonsOK == False else True
```

In a subsequent unit the above will be rewritten so that this property may be proved.

Now consider a comprehension that reverses the elements of a sequence `lst`:

```
Cryptol> let z = [lst] # [[b]#(tail (a >> 1)) | a <- z | b <- lst ]
Cryptol> let rev = z ! 0
```

The following shows the value of `z` given input sequence `lst = [16,42,23,77]`

```
Cryptol> z
[[16,42,23,77],[16,16,42,23],[42,16,16,42],[23,42,16,16],[77,23,42,16]]
Cryptol> z ! 0
[77,23,42,16]
```

The following creates a sequence of 16 numbers, beginning with the first one, x, as input, doubled, then followed by numbers that are double the previous one.

```
Cryptol> let x = 3
Cryptol> let iv = [ 2*lr | lr <- [x] # iv | k  <- [1 .. 16]]
Cryptol> iv
[6,12,24,48,96,192,384,768,1536,3072,6144,12288,24576,49152,98304,196608]
```

The following is an infinite version of the above

```
Cryptol> let x = 2
Cryptol> let  iv = [2*lr | lr <- [x]#iv | k <- [1...]]
Cryptol> iv
[4, 8, 16, 32, 64, ...]
Cryptol> let res = take `{12} iv
[4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
```

Consider the following comprehension (but see below warning before running it):

```
Cryptol>  let ys = [ x + y | x <- ys | y <- [0] # ys ]
Cryptol> :t ys
ys : {n, a} (Ring a, Literal 0 a) => [n]a
```

So, from the type, `ys` exists but it has no elements

```
Cryptol> ys == []
True
```

This is because `x` never gets a value from `x <- ys`.

Warning: computing `ys` may cause all of memory to be consumed eventually, then a crash, so kill cryptol as soon as you can in case you want to try the above.

Finally, consider a problem of supplying fuel to a race car traversing a race circuit. There are a number of fueling stations located at various points around the circuit. Suppose the amount of fuel at each fueling station is different but the total fuel around the circuit is exactly what is needed by the race car to make one complete circuit. Find the point on the circuit from which a race car, with an empty tank, may make one complete circuit without running out of fuel by tanking up at every fueling station in the circuit. Assume the tank is large enough to hold enough fuel to complete the circuit. The location of each fueling station is specified by a sequence of increasing ratios, each of which states the percentage of the total circuit distance that the fueling station, corresponding to a sequence element, is from the starting line. Here is an example:

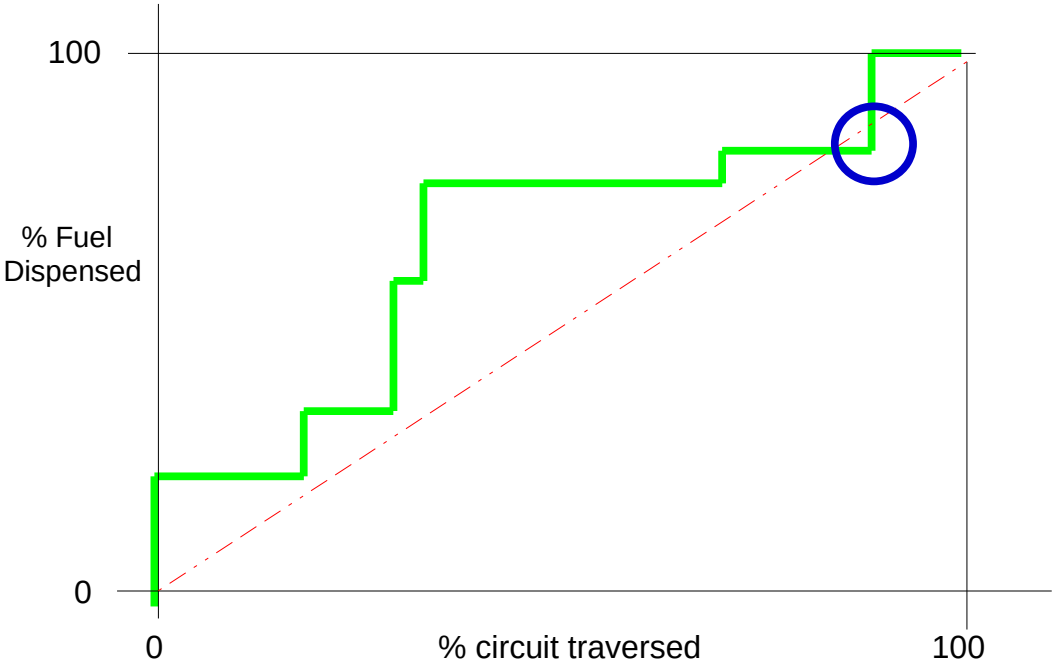
```
Cryptol> let loc = [(ratio 2 10),(ratio 3 10),(ratio 45 100),(ratio 49 100),
                    (ratio 71 100),(ratio 78 100),(ratio 81 100),(ratio 98 100)]
```

The highest ratio in `loc` must be no greater than `(ratio 1 1)`.

The amount of fuel a station contains is stated as a percentage of the fuel needed to complete one circuit. This information is contained in a sequence where the station at position `x` corresponds to the location at position `x` in the location sequence. Here is an example:

```
Cryptol> let gas = [(ratio 1 10),(ratio 13 100),(ratio 1 10),(ratio 15 100),
                    (ratio 23 100),(ratio 7 100),(ratio 17 100),(ratio 5 100)]
```

The sum of the numbers in `gas` must be 1.



The red line in the above Figure depicts the consumption of fuel by the race car over the entire circuit. The green line depicts the fuel that has been dispensed to the race car. The vertical jumps state the fuel supplied by a fueling station at the station's position on the circuit. As long as the green line is above the red line, the race car has fuel and may continue on the circuit. If the green line crosses the red line, the race car has run out of fuel. From the above, it is clear that moving the red line to the right as far as it will go while intersecting the green line and keeping the same slope, the red line will finish moving at the station that is the solution to the problem. In the above Figure that is the station indicated inside the blue circle. Let `fuel_seen` be the percentage of fuel dispensed to the race car so far at the arrival of a fuel station. Let `track_covered` be the percentage of the track covered by the race car at that location. If `track_covered – fuel_seen` is negative, there is a fuel deficit and the race car will run out of fuel. Creating a sequence comprehension one can find the deficit at each station and, considering each station in order, if that deficit is the greatest seen so far then record that station as the current correct station to start from. Thus, the following structure, a tuple, will be updated by the comprehension: `(fuel_seen, deficit, station)`. The station will be identified by its position in the input sequences `p` and `g`. Initially, the tuple is `(0,0,-1)` where the -1 just indicates a lack of station candidate so far and the two 0s represent the origin of the race car in the circuit. Begin building the comprehension like this:

```
let s = (0,0,-1)#[ … | fs <- gas | ps <- loc | ss <- s | i < [0 ...]]
```

Comprehension `s` is finite because its length depends on the length of `gas` and `loc`. Therefore, although i is taken from an infinite sequence, only the numbers demanded are given to `i` and are used to identify the station from which the race car should begin to prevent running out of fuel. Variable `ss` is the current state of execution. The current deficit is the current track coverage, given by ps, minus the current fuel, given by the 0^{th} element of the tuple ss. If ps-ss.0 is greater than the current, saved, deficit, then a better station than the saved one has been identified. That station position and the deficit are recorded in the next tuple to be added to s along with the updated amount of fuel dispensed so far. The ... can now be constructed to update the tuple like this:

```
if (ps-ss.0 > ss.1) then (ss.0+fs, ps-ss.0, I) else (ss.0+fs, ss.1, ss.2)
```

All together the comprehension to solve the problem is

```
Cryptol> let s = [(0,0,-1)]#[(if (ps-ss.0 > ss.1) then (ss.0+fs, ps-ss.0, i)
                                                  else (ss.0+fs, ss.1, ss.2))
                            | fs <- gas | ps <- loc | ss <- s | i <- [0 …]]
```

Note: before cutting and pasting the above three `lets` to Cryptol, make each `let` statement just one line by concatenating all lines. Then take a look at s:

```
Cryptol> s
[((ratio 0 1), (ratio 0 1), -1), ((ratio 1 10), (ratio 1 5), 0),
 ((ratio 23 100), (ratio 1 5), 0), ((ratio 33 100), (ratio 11 50), 2),
 ((ratio 12 25), (ratio 11 50), 2), ((ratio 71 100), (ratio 23 100), 4),
 ((ratio 39 50), (ratio 23 100), 4), ((ratio 19 20), (ratio 23 100), 4),
 ((ratio 1 1), (ratio 23 100), 4)]
```

So, the station to start from is at `(s!0).2` and the location of that station is `loc@(s!0).2`.

```
Cryptol> (s!0).2
4
Cryptol> loc@(s!0).2
(ratio 71 100)
```