A Tutorial on Remora
A Rank-polymorphic Programming Language

Olin Shivers

May 3, 2018

1 Introduction

Remora is an example of a *rank-polymorphic* language, a class of programming languages whose fundamental computational model was invented by Kenneth Iverson for the programming language APL and its successor J. In this tutorial, we'll explore the elements of this model, see how they are incarnated in Remora, and then show how to write some sample programs in the language.

Rank-polymorphic languages are known for not requiring explicit iteration or recursion constructs. Instead, the "iteration space" of a program is made real, or "reified," in the shape of its aggregate data structures: when a function that processes an individual element of this space is applied to such a data structure, it is automatically lifted by general polymorphic mechanism to apply across all of the elements of the aggregate.

In particular, we'll look at the three core mechanisms that exist in the language that work together to constitute Remora's control story:

- Frame polymorphism
- Principal-frame cell replication
- Reranking

The interplay of these three mechanisms permits sophisticated Remora (or APL, or J) programmers to write programs that a startlingly succinct.

In this tutorial, we'll avoid any mention of static types. Once the dynamic mechanisms of the language are understood, we'll return, in a later section of this paper, to the issue of how to capture these mechanisms with a static semantics.

2 Three big ideas and some basic terminology

Everything is an array

In rank-polymorphic languages such as Remora, *all values are arrays*. That is, every Remora expression evaluates to an array. An array is a collection of data arranged in a hyper-rectangle of some given dimensionality. Every array comes with its constituent elements, and a shape. Array elements come from a separate universe of *atoms*; typical atoms are numbers, characters, booleans and functions. Permitting arrays of functions means that Remora is a higher-order functional language. For the purposes of this tutorial, we'll limit ourselves to numbers and functions for our atomic array elements.

For example, consider a matrix that has two rows and three columns of integers

$$\begin{bmatrix} 7 & 1 & 2 \\ 2 & 0 & 5 \end{bmatrix}$$

We say that this matrix has *rank* 2—that is, it has two dimensions or axes of indexing—and *shape* $[2,3]$. The shape of an array is a sequence (or, equivalently, list or vector) giving its dimensions.

As another example, suppose we have rainfall data gathered showing the monthly rainfall for twelve months of the year, across fifteen years of data collection, for all fifty states of the USA. We could collect this data as a numeric array RF of rank 3 and shape $[50,15,12]$.

In principle, we could pull out the rainfall for April (month 3) of year 6 for the state of Georgia (state #9) by indexing into the array with the approriate indices: $RF[9,6,3]$. But well-written programs in rank-polymorphic languages do not operate on individual elements on arrays; as we'll see, programs operate on entire arrays. So indexing is, in fact, frowned upon.

When we say "all values are arrays," we even mean scalar values such as 17 or the boolean false. The rank of a scalar array is 0 and its shape is the empty vector $[]$. Note that the rank of an array is also the length of its shape, which is maintained in the case of scalar values.

In Remora, a language with a Lisp-like s-expression syntax, the primitive notation for writing a literal array is the `array` form, that gives the shape of the array followed by its elements listed in row-major order. So our two example arrays, above, along with the scalar 17, could be written in Remora as the constant expressions

```
(array [2 3] 7 1 2 2 0 5) ; Our 2x3 example matrix
(array [50 15 12]         ; Rainfall data
       8 14 10 10 ...)    ;   9,000 elements here
(array [] 17)             ; The scalar value seventeen
```

Note that Remora's basic s-expression syntax uses square brackets as well as parentheses; these are notationally distinct. Note, also, the Lisp comment syntax: all text from a semicolon to the end of a line is ignored.

Array-producing expressions can be assembled into larger arrays with the `frame` form. The first subform of a `frame` expression is a shape or list of dimensions `[`d_1 `...` d_n`]`. This is followed by as many expressions as the product of the d_i; these must all produce arrays of identical shape `[`d'_1 `...` d'_m`]`. Once these expressions have been evaluated, their result arrays are assembled together to produce a final array of rank $n + m$ and shape `[`d_1 `...` d_n d'_1 `...` d'_m`]`.

For example, the following code defines `v` to be a 3-element vector, and `m` to be a two-row, three-column matrix whose two rows are identical to `v`:

```
(define v (array [3] 8 1 7))
(defime m (frame [2] v v))
```

Note the distinctions between the `array` and `frame` forms. The `array` form is for writing down array *constants*, that is, *literal* arrays; its subforms are literal atoms. The `frame` form causes run-time computation to occur: we evaluate the expressions that are its subforms to produce arrays that are then "plugged into" position in the given frame to make a larger, result array.

Now that we've introduced the `array` and `frame` forms, we'll hide them from view at every turn by means of some convenient syntactic sugar:

- First, whenever an atom (that is, an array *element*) literal a appears in a syntactic context where we expect an expression[1], it is taken to be a scalar array—that is, it is treated as shorthand for `(array []` a`)`.

- Second, whenever a sequence of expressions occurs surrounded by square brackets in an expression context, it is treated as a `frame` form for a vector frame. That is, the expression `[`e_1 `...` e_n`]` is treated as shorthand for `(frame [`n`]` e_1 `...` e_n`)`.

- Finally, a frame whose component expressions are all array literals is, itself, collapsed to a single `array` term.

[1]Remember: all expressions produce arrays.

Thus we could write the scalar array 17 as expression `17`, and the vector of the first five primes as expression `[2 3 5 7 11]`; our original example array could be written as

```
[[7 1 2]    ; A 2x3 matrix
 [2 0 5]]
```

This is *exactly* equivalent to the array-literal expression

```
(array [2 3] 7 1 2 2 0 5)
```

Likewise, we could write the constant array giving the truth table for i xor j xor k, using 0 for false and 1 for true, as the rank-3 array

```
;;; A 2x2x2 array
[[[0 1]   ; i=0 plane / j=0 row
  [1 0]]  ; i=0 plane / j=1 row

 [[1 0]   ; i=1 plane / j=0 row
  [0 1]]] ; i=1 plane / j=1 row
```

When using the square-bracket notation, the shape of the array is inferred from the nesting structure of the expression. It's not allowed for two brother elements in a square-bracket array expression to have different shapes; they must match. Thus, the following "ragged" matrix is not a legal expression, as it doesn't have a well-defined shape:

```
[[7 1 2]
 [9 5]    ; Illegal -- row too short!
 [2 0 5]]
```

This requirement falls out of the rules for `frame` expressions. (As we'll see later, there is a mechanism in Remora called a "box," that permits programmers to make ragged arrays, but we'll ignore this for now.)

Functions operate over a frame of cells

In Remora, every function is defined to operate on arguments of a given rank and produce a result of a given rank; these are called the *cells* of the function application. For example, the addition operator `+` operates on two arguments, each of which is a scalar, that is, of rank 0.

```
    (+ 3 4)
7
    (+ 2 8)
10
```

(In this example, and the examples to come, we'll show code and the result expressions it produces, in an "interactive" style, as if we were presenting Remora expressions and definitions to an interpreter: the input Remora expression will be indented, and the value produced will displayed, flush left, on the following line.)

As a second example, we could have a dot-product function `dot-prod` that operates on two arguments of rank 1; or a polynomial evaluation function `poly-eval` that operates on a vector (rank 1) giving the coefficients of a polynomial, and a scalar (rank 0) giving the x value of the polynomial:

```
    (dot-product [2 0 1] [1 2 3])
5
    ;; Evaluate 2 + 0x - 3x^2 at x=1
    (poly-eval [2 0 -3] 1)
-1
```

The argument and result ranks of a function are part of its static definition; when we define our own functions, we must specify them. We do this by tagging each parameter to the function with its rank. So, both `x` and `y` inputs to `diff-square` function below are specified as being of rank 0:

```
    (define (diff-square [x 0] [y 0])
      (- (* x x)
         (* y y)))

    (diff-square 5 3)
16
```

Functions operate on a frame of cells

Any function that is defined to be applied to arrays of rank r is automatically lifted by the language so that it can be applied to arrays of any rank $r' \geq r$.

```
Mention that this is the polymorphism of "rank polymorphism."
```

This is done by viewing an array as a *frame* of *cells*. Consider our 2×3 matrix:

$$\begin{bmatrix} 7 & 1 & 2 \\ 2 & 0 & 5 \end{bmatrix}$$

We can view this array three ways: (1) as a scalar frame containing a single 2×3 matrix cell; (2) as a vector frame containing two cells that are 3-vectors; or (3) as a 2×3 matrix frame containing six scalar cells which are the individual elements of the array. In general, an array with rank r and shape $s = [d_1, \ldots, d_r]$ can be viewed as a frame of cells in $r + 1$ ways, depending on where one splits the shape into the frame prefix and the cell suffix.

The fundamental iteration mechanism of a rank-polymorphic language such as Remora is this: when we apply a function taking an argument of rank r to an actual argument array of rank $r' \geq r$, we divide the input into a frame of cells; the cells have rank r, and the frame has rank $r' - r$. The function is then applied, in parallel, to each argument cell; the results of all these independent applications (which must all have the same shape) are then collected into the frame to produce the final result.

For example, suppose we have a function `vmag` that takes a vector (that is, an array of rank 1) and returns its Euclidean length or magnitude:

```
    (define (vmag [v 1]) ...)
    (vmag [3 4])
5
    (vmag [1 2 2])
3
```

Note that the `v` parameter to `vmag` is defined to take arguments of rank 1. If we apply `vmag` to a matrix, it is applied independently to each row of the matrix. That is, the matrix is viewed by `vmag` as a vector frame of vector cells. All the scalar results of these `vmag` applications are collected into original argument's vector frame:

```
    (vmag [[1 2 2]
           [2 3 6]])
[3 7]
```

Likewise, if we applied `vmag` to a six-dimensional array, it would be treated as a rank-5 frame of vector cells; each such cell would have its length computed, and we would collect these scalar answers into the frame to produce a rank-5 array result.

The principal frame and replication

The frame-distribution mechanism of function application applies just as well when a function has multiple arguments. For example, consider our polynomial-evaluation function, `poly-eval`, that takes a vector of coefficients and a scalar value at which we wish to evaluate the polynomial. Suppose we apply this function to a matrix and a vector

```
    (poly-eval [[2  0 -3]        ; two polynomials
                 [5 -1  1]]
                [-1 2])          ; two x values
  [5 7]
```

The coefficient matrix has shape $[2, 3]$, and the vector of x values has shape $[2]$. Since `poly-eval` operates on vectors for its first argument, it views the matrix (shape $[23]$) as a rank-1 / vector frame (shape $[2]$) of rank-1 / vector cells (shape $[3]$); likewise, it views its vector of x values (shape $[2]$) as a rank-1 / vector frame (shape $[2]$) of rank-0 / scalar cells (shape $[]$). Note that once we've pulled off the cell-shape suffixes from the shapes of each argument, we are left with identical frame shapes: $[2]$. This is called "frame agreement," which means we have a consistent frame across which to distribute the individual function applications. Thus, we evaluate the polynomial $2 + 0x - 3x^2$ at $x = -1$, and $5 - x + x^2$ at $x = 2$, collecting the results into the vector frame and producing final answer `[5 7]`.

However, the frame-based distribution mechanism is more general than simply requiring the frames of all argument arrays to match. The full rule is driven by the notion of an application's "principal frame." In a given function application, the argument frame with the *longest shape* is considered the principal frame; for the function application to be well-formed, the frames of all other arguments must be a *prefix* of the principal frame. When distributing an argument's cells across the cell-wise invocations of the function, if an argument's frame is shorter than (a proper prefix of) the principal frame, then the array is replicated into the missing higher dimensions to provide enough cells for the full frame of function applications.

For example, suppose we add a vector of 2 numbers to a 2×3 matrix:

```
    (+ [10 20] [[8 1 3]
                 [5 0 9]])
  [[18 11 13]
   [25 20 29]]
```

The addition operator adds two scalars to produce a scalar result. Since the two frames are $[2]$ and $[2, 3]$, the principal frame is $[2, 3]$. The first argument's frame gets replicated from shape $[2]$ to shape $[2, 3]$. The way to think of this replication is that when we select a cell from this argument, for frame element $[i, j]$, we simply *drop* any suffix of the index necessary to index the argument's actual frame—in this case, the column index j. This means that we match every column of the right argument's first row with the first element of the left vector: $10 + 8$, $10 + 1$, and $10 + 3$; likewise, we match the items of right argument's second row with the second element of the left vector: $20 + 5$, $20 + 0$, $20 + 9$. Thus, we get one function application for each element of the principal frame, where the results are collected, producing an answer which is a 2×3 frame of scalar cells.

In short, the frame-agreement rule of Remora means that when we add a vector to a matrix, we add the first element of the vector to the first row of the matrix, the second element of the vector to the second rown of the matrix, and so forth. (What if we want to add the first element of the vector to the first *column* of the matrix, and so forth? We'll come to this later.)

Likewise, if we add a matrix M to a three-dimensional array A, then we add element $M[i, j]$ to each element of plane i, row j of A; that is, we add each scalar cell $M[i, j]$ to each scalar cell $A[i, j, k]$.

Given this rule, adding a scalar s to any array simply adds s to each element of the array:

```
    (+ 10 [7 1 4])
[17 11 14]
    (+ [7 1 4] 10)
[17 11 14]
```

If we wish to evaluate a collection of polynomials at a single x, we simply apply the `poly-eval` function to the collection and the x value. Whereas, if we wish to evaluate a single polynomial at a collection of x values, we apply the function to the polynomial and the collection of x values:

```
    ;;; Evaluate two polynomials at the same x.
    (poly-eval [[2  0 -3]         ; 2 + 0x - 3x^2
                [5 -1  1]]        ; 5 -  x +  x^2
               -1)                ; x = -1
[-1 5]
    ;;; Evaluate 2 - 3*x^2 at three values of x:
    (poly-eval [2 0 -3] [1 2 3])
[-1 -10 -25]
```

Note that in none of these cases do we need to write a loop or index into a collection of data; this is managed for us by the rank-polymorphic lifting of the `poly-eval` function across its arguments.

Frame-replication even applies to the function position

Because Remora is a higher-order functional language, we can write a general expression in the function position of a function-application expression; because expressions in Remora evaluate to arrays, this means that the function position of an application can be an *array* of functions. For example, `+` is a variable whose value is a *scalar array* whose single element is the addition function—as described earlier, functions in Remora are atoms, that is, array elements, just as numbers, booleans, and characters are.

Thus, when we evaluate the expression that is the function position of a function application, we get an array (of functions), and *this array participates in the determination of the principal frame for the application, and is subject to frame replication just as the argument arrays are.* The ability to apply an array of different functions to an argument gives Remora a MIMD-style capability to its parallel semantics.

The function position takes scalar cells, which means that in the common case, when we apply a scalar function array (such as the `+` array) to a pair of argument arrays, it is replicated across all the applications.

But we can use non-scalar arrays of functions, as well. Here, we apply a matrix of functions to the single value 9, collecting the results into the matrix principal frame:

```
        (define m [[sqr  sqrt]      ; M is a 2x2 array
                   [add1 sub1]])    ; of functions

        (m 9) ; Apply all the functions to 9.
    [[81 3]
     [10 8]]
```

Some functions take the entire argument as cell

The frame-of-cells story in Remora has a useful corner case: it is possible to specify that a particular parameter to a function takes its entire argument as a single cell. For example, the append function takes two arrays and appends them along their initial dimension. Appending two matrices appends the rows of the second matrix after the rows of the first matrix. So appending a 3×5 array and a 7×5 array produces a 10×5 result. Likewise, appending two three-dimensional arrays appends the planes of the second array after the planes of the first array: appending a $3 \times 2 \times 2$ array and a $4 \times 2 \times 2$ array produces a $7 \times 2 \times 2$ result.

```
        (define m1 [[0  1]
                    [2  3]])

        (define m2 [[10 20]
                    [30 40]])

        ;;; Append two 2x2 matrices; result is 4x2.
        (append m1 m2)
    [[0  1]
     [2  3]
     [10 20]
     [30 40]]
```

The append function is defined so that it consumes both of its arguments, of any rank, as a single cell; thus its frame is a scalar. When we define our own functions, we declare this by tagging a parameter with the special keyword `all` instead of a natural number for its cell rank:

```
        (define (append [a all] [b all]) ...)
```

One way to view such a parameter is that we fix, not the *cell* rank of the parameter, but its *frame* rank: such a parameter has a scalar frame.

What if we want to append along a different axis of an array? For example, instead of appending the two previous matrices one above the other, suppose we wanted to append them side-by-side, producing:

```
    [[0  1 10 20]
     [2  3 30 40]]
```

We'll see how to do this in a following section.

An important operator with scalar frame rank is the higher-order `reduce` function, which maps a binary function over the initial dimension of an array. For example, if we wish to sum the elements of a vector, we reduce it with +:

```
    (reduce + 0 [1 4 9 16]) ; Sum the first four squares.
30
```

If we reduce a matrix with +, we will sum the first row with the second row, the third row, and so forth. So, in effect, we will sum each column:

```
    (reduce + 0 [[1     2   3]
                 [10   20  30]
                 [100 200 300]])
[111 222 333]
```

(What if we want to sum along a different axis of the array? We'll see how to do this in a following section.)

The `reduce` function requires its operator (+, in our example) to be an associative operator of type $\alpha \times \alpha \to \alpha$. Note that the operator gets automatically lifted to operate on the subarrays if it is defined to take cells of smaller rank. Thus, our + operator, which fundamentally operates on scalars, is lifted to operate on vectors when it was used in the example above. Likewise, the initial "zero" element (which is, in fact, zero in our example), is lifted by `reduce` from its scalar `0` value to the required vector `[0 0 0]`.

Remora also provides a `fold` operator that uses a more general folding operator of type $\alpha \times \beta \to \beta$. For example, we can compute the sum of the magnitudes of a collection of vectors with

```
    (fold (λ ([v 1] [sum 0]) (+ sum (vmag v)))
          0
          [[1 2 2]
           [2 3 6]]))
10
```

The advantage of using the less general `reduce` is that it permits the reduction to be performed in a parallel fashion; `fold` is serial. So we would be better off expressing the above calculation as:

```
    (reduce + 0 (vmag [[1 2 2]
                       [2 3 6]]))
```

The individual `vmag` computations can be executed in parallel, as can the additions of the final summation. This is unimportant in this example, where our matrix represents a collection of only two vectors, but would be significant if the matrix had a large number of rows.

There is also a `scan` operator that produces the "prefix sums" of an operator applied across the initial dimension of an array:

```
    (scan + 0 [2 10 5]) ; Produce [2, 2+10, 2+10+5]
[2 12 17]
```

Remora's reduce/fold/scan set of operators provide an important component of its control story. The frame/cell lifting mechanism of the language enforces a separation of computation when we apply a low-rank operator to a high-rank collection of data. For example, when we apply the vector-length operator `vmag` to a three-dimensional array of shape $7 \times 5 \times 3$, the 35 different applications of `vmag` all run independently of one another. This is desireable, as it permits all the different invocations of `vmag` to be executed in parallel. Sometimes, however, we need to perform a computation on a collection of data that somehow combines together the elements of the collection (a computation with what the scientific-computing community would call a "loop-carried dependency" when expressed in programming languages that have explicit loops). In these cases, we use `fold`, `reduce` or one of its brethren—that is their *raison d'être*.

A short aside on parallelism

The use of `fold` and `reduce` characterises a big distinction between programming in a rank-polymorphic language like Remora and programming in a serial array language like FORTRAN. In FORTRAN, we write loops and then hope the compiler can sort out which computations inside the loop are independent of the iteration order and can therefore be parallelised, and which computations have loop-carried dependencies, and so must be left serialised. In Remora, the notationally simple way to operate on a collection of data is simply to apply the function that processes one item to a collection of items: `(f collection)`, and this "default" case is the parallel case. The *actual semantics* is parallel—we are not just emulating a serial semantics with a parallel implementation—so the compiler is licensed to perform all the per-item calculations in parallel; no heroic analyses are needed to divine this fact. On the other hand, the compiler has no difficulty spotting loop-carried dependencies when they do arise, because the programmer *explicitly marks* them by writing down one of the `fold` / `scan` / `reduce` operators.

Rank-polymorphic array languages have historically been popular with their users because the human programmers like the expressiveness and clarity of the notation, without considering performance. But it ought to be true that such languages are well-suited to high-performance implementations on parallel hardware. (And this is our current research agenda.)

Some basic uses of `reduce`

Here is the definition of the `vmag` function we've been using in our examples:

```
(define (vmag [v 1])
  (sqrt (reduce + 0 (* v v))))
```

The function consumes vectors, hence the "1" rank of its `v` parameter. We first use the scalar multiplication operator `*` to do a point-wise multiplication of the vector with itself, producing a vector whose elements are the squares of the input vector's elements. Then we sum these elements with the `reduce` operator, and take the square-root of the result. Note that we did this without ever indexing into a vector or writing a loop.

To write the factorial function, we use the primitive `iota` function, which takes a vector specifying an array shape, and produces an array of that shape, whose elements are the naturals $0, 1, 2, \ldots$ laid out in row-major order:

```
    (iota [5])
[0 1 2 3 4]
    (iota [2 3])
[[0 1 2]
 [3 4 5]]
    (+ 1 (iota [5]))
[1 2 3 4 5]
    (reduce * 1 (+ 1 (iota [5])))       ; 5! = 120
120
    (define (fact [n 0])
      (reduce * 1 (+ 1 (iota [n]))))
    (fact [1 3 5 10])
[1 6 120 3628800]
```

Some simple statistics

We can average the elements of a vector with this function:

```
(define (mean [xs 1])
  (/ (reduce + 0 xs)
     (length xs)))
```

The `length` function is another function (like `append` and `reduce`) that consumes its entire argument as its cell; it returns the size of its argument's initial or leading dimension. Thus, applying `length` to a 3×5 array produces 3.

We can now define variance and covariance using mean:

```
(define (variance [xs 1])
  (mean (sqr (- xs (mean xs)))))

(define (covariance [xs 1] [ys 1])
  (mean (* (- xs (mean xs))
           (- ys (mean ys)))))
```

In `variance`, the subtraction operation uses principal-frame replication to subtract a scalar (the mean of the vector) from every element of the vector. The scalar `sqr` function is lifted to apply it pointwise to all the elements of its vector argument. Similarly, in `covariance`, the scalar multiply operation `*` is lifted to pointwise multiply the two argument vectors, producing a vector result, which is then averaged with `mean`. All of this is accomplished without needing to write an explicit loop or array index; instead of operating on scalar data, the program's operations are applied to entire collections.

One-dimensional convolution

We can convolve a vector of sample data `v` with a weighted window `w` in three lines of code:

```
(define (vector-convolve [v 1] [w 1])
  (reduce + 0
          (* (rotate v (iota (shape w)))
             w)))
```

The key to this function is the lifted `rotate` operation. The `rotate` function takes an array and a scalar amount to rotate the array along its initial, or leading, axis. For example,

```
    (rotate [2 3 5 7 11] 2)
[5 7 11 2 3]

    (rotate [[2 3 4  5 11]
             [1 4 9 16 25]] 1)
[[1 4 9 16 25]
 [2 3 4  5 11]]
```

When `rotate` is given an array and a *vector* of rotation distances, the rank-polymorphic lifting rules of Remora cause it to rotate the array by each of the distances:

```
    (rotate [2 3 5 7] [0 1 2])
[[2 3 5 7]
 [3 5 7 2]
 [5 7 2 3]]
```

The principal frame of the operation is given by the rotation vector—since `rotate` consumes its entire first argument as its cell, the frame for the first argument is always a scalar frame, whose shape `[]` is always a prefix of the second argument's frame. Here, each individual rotation produces a vector result; these three vectors are collected into the principal frame to produce the final matrix result.

In the `vector-convolve` function, we rotate the vector of sample data `v` by the rotation distances `[0 1 ...` $n-1$`]`, where n is the length of the weight vector `w`. This produces the matrix

$$\begin{bmatrix} v_0 & v_1 & v_2 & \ldots & v_{m-1} \\ v_1 & v_2 & v_3 & \ldots & v_0 \\ \ldots & & & & \\ v_{n-1} & v_{n-2} & v_{n-3} & \ldots & v_{n-2} \end{bmatrix}.$$

That is, the top row is the original sample vector; the second row is the sample vector rotated once; the third row is the sample vector rotated twice; and so forth. Note that the height of the matrix is the length of the weight vector, and each column of the matrix is one sample window's worth of data. When we multiply this matrix by the weight vector, the rank-polymorphic lifting rules of Remora multiply the top row of the matrix by the first weight; the second row by the second weight; and so forth. After this, we simply sum each column of the result, collapsing the matrix vertically and producing the final convolution vector.

Again, note that we did not have to write explicit loops, nor did we ever need to use indexing to extract scalar values out of an array, instead operating on entire aggregates in parallel.

We leave it as an exercise for the interested reader to write a version of this function that performs a two- or three-dimensional convolution.

Reranking gives control of the frame/cell mechanism

Remora's fixed frame-replication strategy sometimes doesn't do what we want. For example, if we have an n-element vector `v` and an $n \times n$ matrix `m`, we can add the first element of `v` to the first row of `m`, the second element of `v` to the second row of `m`, and so forth, very simply. The structure of the addition exactly matches the fixed architecture of Remora's principal-frame replication machinery, so we only need to write:

```
(+ v m)
```

However, suppose we want to add the first element of `v` to the first *column* of `m`, and so forth? We manage this by means of η-expanding the + operation, adjusting the frame/cell split with the cell-rank parameter annotations on the wrapper λ term. Consider this example:

```
(define v [10 100])
(define m [[1 2]
           [3 4]])

((λ ([x 1] [y 1]) (+ x y))
 v
 m)
```

The function we are applying to our two arguments explicitly takes vector cells, as marked by the rank-1 annotations on its formal parameters `x` and `y`. The principle frame is given by the two rows of the matrix argument, so the application is distributed over these rows, and the single cell of the `v` argument is replicated across this distribution, giving the following sequence of reductions:

```
[((λ ([x 1] [y 1]) (+ x y)) [10 100] [1 2])
 ((λ ([x 1] [y 1]) (+ x y)) [10 100] [3 4])]
⇒
[(+ [10 100] [1 2])
 (+ [10 100] [3 4])]
⇒
[[(+ 10 1) (+ 100 2)]
 [(+ 10 3) (+ 100 4)]]
⇒
[[11 102]
 [13 104]]
```

. . . which is exactly what we wanted.

Manipulating the way Remora's frame-based distribution works by means of a re-ranking η-expansion is a standard idiom when programming in rank-polymorphic languages. One way to think of this is to bear in mind that function application, in a rank-polymorphic language, is a much more heavyweight mechanism than in the classic λ calculus. In some sense, every function application comes wrapped in its own set of nested loops. When the computation pushes an array argument from the site of the function application off to the body of the function being called, the argument is "cut up" into a collection of cells, and the function application is replicated in parallel across these cells. (All of this implicit loop structure is why we never have to write our own, explicit loops.)

Programming in a rank-polymorphic language such as APL, J and Remora involves developing a reflexive understanding of how principal-frame cell replication causes arguments to be broken up and distributed. Because programmers frequently tune this mechanism with reranking, Remora provides a syntactic shorthand for doing so. We can write the η-expanded addition term from the above example with the reranking `~` notation: `(~(1 1)+ v m)`. In general, writing

~(r_1 ... r_n) *exp*

desugars to

(λ ([v_1 r_1] ... [v_n r_n]) (*exp* v_1 ... v_n))

for fresh parameters v_i.[2] That is, it permits us to specify the cell ranks r_i for the function's arguments.

[2] This is almost true: it ignores the possibility that the evaluation of the function expression *exp* might have a side effect of some kind. In the presence of side effects and a call-by-value

Reranking is often useful in the context of the special functions that consume their entire actual argument as their cell, such as `append`, `rotate` and the `reduce`/`fold`/`scan` family of functions (which effectively constitute a distinct component of Remora's control story).

For example, recall that `append` assembles its arguments together along their leading or initial dimension, so appending two matrices stacks one above the other. A re-ranked append, however, can assemble two matrices side-by-side:

```
    (define m1 [[0 1]
                [2 3]])
    (define m2 [[10 20]
                [30 40]])

    (append m1 m2) ; m1 above m2
[[0  1]
 [2  3]
 [10 20]
 [30 40]]

    (~(1 1)append m1 m2) ; m1 to the left of m2
[[0 1 10 20]
 [2 3 30 40]]
```

The reranked append works by distributing the append across the vector cells of the two arguments, assembling the results into a vector frame. After the reranked application, we have the intermediate result:

```
  [(append [0 1] [10 20])
   (append [2 3] [30 40])]
```

The `rotate` function is similar to `append` in that it rotates its first argument along its initial axis, with the rotation amount given by its scalar second argument. So rotating a matrix rotates its rows vertically; each row is moved as an atomic unit. (Equivalently, we could say that each column is rotated vertically by the same amount as the other columns.)

semantics, we must use the safer desugaring

(let ((f *exp*)) (λ ([v_1 r_1] ...[v_n r_n]) (f v_1 ...v_n))

In practice, the correct, side-effect-safe desugaring almost always reduces to the more informal one we initially gave, as the function term being reranked is typically either a variable or a λ term.

```
    (rotate [[0 1 2]    ; Rotate the rows down
             [3 4 5]    ; by one, and bring the
             [6 7 8]]   ; bottom row up to the top.
            1)
[[6 7 8]
 [0 1 2]
 [3 4 5]]
```

If we wish to rotate the matrix in a horizontal way, we use reranking to apply the rotation to each row of the matrix:

```
    (~(1 0)rotate [[0 1 2]   ; Rotate the columns right
                   [3 4 5]   ; by one, and bring the
                   [6 7 8]]  ; rightmost col around to the left.
                  1)
```

This steps to the intermediate

```
    [(rotate [0 1 2] 1)
     (rotate [3 4 5] 1)
     (rotate [6 7 8] 1)]
```

The results of the three rotations (each a vector) are collected into the vector frame, producing the final result

```
[[2 0 1]
 [5 3 4]
 [8 6 7]]
```

Reranking is especially useful in the context of the `reduce`/`scan` family of operators. The default behavior of `reduce` is to collapse the array argument along its initial dimension:

```
    (reduce + 0 [[0  1   2]      ; Add the first row to
                 [0 10 100]])    ; the second row.
[0 11 102]
```

However, we can sum *across* the matrix by reranking the reduction to apply the reduction operation independently to each row of the input:

```
    (~(0 0 1)reduce + 0 [[0  1   2]
                         [0 10 100]])
[3 110]
```

Matrix multiplication

All of these computational mechanisms come together when we write the standard matrix-multiplication function from linear algebra. We begin by defining a function `v*m` that multiplies an n-element vector v times an $n \times p$ shaped matrix m, producing a p-element vector result. We want the first element of the result to be the dot product of v with the first column of m; the second element of the result to be the dot product of v with the second column of m, and so forth:

```
(define (v*m [v 1] [m 2]) (reduce + 0 (* v m)))
```

We're done: to multiply matrix `a` by matrix `b`, we simply apply `v*m` to the two matrices! The individual rows of `a` will be taken as the vector cells of the first argument, and each one independently multiplied by the entire matrix `b`, which will be taken as a single cell and replicated across the individual multiplies.

We can package this up with a definition that specifies rank-2 (that is, matrix) inputs as follows:

```
(define (m*m [a 2] [b 2]) (v*m a b))
```

...but note that this is just a reranked `v*m`, so could alternatively define the function this way:

```
(define m*m ~(2 2)v*m)
```

If two lines of code seems overly prolix, we can write matrix multiply in a single line of code by pulling all the cell/frame rank manipulation into the reranking notation:

```
(define (m*m [a 2] [b 2])
  (~(0 0 2)reduce + 0 (~(1 2)* a b)))
```

2.1 Polynomial evaluation and efficiency

At the beginning of this tutorial, we imagined a polynomial-evaluation function that takes a vector of coefficients, and an x value at which to evaluate the polynomial. We can define this function in several ways; the various definitions illuminate the considerations that apply to writing efficient, scalable code in Remora.

We begin with a straightforward definition:

```
;;; Simple polynomial evaluation
(define (poly-eval [coeffs 1] [x 0])
  (reduce + 0
          (* coeffs (expt x (iota [(length coeffs)])))))
```

The innermost `(iota [(length coeffs)])` term produces a vector of exponents. Suppose, for example, that the coefficients vector is length 4, with elements [c_0 c_1 c_2 c_3]. Then this inner expression produces `(iota [4])`, which is the vector `[0 1 2 3]`. The exponentiation function `expt` raises `x` to all four of these powers, producing the vector [x^0 x^1 x^2 x^3]. We multiply this vector, point-wise, by the coefficients vector, and sum the result with a `reduce`, producing the final answer.

This definition is simple and clear, but we do a lot of redundant multiplication when we compute each power of x independently of the others. All told, our four-term polynomial example will do $0+0+1+2=3$ multiplies to compute the four powers of x that we need—that is, this code does a quadratic number of multiplies.

For a four-term polynomial, this is not much of a problem, but if our polynomials have a hundred terms, it is a significant waste of computation. We would be better off computing our polynomial with Horner's rule, that is, $c_0+x(c_1+x(c_2+xc_3)$, which only requires a linear number of multiplications. This gives us the following definition, which directly instantiates Horner's rule using a right-to-left fold along the coefficients vector:

```
;;; Efficient on serial processor
(define (poly-eval [coeffs 1] [x 0])
  (foldr (λ ([coeff 0] [acc 0]) (+ coeff (* x acc)))
         0
         coeffs))
```

Unfortunately, using a fold operation essentially is a guarantee that this code cannot be parallised. Again, this is not particularly important if our polynomials are of low degree, but if we were evaluating million-coefficient polynomials, we might want a function that can efficiently make use of multiple processors to execute in parallel. We can achieve this with our final definition:

```
;;; Efficient on serial or parallel processor
(define (poly-eval [coeffs 1] [x 0])
  (reduce + 0
          (* coeffs
             (scan * 1
                   (reshape [(length coeffs)] x))))))))
```

If the length of the `coeffs` vector is n, this definition uses the `reshape` function to make an array of shape `[`n`]` (that is, an n-element vector), all of whose elements are `x`. We then use the `scan` function to multiply together the elements of this vector—`scan` is like `reduce`, except that, instead of reducing the elements down to a single result, it produces all the intermediate prefixes of the reduction. In this case, this produces our vector of exponentiations `[`x^0 x^1 `...` x^{n-1}`]`. Computing this with a scan has two advantages: we only do a linear number of multiplies, and we do so in a fully parallelised manner. From here, the code is straightforward: we multiply the vector of x powers by the coefficients vector and sum the terms. Note that the summation is done with a `reduce`, so this part of the computation is also parallelisable. (And, of course, all the data-parallel bits of the computation expressed with basic rank-polymorphic frame/cell distribution are trivially parallelisable, as well.)