Background: Functions

A function is a mapping from one or more input values to an output value. Here is a simple example:

```
  Cryptol> let f x = x^^2
```

Set base = 10 then run the above like this

```
  Cryptol> f 5
  25
```

The inferred type of `f` is `f : {a} (Ring a) => a -> a`. The type can be changed, for example, with:

```
  Cryptol> let f (x:[8]) = x^^2.
```

The type signature of `f` is then `f : [8] -> [8]`.

Usually, a function uses several subordinate computations in one or more `where` blocks and outputs from some manipulation of those subordinate computations. For example:

```
  Cryptol> let g x = a+b where (a:[32]) = x*2 ; (b:[32]) = x-1
  Cryptol> g 10
  29
```

The declaration of a function may require Cryptol to infer inputs are sequences or tuples or some other structure. For example:

```
  Cryptol> let headx x = x @ 0
```

where the operator `@` requires `x` to be of type `[n]a` since the type signature of `@` is

```
  (@) : {n, a, ix} (Integral ix) => [n]a -> ix -> a
```

An example of using headx is this:

```
  Cryptol> headx [2,3,4,5,6]
  2
```

Here is a simple padding function for 512 bit blocks (set base=16):

```
Cryptol> let pad (key:[64]) = key # (0:[1]) # sz where sz = (length key):[447]
Cryptol> :t pad
pad : [64] -> [512]
Cryptol> pad 0x1234567812345678
0x1234567812345678000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000040
```

Swapping in Cryptol requires just one line of code:

```
  Cryptol> let swap [x,y] = [y,x]
  Cryptol> swap [1,2]
  [2,1]
```

Observe the input to swap is one argument: a sequence of 2 elements. So, arguments can be all kinds of types.

Cryptol can deal with infinite sequences fairly easily. The next example shows how to merge two non-decreasing infinite sequences of numbers.

```
Cryptol> let merge x y = if (x@0) < (y@0) then [x@0] # merge (tail x) y else [y@0] # merge x (tail y)
Cryptol> let mrg = merge [1,3 ...] [2,4 ...]
Cryptol> mrg
[1, 2, 3, 4, 5, ...]
Cryptol> take `{20} mrg
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Cryptol> :t merge
merge : {a} (Cmp a) => [inf]a -> [inf]a -> [inf]a
```

Writing `merge` like this for finite sequences would be a problem because `merge` would have to deal with changing lengths of input and output and Cryptol would not accept such a definition unless something strange is done. Fortunately, the strange way is not difficult to manage and will be presented later. It is also easy to multiply all numbers in an infinite sequence:

```
Cryptol> let times s m = [(s@0)*m]#times (tail s) m
Cryptol> :t times
times : {a} (Ring a) => [inf]a -> a -> [inf]a
Cryptol> times mrg 7
[7, 14, 21, 28, 35, ...]
Cryptol> take `{20} times mrg 7
[7,14,21,28,35,42,49,56,63,70,77,84,91,98,105,112,119,126,133,140]
```

Again, an infinite type is inferred. But some problems require infinite sequences. For example, consider the problem of finding a Hamming sequence for a given sequence of prime numbers. A Hamming sequence is an increasing sequence of numbers such that all the prime factors of each number are members of the given sequence of prime numbers which we call `primes`. A Hamming sequence consists of all such numbers and only such numbers. To find a Hamming sequence observe that after pulling the first number of the Hamming sequence the remaining numbers can be partitioned into two sequences: one containing the smallest numbers from `primes` as a factor and the other containing all other numbers in the Hamming sequence. The second sequence is a Hamming sequence involving the same input sequence `primes` except without the smallest number from `primes`. The first sequence is a Hamming sequence where all numbers are multiplied by the smallest number from `primes`. Here is an example for the sequence of prime numbers `primes=[3,5,11]`. The Hamming sequence is:

```
3,5,9,11,15,25,27,33,45,55,75,81,99,121,125,135,165,225,243,275,297...
```

Remove the smallest (leftmost) number (3) to get this sequence:

```
5,9,11,15,25,27,33,45,55,75,81,99,121,125,135,165,225,243,275,297,605...
```

Partition according to whether a number has a factor of 3 (in this example) and factor:

```
5,11,25,55,121,125,275,605,625,1331,1375,3025,3125,6655...
and
3*(3,5,9,11,15,25,27,33,45,55,75,81,99,121,125,135,165,225,243...)
```

So, the Hamming sequence for `[3,5,11]` is `3` followed by the `merge` of the Hamming sequence for `[5,11]` and `3` times the Hamming sequence for `[3,5,11]`. Generally, if `primes` is greater than 1 in length, the Hamming sequence, given `primes`, is this:

```
[head primes]#merge (times (ham primes) (head primes)) (ham (tail primes))
```

But, if the length of the `primes` sequence is 1 the Hamming sequence is

```
times (ham primes) (head primes)
```

So, the first attempt at writing a solution to the Hamming sequence problem is

```
ham primes =
  if (length primes) == 1
  then [head primes] # times (ham primes) (head primes)
  else [head primes] # merge (times (ham primes) (head primes)) (ham (tail primes))
```

Unfortunately, `(ham (tail primes))` forces an unsolvable constraint error due to calls to `ham` with shrinking `primes` sequence lengths. A workaround is to keep the `primes` sequence the same length but use an index variable `idx` to identify where the beginning of the shortened primes sequence is: if `idx` is `0` then the beginning is the beginning of `primes`, if `idx` is `(length primes)-1` the shortened primes sequence is just the last number of the original `primes` sequence, and so on. Now also, `(head primes)` becomes `(primes @ idx)` and `(ham (tail primes))` becomes `(ham primes (idx+1))`. With this workaround the solution to the Hamming sequence solution looks like this:

```
Cryptol> let ham primes idx = \
  if (length primes)-1 == idx  // primes has length 1 \
  then [primes @ idx] # times (ham primes idx) (primes @ idx) \
  else [primes @ idx] # merge (times (ham primes idx) (primes @ idx)) (ham primes (idx+1))
```

The `\` character above allows copying the above from the `let` to the end and pasting to the Cryptol console. There must be no space characters after any `\`.

```
Cryptol> :t ham
ham :
  {n, a, b}
    (Cmp a, Eq b, Integral b, Ring a, Literal (max 1 n) b, fin n) =>
    [n]a -> b -> [inf]a
```

The inferred type signature for `ham` shows a finite sequence of numbers is accepted as input plus an Integer `b`. The output is an infinite sequence. This should be changed so that `a` is Integral as well as `b` as literals such as `(ratio 3 1)` should not be allowed. The easiest way to do this is add a line that fixes a broken result if an empty sequence is input. The modified `ham` is embedded in a function named `hamming` to avoid worry about adding a `0`, which means little to a user, to the parameter sequence of, now, hamming:

```
Cryptol> let hamming primes = ham primes 0 \
  where \
    ham p idx = \
      if ((length p) == 0) then [-1,-1 ...] \
      else if (length p)-1 == idx then \
           [p @ idx] # times (ham p idx) (p @ idx) \
      else \
           [p @ idx] # merge (times (ham p idx) (p @ idx)) (ham p (idx+1))
Cryptol> :t hamming
hamming : {n, a} (Literal 1 a, Integral a, Cmp a, fin n) => [n]a -> [inf]a
Cryptol> hamming []
[-1, -1, -1, -1, -1, ...]
Cryptol> take `{20} (hamming [3,5,11])
[3,5,9,11,15,25,27,33,45,55,75,81,99,121,125,135,165,225,243,275]
```

Now consider the problem of sorting a sequence of numbers. Mergesort is a theoretically optimal way to do this: split a sequence in two, `mergesort` both sub sequences, then `merge` the sorted sub sequences to get the desired sorted sequence. Again, since sequence sizes change as the `mergesort` algorithm is called recursively, the use of infinite sequences for this finite sequence problem becomes tenable. To simplify a little, assume non-negative input sequences are to be sorted and even positions are 0,2,4… and odd positions are 1,3,5… Therefore, even positions in the sequence [10,11,12,13,14,15] hold numbers 10,12,14 and odd positions hold numbers 11,13,15. Also assume finite sequences will be padded with an infinite sequence of -1s. One of several ways to split a sequence in two is to put all literals in even positions into one subsequence and all literals in odd positions into the other. To get the odds subsequence do this:

```
  Cryptol> let splito x = s ax \
    where \
      s p = if ((p @ 0) == -1) then p \
            else if ((p @ 1) == -1) then (drop `{1} p) \
            else [(p @ 1)] # (s (drop `{2} p)) \
      ax = x # [-1,-1 ...]
```

The input is `x`, a finite sequence of literals to be split. The output is an infinite sequence, padded with `-1`s after the split literals. Padding takes place at `ax = x#[-1,-1 ...]`. The padded sequence `ax` is input to function `s` which is defined in the `where` block. If `p` is `[-1,-1 ...]` then there are no literals from the input sequence to place in the output sequence so just the padding is appended to the output sequence. This is detected by `((p @ 0) == -1)`. If, as in the next line of `s`, `((p @ 1) == -1)` then there is just one number in `p` remaining from the original input sequence and it is in the `0`th position which is an even position so `(drop `{1} p)` disposes of that number and appends the padded sequence to the output sequence. Otherwise, there are at least two literals in p that remain from the original sequence and the 1th literal (earliest odd position literal) is appended to the output sequence and two literals are removed from p and s is called on what is left. This is what `[(p @ 1)] # (s (drop `{2} p))` does. To get the evens subsequence do this:

```
  Cryptol> let splite x = s ax \
    where \
      s p = if ((p @ 0) == -1) then p \
            else [(p @ 0)] # (s (drop `{2} p)) \
      ax = x # [-1,-1 ...]
```

which is `splito` except the line with `(drop `{1} p)` is removed and `(p @ 0)` replaces `(p @ 1)` in the last line of `s`. The `merge` function above assumes every literal in the output sequence belongs to the correct output but `merge` is modified slightly now to deal with the padding.

```
  Cryptol> let merge px py = \
        if (px @ 0) == -1 then py \
        else if (py @ 0) == -1 then px \
        else if (px @ 0) < (py @ 0) then [(px @ 0)] # (merge (tail px) py) \
        else [(py @ 0)] # (merge px (tail py))
  Cryptol> let mrg = take `{8} (merge ([1,5,7,10] # [-1,-1 ...]) \
                                      ([3,7,11,14] # [-1,-1 ...]))
  [1, 3, 5, 7, 7, 10, 11, 14]
```

```
Cryptol> :t splite
splite : {n, a} (Literal 1 a, Integral a, Eq a, fin n) => [n]a -> [inf]a
Cryptol> :t splito
splito : {n, a} (Literal 1 a, Integral a, Eq a, fin n) => [n]a -> [inf]a
Cryptol> :t merge
merge : {a} (Literal 1 a, Ring a, Cmp a) => [inf]a -> [inf]a -> [inf]a
```

This is now the interesting part. The `mergesort` function is going to operate on infinite sequences but when it is finished it needs to return a sequence of length equal to the length of the input sequence. We can't just use `` take `{length lst} (mergesort lst) ``, where `lst` is the input sequence, because for `` take `{cnt} (mergesort lst) ``, `cnt` is expected to be constant. But we can get that constant from the type signature of `mergesort`. We know the input and output shape to be this: `[cnt][a] -> [cnt][a]`. Then the type variables look like this in the type signature: `mergesort : {cnt, a}`. Conditions on cnt and a are they are both finite and greater than 0. The entire type signature for mergesort then can be hand crafted as

```
mergesort : {cnt, a} (fin cnt, fin a, a >= 1, cnt >= 1) => [cnt][a] -> [cnt][a]
```

Now that we have a constant sequence length it can be used like this:

```
mergesort lst = take `{cnt} (mergesrt ax)
  where
    mergesrt p =
      if (((p@0) == -1) \/ ((p@1) == -1)) then p
      else (merge (mergesrt (splite t)) (mergesrt (splito t)))
      where
        t = take `{cnt} p
    ax = lst#[-1,-1 …]
```

Thus, `mergesrt` is applied to the padded input `ax`. Internal to `mergesrt`, `mergesrt` is applied to `(splite t)` and `(splito t)` where the purpose of `t` is to supply a relevant, finite subset of `p`, which is infinite, to `splite` and `splito` which require finite inputs. The output of `splite` and `splito` are infinite which is fine for `merge`. If one or both lowest positions of `p` are `-1` then `p` is returned – this could be one or two relevant literals followed by the padding. Then `take` takes care of that by stripping the padding. **Note:** Cryptol does not allow the type signature to be added at the console without the function the type signature applies to. Cryptol will not let `mergesort` be added at the console without the type signature and will not infer a type signature for `mergesort` because `cnt` is set in the type signature and is used in `mergesort`. The only way to run `mergesort` is to put the type signature and the `mergesort` implementation in a file and load the file via `:l`. In this directory the file `mrgsrt.cry` has the type signatures and implementations of `splite`, `splito`, `merge`, and `mergesort` tested and ready for loading. **Note:** the empty sequence is not a valid input in this version. It is trivial to modify the code slightly to allow input `[]` to translate to output `[]`.

Back to non-recursive functions and no need for infinite sequences. Consider the problem of implementing a Caesar cipher, one of the simplest encryption schemes ever devised. The ascii alphabet is implemented as a sequence of integers corresponding to a natural sequence of characters. For example, the characters from `' '` to `'~'` are implemented as the sequence `[32,33,34, ... ,124,125,125]`. Letter `'a'` corresponds to number `97`, Letter `'z'` to `122`, Letter `'A'` to `65`, Letter `'Z'` to `90`. Encryption entails rotation of the ascii sequence left through Letter ‘ ‘ by some number X. A second rotation by 95-X will return the ascii sequence to its original state. The following function produces a rotated ascii sequence:

```
Cryptol> let rot s = [' ' .. '~'] <<< s
Cryptol> rot 0
[32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
 55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,
 78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,
101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,
119,120,121,122,123,124,125,126]
Cryptol> rot 95
[ ... identical to above ... ]
Cryptol> rot 84
[116,117,118,119,120,121,122,123,124,125,126,32,33,34,35,36,37,38,39,40,
 41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,
 87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,
 108,109,110,111,112,113,114,115]
```

The following is a function that makes a translation from a given character to a rotated character: c is the character in ‘plaintext’, the key is the amount of rotation for encryption.

```
Cryptol> let shift c key = (rot key) @ (c - ' ')
Crytpol> shift ‘a’ 0
97
Cryptol> shift ‘a’ 84
86
```

Here is a function that converts numbers to ascii printable characters

```
Cryptol> let show msg = [ c:Char | c <- msg ]
```

Finally, the following is a function that shifts all the characters of a plaintext message by key positions.

```
Cryptol> caesar key msg = [ shift c key | c < - msg ]
Cryptol> show (caesar 84 “How Now Brown Cow”)
"=dltCdlt7gdlct8dl"
Cryptol> show (caesar (95-84) "=dltCdlt7gdlct8dl")
"How Now Brown Cow"
```

The above is an example of using the caesar algorithm for encryption and decryption. The number 95 is the size of the translation table.