Proposal: Static Semantics and Rank Polymorphism

Justin Slepak

College of Computer and Information Science
Northeastern University
Boston, MA
jrslepak@ccs.neu.edu

1 Introduction

The rank-polymorphic array-oriented programming model offers a powerful, flexible control-flow mechanism based on implicitly transforming every function to operate on arrays of arbitrarily high dimension. Array-oriented programs operate directly on large aggregate data as a whole, instead of one element at a time. Expressing control flow in this manner, where the data itself *is* the iteration space, decouples the code from the shape and rank of the collections over which it operates. A single vector-averaging function can convert one RGB pixel into greyscale, or it can convert a whole image or video: no loops required. This enables a much more concise style of programming than using explicit loops while also making explicit the dependence information that a compiler must otherwise work to reconstruct from a loop nest in order to emit efficient code. This programming model's power has earned both a Turing award for Ken Iverson, who first devised the model in his programming language APL, and a well-established, enthusiastic industrial user base working in statistics and financial analysis. The model is also well-suited for writing concise code in other application areas, such as signal processing, scientific programming, and machine learning.

Unfortunately, this programming model developed in isolation from the programming language research community. Existing instantiations of the model come with too much *ad hoc* extra behavior. They are also too dynamic for good compilation: it is difficult for a compiler to identify statically the iteration space, due to lack of information about both the function and its arguments. **My goal is to isolate and characterize the core mechanics of rank-polymorphic array-oriented programming, by developing a formal dynamic and static semantics, and then to build a core language and associated compiler whose code generation is informed by this distilled understanding.**

Array-oriented programming treats every piece of data exposed to the programmer as an array, which has a sequence of atomic elements and a "shape," the sequence of dimensions. The number of dimensions in the shape is the array's "rank," which is the basis of control flow in array-oriented programming. Matrices are rank-2 arrays, vectors are rank-1 arrays, and even scalars are considered rank-0 arrays. Rank-polymorphic function application splits an argument array's shape into a "frame" of "cells." To apply a square root function to a matrix, we treat it as a matrix frame containing scalar cells, building a matrix containing the square root of each individual cell. Applying a vector-norm function to the same matrix would treat it as a vector frame of vector cells, producing a vector of the norms of the matrix's rows. Since the iteration space is reified as an array, the style of programming is driven by the ability to manipulate the iteration space as ordinary data. The factorial function, for example, can be written as

```
(λ ((n 0)) (reduce * 1 (+ 1 (iota [n]))))
```

This is a function which takes a rank-0 argument called `n`. The `iota` function builds an array with shape `[n]` whose atoms are `0, 1, 2, ..., n-1`. Adding `1` to this `n`-element vector gives the vector `[1 2 3 ... n]` (we will discuss the mechanism that controls how scalar/vector addition is managed later). Finally, we `reduce` by multiplication to get `n!`. Instead of a loop over a range of integer values, the code begins by building an iteration space.

Using the argument arrays as the iteration space enables a form of polymorphic code reuse. For example, consider this linear-interpolation function written for scalars:

```
(λ ((lo 0) (hi 0) (α 0)) (+ (* lo (- 1 α)) (* hi α)))
```

With no modification, the same function can also be used for

– α-blending two RGB pixels (rank-1: a vector of numbers representing color channels)
– dimming or brightening an image (rank-3: a matrix of pixels)
– fade transition between video scenes (rank-4: vectors of images)

Although these are all distinct tasks, the only real difference is in the iteration space. Rank polymorphism allows us to elide the loop-nest boilerplate that would be needed to adapt this function to a particular iteration space.

Rank polymorphism favors a code style that operates directly on large aggregates, which avoids the von Neumann bottleneck of working with individual scalar values. Since the iteration space of any function is concrete data, iteration is bounded. The flexiblity of rank-polymorphic array programming is available even without Turing-completeness. This control-flow mechanism also makes the dependence structure within an aggregate computation clear, without requiring the complex dependence analysis needed for transforming and parallelizing code built around explicit loop nests.

Thesis: The essential flexibility of the rank-polymorphic array programming model can be captured in a core language based on a formal dynamic and static semantics of rank polymorphism which enables efficient compilation without overburdening the programmer with excessive annotations.

This static reasoning and the compilation strategy it enables did not arise alongside the programming model itself because the rank-polymorphic model predates many developments in programming languages that are important for this task:

– **Formal operational semantics** serves as a way to precisely specify the behavior of rank-polymorphic code.
– **Dependent types** offer the flexibility needed to describe the shape of program data, which *must* be determined statically in order for a compiler to understand the shape of the computation.
– With **bidirectional typechecking**, a compiler can propagate its partial knowledge about types in a program in a way programmers can understand, reducing the amount of work put into type annotations even for heavy-weight type systems.
– Advances in **decision procedures**, such as ILP modulo theories, allows us to solve complex constraints so that a type checker, and potentially a type inferencer, can get answers to its questions about data shape.

Each of these technologies plays an essential part in building sufficient static understanding of a program for a compiler to produce efficient code.

2 The programming model

The model of computation I describe here is my own contribution, but it generalizes Iverson's original model, which underlies APL and J. This generalization maintains the conceptual integrity of the original, while integrating it with the higher-order λ-calculus.

In the rank-polymorphic array programming model, every piece of data exposed to the programmer is an array. An array contains a sequence of "atoms," such as numbers or booleans. The array also has a "shape," which is the sequence of its dimensions. The length of this sequence is the array's "rank," which is the main driver of control flow during function application. For example, the matrix $\begin{bmatrix} 9\,8\,7 \\ 6\,5\,4 \end{bmatrix}$ is a rank-2 array, with shape $[2, 3]$, which can be written out in a general s-expression notation as `#A(2 3)(9 8 7 6 5 4)`, or alternatively as `[[9 8 7] [6 5 4]]`. In this nested bracket syntax, a rank-0 array (*i.e.*, a scalar) is written as an unbracketed atom.

2.1 The frame/cell evaluation model

A single array can be viewed at several different ranks, with each view decomposing the array into a *frame* of *cells*. The frame is the outer portion of an array's shape, and the cells are the components of the array within that frame. As suggested by the nested bracket syntax, a (non-scalar) array is a vector of lower-ranked arrays (of identical shape). This generalizes to viewing, for example, an array with shape $[3, 2, 4]$ as any of

– A $3 \times 2 \times 4$ tensor frame of scalar cells
– A 3×2 matrix frame of 4 vector cells
– A 3 vector frame of 2×4 matrix cells
– A scalar frame containing a single $3 \times 2 \times 4$ tensor as its cell

Which of these views to use is a property of the function being applied. A vector-norm function would choose to view its argument as a frame containing vector cells, whereas square root would treat its argument as a frame containing scalar cells. A single function may view different arguments differently: a polynomial-evaluation function might evaluate a polynomial at a scalar value but take the polynomial itself as a vector of coefficients. This means one argument is a frame of scalar cells, and the other is a frame of vector cells.

For both the vector norm and square root cases, the frame shape is chosen to be whatever will make the cells have the desired rank. Less commonly, a function may work on any arbitrary *cell* rank and view an argument at a specified *frame* rank. For example, `reduce` operates along the major axis of an array, and it always uses a scalar frame. Such functions are polymorphic in their cell rank rather than in their frame rank.

Frame agreement and cell replication When a function is applied to several arguments, their frames must all be brought into agreement. Dimensions are added to the ends of the lower-ranked frames, which entails replicating the cells of these arguments, as demonstrated in Figure 1. The `*` and `+` functions operate on scalar cells. The argument frame shapes for the `*` example are `[3]` and `[]` (*i.e.*, a 3-vector frame and a scalar frame). The `[]` frame must be extended to a `[3]` frame by replicating the cell `10`. We end up with a vector of multiplication expressions. For the `+` example, the argument frames are `[3]` and `[3 2]`. The first argument frame can be extended with a new axis of size `2`. Doing so requires making a second copy of each (scalar) cell. Execution proceeds with a matrix of additions. The `dot` product operator works on vector cells instead. Thus its argument frame shapes in Figure 1 are `[]` and `[2]`. The matrix argument is viewed as a vector frame of vector cells, so the semantics calls for the vector argument to be replicated[1] in its entirety.

$$\texttt{(* [1 2 3] 10)} \quad\mapsto\quad \texttt{(* [1 2 3] [10 }\underline{\texttt{10}}\ \underline{\texttt{10}}\texttt{])} \quad\mapsto\quad \texttt{[(* 1 10) (* 2 10) (* 3 10)]}$$

$$\texttt{+ [10 20 30]} \begin{bmatrix}\texttt{[1 2]}\\ \texttt{[3 4]}\\ \texttt{[5 6]}\end{bmatrix} \quad\mapsto\quad \texttt{+} \begin{bmatrix}\texttt{[10 }\underline{\texttt{10}}\texttt{]}\\ \texttt{[20 }\underline{\texttt{20}}\texttt{]}\\ \texttt{[30 }\underline{\texttt{30}}\texttt{]}\end{bmatrix} \begin{bmatrix}\texttt{[1 2]}\\ \texttt{[3 4]}\\ \texttt{[5 6]}\end{bmatrix} \quad\mapsto\quad \begin{bmatrix}\texttt{[(+ 10 1) (+ 10 2)]}\\ \texttt{[(+ 20 3) (+ 20 4)]}\\ \texttt{[(+ 30 5) (+ 30 6)]}\end{bmatrix}$$

$$\texttt{dot [10 20 30]} \begin{bmatrix}\texttt{[1 2 3]}\\ \texttt{[4 5 6]}\end{bmatrix} \mapsto \texttt{dot} \begin{bmatrix}\texttt{[10 20 30]}\\ \underline{\texttt{[10 20 30]}}\end{bmatrix} \begin{bmatrix}\texttt{[1 2 3]}\\ \texttt{[4 5 6]}\end{bmatrix} \mapsto \begin{bmatrix}\texttt{(dot [10 20 30] [1 2 3])}\\ \texttt{(dot [10 20 30] [4 5 6])}\end{bmatrix}$$

Fig. 1. Automatic expansion of array arguments, followed by the rank-lifted distribution of the operator across the cells of the frames.

Mechanics of rank Recall our example linear-interpolation function:

```
(define lerp
  (λ ((lo 0) (hi 0) (α 0))
    (+ (* lo (- 1 α)) (* hi α))))
```

[1] A compiler may still avoid physically copying the vector.

$$\left(\texttt{curry-add}\begin{bmatrix}1\\2\end{bmatrix}\right)\begin{bmatrix}20\\30\end{bmatrix} \mapsto \begin{bmatrix}\texttt{(curry-add 1)}\\\texttt{(curry-add 2)}\end{bmatrix}\begin{bmatrix}20\\30\end{bmatrix} \mapsto \begin{bmatrix}\texttt{((curry-add 1) 20)}\\\texttt{((curry-add 2) 30)}\end{bmatrix}$$

$$\begin{bmatrix}\texttt{sum}\\\texttt{length}\end{bmatrix}_2 \begin{bmatrix}8\\9\\6\end{bmatrix}_3 \mapsto \begin{bmatrix}\texttt{sum}\\\texttt{length}\end{bmatrix}_2 \begin{bmatrix}8\ 9\ 6\\8\ 9\ 6\end{bmatrix}_{\underline{2},3} \mapsto \begin{bmatrix}(\texttt{sum}\ [8\ 9\ 6]_3)\\(\texttt{length}\ [8\ 9\ 6]_3)\end{bmatrix}_2$$

Fig. 2. Lifting the implicit `apply`

When used to α-blend two pixels, such as

```
(lerp [3 8 190] [120 150 0] 0.2)
```

each pixel argument is a vector of three color channel values, so their frame shape is `[3]`. The α argument is a scalar with frame shape `[]`, so its one cell (itself a scalar) must be replicated three times to produce a vector with shape `[3]`. Then `lerp` is applied pointwise to the three vectors. For dimming an image, we represent the image as a rank-3 array, whose axes represent which row of pixels, which pixel within the row (*i.e.*, which column), and which color channel. The α and brightness adjustment scalars must have three axes added to their respective frames, so that `lerp` can be applied to three rank-3 arrays. Scene fading uses a rank-4 array to represent a video: time, row, column, and color channel. The α argument is a vector gradually transitioning from 0 to 1, meaning that one scene should be completely ignored at time t_0 while the other is completely ignored at time t_n. Part-way between, we want an intermediate blend of the two video segments, weighted based on how much time has elapsed.

```
(lerp scene1 scene2 [0.0 0.1 0.2 ... 1.0])
```

At execution, the α vector, with just the "time" axis, has the "row," "column," and "channel" axes added to its frame shape, expanding it into a rank-4 array, matching the two videos.

First-class functions With functions as first-class values, the result of some function might itself be a function. When that function-producing function is applied with a non-scalar frame, the result is an array of functions. So the rule for lifting function application must handle function arrays as well as argument arrays. The way to apply an array of functions is to consider function application itself a liftable operation that expects a scalar cell in function position. The expected ranks for argument positions are the same as those of the functions in the array being applied. This requires that all functions in the array agree exactly as to argument rank. A vector-summation function and a vector-length function can be applied together, but a matrix-inversion function cannot be included with them.

Controlled irregularity There is an escape hatch from the requirement for regular, rectangular array data. A "box" wraps an arbitrary array in a scalar value. The box can be opened to produce a regular array, but packing several boxes together effectively builds an irregular array. Hiding the shape of each box's contents while exposing the shape of the collection of boxes means that irregular arrays can reveal their partial regularity for rank-polymorphic lifting. So a function which consumes a box can be lifted over an irregular array, with each box serving as a cell. This lifting does not reach through to the box's contents: passing a matrix of boxed vectors to a scalar-consuming function will apply the function to each boxed vector.

2.2 Programming with rank polymorphism

A common theme in rank-polymorphic programming is constructing an iteration space from a function argument. Recall the earlier factorial example, which builds a vector of numbers to multiply together:

```
(define (fact (n 0))
  (reduce * 1 (+ 1 (iota [n]))))
```

Likewise, we can define a 1-dimensional `convolve` function, which computes a weighted moving average:

```
(define (convolve (filter 1)   ; vector of length f
                  (signal 1))  ; vector of length s
  (reduce
   + 0
   (* filter
      (rotate signal
              (iota [(length filter)])))))
```

The overall strategy is to build the 2-dimensional iteration space and then `reduce` it to a single vector. That iteration space is a vector whose elements are themselves time-shifted versions of the original `signal` vector. The shift amounts are generated by `iota`: $0, \ldots, f-1$. Then `rotate` uses each one to produce a separate version of `signal`, and they are assembled into a matrix with shape `[f s]`. When we multiply this matrix by the `filter`, the frame shapes in this multiplication are `[f]` for `filter` and `[f s]` for the matrix. So `filter` must expand by adding an `s` onto its frame, replicating each coefficient (*i.e.*, each scalar cell) `s` times. The expanded version of `filter` has each row filled with a single coefficient, so the lifted `*` effectively treats `filter` as a column vector and multiplies it by each column in the matrix made of shifted `signal`s. Each time-shifted `signal` has now been weight-adjusted according to the `filter`. Finally, `reduce` adds all of these adjusted, shifted `signal`s to each other, producing the weighted moving average.

While the code for vector-matrix addition could choose whether to use a row vector or a column vector by transposing the matrix before and after performing addition, this is not the preferred strategy for axis alignment. The general `transpose` operator takes a vector specifying a permutation of axes. This dependence on input values, as opposed to the input shape, makes the shape-level behavior of a program less apparent.

Reranking to perturb the iteration space In the vector-matrix addition example in Figure 1, the vector is effectively treated as a column in that replication adds new axes on the right side of its shape. That is, in the vector/matrix sum `(+ v m)`, the rules of frame replication end up adding the i^{th} element of `v` to the i^{th} row of `m`. Suppose we instead wanted to add the i^{th} element of `v` to the i^{th} *column* of `m`. An η-expansion wrapper around `+` that changes the argument rank from 0 to 1 would treat the vector argument as a scalar frame of vector cells. When lifting this version of `+`, new axes would be added on the left, treating it as a row vector. The appropriate wrapper, demonstrated in Figure 3, is a cell-rank shifting η-expansion of `+`:

```
(λ ((n 1) (m 1)) (+ n m))
```

This is a function which operates on vector cells for both of its arguments, but it passes those two vectors to `+` to produce a vector result cell.

This technique of shifting cell rank is called "reranking" the operator, and it is such a common occurrence in rank-polymorphic programming that such languages typically offer compact notation to support it.

Figure 3 also demonstrates how to `append` or `reduce` along a chosen axis. The `append` function stitches two arrays together along their major axis, For example, a pair of 2×2 matrices can be appended to form a 4×2 matrix. To join them along their minor axis, we can use `(λ ((n 1) (m 1)) (append n m))`. Applying this vector-appending function to our two matrices will see them both as vector frames containing vector cells. Each corresponding pair of 2-vectors is joined, producing three 4-vectors, assembled inside the 2-vector frame, *i.e.*, a 2×4 matrix. Similarly, `reduce` accumulates a result value by performing an operation on the $(n-1)$-cells of a rank-n array. When reranked to expect a vector argument, it will reduce each individual row of a matrix and reassemble a vector of the results.

The intuitive idea of aligning array axes can be carried quite far. Recall that a scalar function effectively treated vectors as rows, while reranking that function to 1 treated them as columns. Suppose we reranked on one argument but not the other: `(define o* (λ ((n 0) (m 1)) (* n m)))`. When we apply this to a

$$\left(\texttt{+}\begin{bmatrix}1&2\\3&4\end{bmatrix}\texttt{[10 20]}\right)\mapsto\left(\texttt{+}\begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}10&10\\20&20\end{bmatrix}\right)\mapsto\begin{bmatrix}11&12\\23&24\end{bmatrix}$$

$$\left(\begin{array}{l}\texttt{(}\lambda\texttt{ ((xs 1) (ys 1)) (+ xs ys))}\\ \begin{bmatrix}1&2\\3&4\end{bmatrix}\texttt{[10 20]}\end{array}\right)\mapsto\left(\texttt{+}\begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}10&20\\10&20\end{bmatrix}\right)\mapsto\begin{bmatrix}11&22\\13&24\end{bmatrix}$$

$$\left(\texttt{append}\begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}5&6\\7&8\end{bmatrix}\right)\mapsto\begin{bmatrix}1&2\\3&4\\5&6\\7&8\end{bmatrix}$$

$$\left(\begin{array}{l}\texttt{(}\lambda\texttt{ ((xs 1) (ys 1)) (append xs ys))}\\ \begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}5&6\\7&8\end{bmatrix}\end{array}\right)\mapsto\begin{bmatrix}\texttt{(append [1 2] [5 6])}\\\texttt{(append [3 4] [7 8])}\end{bmatrix}\mapsto\begin{bmatrix}1&2&5&6\\3&4&7&8\end{bmatrix}$$

$$\left(\texttt{reduce + 0}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}\right)\mapsto\texttt{[12 15 18]}$$

$$\left(\begin{array}{l}\texttt{(}\lambda\texttt{ ((xs 1)) (reduce + 0 xs))}\\ \begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}\end{array}\right)\mapsto\begin{bmatrix}\texttt{(reduce + 0 [1 2 3])}\\\texttt{(reduce + 0 [4 5 6])}\\\texttt{(reduce + 0 [7 8 9])}\end{bmatrix}\mapsto\texttt{[6 15 24]}$$

Fig. 3. Reranking to operate along a different axis

$$\begin{array}{l}
\texttt{((}\lambda\texttt{ ((n 0) (m 1)) (* n m)) [1 10 100] [1 2 3 4])}\\
\mapsto \left(\texttt{(}\lambda\texttt{ ((n 0) (m 1)) (* n m))}\quad \texttt{[1 10 100]}\ \begin{bmatrix}\texttt{1 2 3 4}\\ \texttt{1 2 3 4}\\ \texttt{1 2 3 4}\end{bmatrix}\right)\\
\mapsto \begin{bmatrix}\texttt{(*~~~1 [1 2 3 4])}\\ \texttt{(*~~10 [1 2 3 4])}\\ \texttt{(* 100 [1 2 3 4])}\end{bmatrix}\\
\mapsto \begin{bmatrix}\texttt{(* [~~1~~~1~~~1~~~1] [1 2 3 4])}\\ \texttt{(* [~10~~10~~10~~10] [1 2 3 4])}\\ \texttt{(* [100 100 100 100] [1 2 3 4])}\end{bmatrix}\\
\mapsto \begin{bmatrix}\texttt{[(*~~~1 1) (*~~~1 2) (*~~~1 3) (*~~~1 4)]}\\ \texttt{[(*~~10 1) (*~~10 2) (*~~10 3) (*~~10 4)]}\\ \texttt{[(* 100 1) (* 100 2) (* 100 3) (* 100 4)]}\end{bmatrix}\\
\mapsto \begin{bmatrix}\texttt{[~~1~~~2~~~3~~~4]}\\ \texttt{[~10~~20~~30~~40]}\\ \texttt{[100 200 300 400]}\end{bmatrix}\\
\mapsto \begin{bmatrix}1 & 2 & 3 & 4\\ 10 & 20 & 30 & 40\\ 100 & 200 & 300 & 400\end{bmatrix}
\end{array}$$

Fig. 4. Computing the outer product by reranking the `*` function

3-vector and a 4-vector, as in Figure 4, the first is treated as a vector frame of scalar cells, while the second is a scalar frame containing a vector cell. So the application frame shape is `[3]`, and we make 3 copies of the 4-vector, *i.e.*, an array with shape `[3 4]`. Inside the function body, we apply `*` itself to these arrays with shapes `[3]` and `[3 4]`. This application frame shape is `[3 4]`, with the 3-vector having its scalar cells replicated so that both arrays are 3×4 matrices. The final result is the outer product of the two vectors.

The vector dot product, used in Figure 1, can be implemented as:

```
(define (dot (xs 1)   ; vector of length n
             (ys 1))  ; vector of length n
  (reduce + 0 (* xs ys)))
```

Written and used as above, the function is far less flexible than it could be. Using cell rank of 1 forces it to operate on its arguments' minor axes. Functions like `reduce`, `append`, and `rotate` can accept cells of any rank (*i.e.*, the frame shape is always `[]`) and operate along their arguments' major axes. We can do the same with dot product:

```
(define (dot (xs all)
             (ys all))
  (reduce + 0 (* xs ys)))
```

Passing the two arguments directly to the scalar-consuming `*` operator means that this version of `dot` is only safe to use when one argument's shape is a prefix of the other's shape. Within the function body, the lower-ranked argument grows to match the higher-ranked one. Now that we have a major-axis operation, applying it to two matrices will effectively compute dot products of corresponding columns. If we want dot products of rows instead, we can rerank it. Preferring to operate along the major axis of high-rank arguments is a good general design principle: As with the `append` example above, reranking a major-axis operation lets us choose any axis we want without resorting to use of `transpose`.

One interesting use of reranked `dot` is to take argument cells of ranks 1 and 2 for its respective arguments:

```
(define (m* (x 1)
            (y 2))
  (dot x y))
```

When applied to two matrices, this breaks the first argument into rows. A row's major-axis `dot` product with the full matrix `y` will multiply each scalar in `x` (a row from the first matrix) by each row in `y`. As in the vector-matrix addition, this treats `x` as a column vector. Summing these individual products produces an element of the matrix product of the two original arguments. So the final rank-2 result is that matrix product.

Matrix multiplication itself generalizes usefully to algebras other than the real (or complex) numbers. First-class functions let us parameterize the dot product over the particular addition and multiplication operators we wish to use:

```
(define (dot (add 0)
             (mult 0)
             (xs all)
             (ys all))
  (reduce add 0 (mult xs ys)))
```

A matrix product built from this version of `dot` can be used for a wide range of applications, including transitive closure or path search in a graph and transfer functions for dataflow analysis [9].

3 Why a new language

Given the practical industrial value of APL and J as a major motivation for this work, why not dissect, analyze, and compile APL or J themselves? While these languages have historically resisted attempts to compile them, much of the difficulty has little to do with rank polymorphism itself. For example, J's function-exponentiation operator, `^:`, repeatedly applies a function a specified number of times to some input (like a Church numeral), but it also allows raising a function to a *negative* power. This relies on having a designated "inverse" for the function in question, which is not always possible for arbitrary functions. It also privileges built-in primitive operations. Many primitive operators also come with special-case behavior which must add more special cases to any description of how such operators behave at the shape level. APL and J both separate first-order functions from second-order functions at the *syntactic* level, with different parsing rules for writing and applying them. Worse, whether a piece of program text parses as a first-order function, second-order function, or ordinary data depends on the run-time values held by variables in the text. Since parsing a whole program statically is impossible due to this phase crossing, APL compilers have been forced to work in line-at-a-time mode. While satisfactory for strictly interactive use, looking only at small snippets of code leaves a lot of optimization opportunities on the table.

Furthermore, APL and J place unsatisfying restrictions on lifting functions to higher-ranked arguments. Beyond the syntactic separation between first-order and second-order functions (and the lack of general higher-order functions), rank-polymorphic lifting is only available for first-order functions in APL and J. Functions are required to be unary or binary. Programmers use two techniques to get around this restriction, but both give up significant power. A second-order function can consume one or two arguments and then produce a first-order function to consume the remaining arguments. Since there is no aggregate lifting for second-order functions, a library designer is forced to designate some function arguments as only usable with a scalar frame. Alternatively, programmers can package multiple pieces of data together. While there is no distinct "tuple" construct, a heterogeneous vector is permitted. This technique forces several pieces of input to have the same frame shape, and choosing which ones to group together presents a similar problem for a library designer. No matter which technique is used and how many separate inputs are shoehorned into an application, a function can only be used with two distinct frames.

Leaving aside these issues, efforts to compile array languages have also run into trouble determining program control flow. The iteration structure in an application of a rank-polymorphic function depends on the ranks of the argument arrays. Viewing rank and shape as values that can be computed from run-time data, such as with APL's ρ function, casts this as an expensive data-flow problem (even more expensive if first-class functions are supported). Beyond dynamically-computed array rank, the argument rank expected by a function can be chosen by arbitrary run-time computation. In practice however, programmers rarely rely on the ability to have completely arbitrary shapes flow into an argument position. Instead, they have an expected range of possible shapes, and the code would likely raise a run-time error if that expectation is violated. This mode of thinking follows a type-like discipline, which calls for an actual type system not only to enforce safety by ensuring arguments have compatible shapes but also statically identify the iteration space for a compiler's benefit. Characterizing this control flow at compile time is the missing piece holding back effective compilation for rank polymorphism.

4 Remora

Design goals Remora is a new core language, which avoids the historic baggage accreted by APL and J. It uses a simple s-expression syntax where functions state their parameters' expected ranks as literals rather than arbitrary expressions. This gives up some of the flexibility of computing expected argument rank, but that capability appears to be generally unused in practice[2]. Those particular functions whose cell rank does vary (such as `reduce`, `scan`, and `append`) have always-0 frame rank instead. Remora also treats functions as first-class data, rather than syntactically distinguishing functions which consume or produce functions from functions which don't. Second-order functions are essential to APL and J, but a restriction against functions of even higher order is not. Similarly, using s-expression syntax instead of strictly infix notation permits functions with any arity.

Remora comes with a formal operational semantics so that we can justify claims that some term should result in some particular behavior. This is effectively a specification for rank polymorphism's implicit aggregate lifting, taking the place of a potentially ambiguous prose description or a specific implementation. The semantics also highlights a "hole" in the lifting rules: When the frame contains 0 as one of its dimensions, the cell portion of the result array's shape cannot be determined from values computed at run time.

The precise description of how rank-polymorphic code behaves at run time provides the foundation needed for a static semantics of rank polymorphism. Dependent types in the style of Dependent ML [26] are used to characterize the shapes of array data as lists of natural numbers. Function types describe both the required input cell shape and the output cell shape, but the frame shape for a function application is determined on a per-application basis, by splitting out the frame portions of arguments' shapes. Thus even in the typed setting, all functions are still implicitly polymorphic over argument rank (as long as a suitable cell shape is available). Functions which can vary in *cell* rank, such as `reduce`, are typed as dependent products. They are universally quantified over a shape, and that shape variable is used to build an actual cell shape at a particular call site. Functions can also quantify over individual dimensions rather than entire shapes, so that a vector-norm function can specify that cells are vectors without fixing the vectors' length.

Some functions' output shape depends on the values in the input array, not just the shape in which they are arranged. For example, `(iota [3])` produces a 3-vector, and `(iota [4])` a 4-vector. The output of `filter` is a vector of items whose length depends on how many input items satisfied a given predicate. Boxes, the escape hatch from regularity, are the proper tool for taming this behavior. Lifting `iota` to consume a higher-rank argument, such as `[[5] [4]]`, may produce a vector whose elements are themselves arrays of differing shape, *i.e.*, an irregular array. In Remora, such functions produce dependent sums, where some part of the shape is existentially quantified. In general, a dependent sum can be seen as a sum type with one possible variant for every inhabitant of the quantified-over sort. Some element of that sort is carried along with the sum as a tag to identify the particular variant. A dependent sum can also be thought of as a generalization of a pair, where the type of the second element depends on the value of the first element, or an existential type that quantifies over an index instead of a type.

The result type of `iota` is

$$\Sigma\,[d : \texttt{Shape}]\, \texttt{A}_d\texttt{Num}$$

[2] The "Essays" on the J wiki, which demonstrate typical J code for , occasionally construct functions with explicitly-specified rank, but the rank annotations used are numeric literals

This means we have an array of numbers, with shape d (that is $\mathtt{A}_d\mathtt{Num}$), but d itself is effectively "sealed" inside the dependent sum (written as $\Sigma\,[d : \mathtt{Shape}]\ldots$). Dependent sums offer more flexibility than boxes, which always hide the entire shape of their contents. For example, the result of using `filter` on a 4×7 matrix of numbers is

$$\Sigma\,[l : \mathtt{Nat}]\,\mathtt{A}_{[l\ 7]}\mathtt{Num}$$

i.e., a $l \times 7$ matrix for some unknown l. We are removing and retaining entire rows from the matrix, so we know how long each row is but not how many we end up with; the index variable l represents the unknown number of rows. On the other hand, reranking `filter` so that we independently filter each row of the matrix produces

$$\mathtt{A}_{[4]}\,\Sigma\,[l : \mathtt{Nat}]\,\mathtt{A}_{[l]}\mathtt{Num}$$

a 4-vector frame of boxed vectors, each with unknown length. Using dependent sums allows data to be irregular along specific axes. The previous result type is different from

$$\Sigma\,[l : \mathtt{Nat}]\,\mathtt{A}_{[4\ l]}\mathtt{Num}$$

While the vector of dependent sums can be ragged in its minor axes, this sum contains a matrix (of unspecified width), guaranteeing regularity. By contrast, a box-based data model could not guarantee that all vectors in the matrix have the same length.

This static semantics also provides a way to resolve the uncertainty about result shape when a function is applied over an empty frame. Even though there are no result cells to inspect for shape, the function's type specifies the shape of the data it will return.

Syntax The abstract syntax for Remora is given in Figure 5—this notation differs from the code samples used in earlier sections. Here, $t\ \ldots$ denotes a (possily empty) sequence, t_1 through t_k, and $f(t)\ \ldots$ denotes the sequence $f(t_1)$ through $f(t_k)$. Then $t\ t'\ \ldots$ is a guaranteed-nonempty sequence.

The syntax for array literals α includes a type annotation to specify the shape as well as the type of atomic elements in an empty array: $[l\ \ldots]^\tau$. In the case of a nonempty array, a shape annotation alone suffices: $[l\ l'\ \ldots]^\iota$. Every nested representation of the same array will eventually reduce to the same flat representation, *e.g.*, the expression $\Big[[1\ 2]^{\mathtt{A}_{[2]}\mathtt{Num}}\ [3\ 4]^{\mathtt{A}_{[2]}\mathtt{Num}}\Big]^{\mathtt{A}_{[2]}\mathtt{A}_{[2]}\mathtt{Num}}$ will evaluate to $[1\ 2\ 3\ 4]^{\mathtt{A}_{[2\ 2]}\mathtt{Num}}$.

$$\begin{array}{rcll}
e & ::= & \alpha \mid x \mid (e\ e'\ \ldots) \mid (\mathtt{T}\lambda\,[x\ \ldots]\,e) \mid (\mathtt{T\text{-}APP}\ e\ \tau\ \ldots) \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,e) \mid (\mathtt{I\text{-}APP}\ e\ \iota\ \ldots) & \textit{(exressions)} \\
 & & \mid (\mathtt{PACK}\ \ \iota\ \ldots\ \ e)^\tau \mid (\mathtt{UNPACK}\ (\langle x\ \ldots\ |y\rangle = e)\ \ e') & \\
\alpha & ::= & [l\ \ldots]^\tau \mid [l\ l'\ \ldots]^\iota & \textit{(arrays)} \\
l & ::= & b \mid f \mid e \mid (\mathtt{T}\lambda\,[x\ \ldots]\,l) \mid (\mathtt{T\text{-}APP}\ l\ \tau\ \ldots) \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,l) \mid (\mathtt{I\text{-}APP}\ l\ \iota\ \ldots) & \textit{(array elements)} \\
f & ::= & \pi \mid (\lambda\ [(x\ \tau)\ \ldots]\ \ e) & \textit{(functions)} \\
\tau, \sigma & ::= & B \mid x \mid \mathtt{A}_\iota\tau \mid \tau\ \ldots \to \sigma \mid \forall\,[x\ \ldots]\,\tau \mid \Pi\,[(x\ \gamma)\ \ldots]\,\tau \mid \Sigma\,[(x\ \gamma)\ \ldots]\,\tau & \textit{(types)} \\
\iota, \kappa & ::= & n \mid x \mid [\ \iota\ \ldots] \mid (+\ \iota\ \kappa) & \textit{(indices)} \\
\gamma & ::= & \mathtt{Nat} \mid \mathtt{Shape} & \textit{(index sorts)} \\
z & \in & \mathbb{Z} & \textit{(numbers)} \\
n, m & \in & \mathbb{N} & \\
v & ::= & [b\ \ldots]^\tau \mid [f\ \ldots]^\tau \mid b \mid f \mid (\mathtt{T}\lambda\,[x\ \ldots]\,l) \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,l) \mid (\mathtt{PACK}\ \ \iota\ \ldots\ \ v) & \textit{(value forms)} \\
 & & \mid [(\mathtt{PACK}\ \ \iota\ \ldots\ \ v)\ \ldots]^{\mathtt{A}_{[\ m\ n\ \ldots]}\tau} & \\
E & ::= & \square \mid (v\ \ldots\ \ E\ e\ \ldots) \mid [v\ \ldots\ \ E\ l\ \ldots]^\tau \mid (\mathtt{T\text{-}APP}\ E\ \tau\ \ldots) \mid (\mathtt{I\text{-}APP}\ E\ \iota\ \ldots) & \textit{(evaluation contexts)} \\
 & & \mid (\mathtt{PACK}\ \ \iota\ \ldots\ \ E)^\tau \mid (\mathtt{UNPACK}\ (\langle x\ \ldots\ |y\rangle = E)\ \ e) & \\
\Gamma & ::= & \cdot \mid \Gamma,\ (x : \tau) & \textit{(type environments)} \\
\Delta & ::= & \cdot \mid \Delta,\ x & \textit{(kind environments)} \\
\Theta & ::= & \cdot \mid \Theta,\ (x :: \gamma) & \textit{(sort environments)}
\end{array}$$

Fig. 5. Abstract syntax for explicitly typed Remora. Variables are denoted with x, base values with b, and primitive operators with π.

While programs deal only with arrays, an array is ultimately made up of atomic (*i.e.*, non-array) elements. So bare functions and base values (*i.e.*, atomic elements) are in a broader syntactic class than expressions. This class of elements does include all expressions, so that a frame of cells can be written as an array containing its sub-arrays. Since function application requires expressions in all argument positions, a variable can only be bound to an array. Type and index abstractions and applications can be used as array elements, allowing different elements in the same array to be instantiated with different type and index arguments to produce functions of the same type. If the body of a type or index abstraction or application is an expression (*i.e.*, an array-producing term), then the application or abstraction element is also an expression.

Functions specify their arguments' cell shapes, as in

$$(\lambda \left[(\texttt{rgb}\ \texttt{A}_{[3]}\texttt{Num})\ (\texttt{gamma}\ \texttt{A}_{[]}\texttt{Num})\right]\ (\texttt{expt}\ \texttt{rgb}\ \texttt{gamma}))$$

for a γ-correction on an RGB pixel.

Value forms are fully-reduced elements (whether array or atomic). They may contain base values or functions but not application forms or variables. A dependent sum is a value as long as it contains a syntactic value, and a non-scalar array of dependent-sum values is also a value (a scalar array containing a dependent sum will reduce to the dependent sum itself).

Dynamic Semantics The reduction relation is given in Figure 6. It assumes every expression has been annotated with its type. Most of these annotations can be generated during type checking, using only programmer-supplied argument cell types on functions and type annotations on arrays (or shape annotations for non-empty arrays). This run-time type information is required by function application in order to determine the result shape if the application frame is empty, so type annotations are maintained during reduction.

The reduction rules capture the implicit parallelizability of rank-polymorphic code. The *lift* rule's replication of the arrays' cells corresponds to scattering data to several processing units (threads, cluster nodes, *etc.*). This is an explicit representation of the "free"[3] communication pattern: while we generate argument cells for each independent task, an actual implementation of Remora may choose to group several of these independent tasks together, resulting in less actual communication at run time. This step only transforms the arrays in an application form—we still have an application form afterwards. Next, *map* kicks off independent computation of each result cell. In the formalism, this produces a frame of expressions, where each one computes one cell of the final result. In the case of an empty application frame, the result of *map* is an empty array, still tagged with the appropriate type (thus the appropriate shape as well). Each one can then proceed with its own β or δ reduction. Finally, a *collapse* reduction shows the ending gather step, merging the pieces of the result into a single array.

The β rule requires an application form with scalar frame in function and argument positions. So there must be a single function, and the argument arrays must match the function's noted argument cell types. Setting up this situation (*i.e.*, performing the replication or communication) is the responsibility of the *lift* and *map* rules.

$T\beta$, $I\beta$, and *proj* are standard reduction rules for respectively consuming a type abstraction, consuming an index abstraction, and consuming a dependent sum. Maintaining type annotations in $T\beta$ and $I\beta$ requires performing the same substitution in both the term itself and in the type annotation.

Static Semantics Remora's type, kind, and sort rules are given in Figures 7 and 8. Types classify both elements and array expressions. T-App must identify the frame associated with an application form, which requires identifying the frames associated with the individual terms in the application form. Recall that for a *map* reduction, the frames of every term in the application must be the same, and for a *lift* reduction, there must be one frame which is prefixed by every other frame. Once every term's frame has been determined, the next step is to find the largest frame, with the order given by $x \sqsubseteq y$ iff x is a prefix of y. This will be the

[3] Free as in algebra

Applying a term abstraction:

$$\left([(\lambda\ (x\ \ldots)\ e^{\sigma})^{\tau\ \ldots \to \sigma}]^{\mathtt{A}\tau\ \ldots \to \sigma}\ \ v^{\tau}\ \ldots\right)^{\sigma}\ \mapsto_{\beta}\ (e\,[(x \leftarrow v)\ \ldots])^{\sigma}$$

Applying primitive operator:

$$(\pi^{\tau\ \ldots \to \sigma}\ v^{\tau}\ \ldots)^{\sigma}\ \mapsto_{\delta}\ \delta(\pi, v\ \ldots)$$

Pointwise application:

$$\left([f\ \ldots]^{\mathtt{A}[\ n_f\ \ldots\ ]^{\mathtt{A}[\ n_a\ \ldots\ ]\tau\ \ldots \to \tau'}}\ \ v^{\mathtt{A}[\ n_f\ \ldots\ n_a\ \ldots\ ]^{\tau}}\ \ldots\right)^{\mathtt{A}[\ n_f\ \ldots\ n_c\ \ldots\ ]^{\tau'}}$$

$$\mapsto_{map}\ \left[\left([f]^{\mathtt{A}[]^{\mathtt{A}[\ n_a\ \ldots\ ]\tau\ \ldots \to \tau'}}\ \ \alpha^{\mathtt{A}[\ n_a\ \ldots\ ]^{\tau}}\ \ldots\right)^{\tau'}\ \ldots\right]^{\mathtt{A}[\ n_f\ \ldots\ ]^{\tau'}}$$

where $\rho = \mathit{length}\,(n_f\ \ldots) > 0$

$((\alpha\ \ldots)\ \ldots) = ((\mathit{Cells}_{\rho}\,[\![v]\!])\ \ldots)^{\top}$

Duplicating cells:

$$\left([f\ \ldots]^{\mathtt{A}[\ m\ \ldots\ ]^{\mathtt{A}[\ n\ \ldots\ ]\tau\ \ldots \to \tau'}}\ \ v^{\mathtt{A}[\ m'\ \ldots\ ]^{\tau}}\ \ldots\right)^{\sigma}$$

$$\mapsto_{lift}\ \left(\mathit{Dup}_{\mathtt{A}[\ n\ \ldots\ ]\tau\ \ldots \to \tau', \iota}\,[\![[f\ \ldots]]\!]\ \ \mathit{Dup}_{\mathtt{A}[\ m'\ \ldots\ ]\tau, \iota}\,[\![v]\!]\ \ldots\right)^{\sigma}$$

where $(m\ \ldots), (m'\ \ldots)\ \ldots$ not all equal

$\iota = \mathit{Max}\,[\![(m\ \ldots), (m'\ \ldots)\ \ldots]\!]$

Converting nested to non-nested:

$$\left[[v\ \ldots]^{\mathtt{A}[m\ \ldots\ ]\tau}\right]^{\mathtt{A}[n\ \ldots\ ]^{\mathtt{A}[m\ \ldots\ ]\tau}}\ \mapsto_{collapse}\ [\mathit{Append}\,[\![(v\ \ldots)\ \ldots]\!]]^{\mathtt{A}[n\ \ldots\ m\ \ldots\ ]\tau}$$

Applying a type abstraction:

$$\left(\mathtt{T\text{-}APP}\ (\mathtt{T}\lambda\,[x\ \ldots]\ e^{\tau})^{\forall[x\ \ldots]\tau}\ \sigma\ \ldots\right)^{\tau[(x\ \leftarrow_t\ \sigma)\ \ldots]}\ \mapsto_{T\beta}\ e^{\tau}\,[(x\ \leftarrow_t\ \sigma)\ \ldots]$$

Applying an index abstraction:

$$\left(\mathtt{I\text{-}APP}\ (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\ e^{\tau})^{\Pi[(x\ \gamma)\ \ldots]\tau}\ \iota\ \ldots\right)^{\tau[(x\ \leftarrow_i\ \iota)\ \ldots]}\ \mapsto_{I\beta}\ e^{\tau}\,[(x\ \leftarrow_i\ \iota)\ \ldots]$$

Projecting from a dependent sum:

$$\left(\mathtt{UNPACK}\ \left(\langle x\ \ldots\,|y\rangle = (\mathtt{PACK}\ \ \iota\ \ldots\ \ v^{\tau})^{\tau'}\right)\ e^{\sigma}\right)^{\sigma}\ \mapsto_{proj}\ e^{\sigma}\,[(x\ \leftarrow_i\ \iota)\ \ldots\ (y\ \leftarrow_e\ v)]$$

Fig. 6. Small-step operational semantics for Remora

$\boxed{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash l : \tau}$

$$\frac{}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash \mathit{num} : \mathtt{Num}} \quad (\text{T-Num})$$

$$\frac{(x : \tau) \in \mathit{\Gamma}}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash x : \tau} \quad (\text{T-Var})$$

$$\frac{\tau \cong \sigma \qquad \mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash l : \tau}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash l : \sigma} \quad (\text{T-Equiv})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash l_j : \tau \quad \text{for each } l_j \in l \ \ldots \\ \mathit{Product}\,[\![n \ \ldots]\!] = \mathit{Length}\,[\![elt \ \ldots]\!]\end{array}}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash [l \ \ldots]^{\mathtt{A}_{[n \ \ldots]}\tau} : \mathtt{A}_{[n \ \ldots]}\tau} \quad (\text{T-Array})$$

$$\frac{\mathit{\Gamma}, (x : \tau) \ \ldots ; \mathit{\Delta}; \mathit{\Theta} \vdash e : \sigma}{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\lambda\,[(x\ \tau) \ \ldots]\ e) : \\ (\tau \ \ldots \to \sigma)\end{array}} \quad (\text{T-Abst})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash e : \mathtt{A}_\iota\sigma \ \ldots \to \tau \\ \mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash e'_j : \mathtt{A}_{\kappa_j}\sigma_j \quad \text{for each } j \\ \iota' = \mathit{Max}\,[\![\iota, \kappa \ \ldots]\!]\end{array}}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash \big(e\ e' \ \ldots\big) : \mathtt{A}_{\iota'}\tau} \quad (\text{T-App})$$

$$\frac{\mathit{\Gamma}; \mathit{\Delta}, x \ \ldots ; \mathit{\Theta} \vdash e : \tau}{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\mathtt{T}\lambda\ [x \ \ldots]\ e) : \\ (\forall\,[x \ \ldots]\,\tau)\end{array}} \quad (\text{T-TAbst})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash l : (\forall\,[x \ \ldots]\,\sigma) \\ \mathit{\Delta}; \mathit{\Theta} \vdash \tau_j \quad \text{for each } j \\ \text{no } \tau_j \text{ is an array type}\end{array}}{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\mathtt{T\text{-}APP}\ l\ \tau \ \ldots) : \\ \sigma[(x \leftarrow_t \tau) \ \ldots]\end{array}} \quad (\text{T-TApp})$$

$$\frac{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta}, (x :: \gamma) \ \ldots \vdash e : \tau}{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\mathtt{I}\lambda\ [(x) \ \ldots]\ e) : \\ (\mathit{\Pi}\,[(x\ \gamma) \ \ldots]\,\tau)\end{array}} \quad (\text{T-IAbst})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash e : (\mathit{\Pi}\,[(x\ \gamma) \ \ldots]\,\tau) \\ \mathit{\Theta} \vdash \iota_j :: \gamma_j \quad \text{for each } j\end{array}}{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\mathtt{I\text{-}APP}\ e\ \iota \ \ldots) : \\ \tau[(x \leftarrow_i \iota) \ \ldots]\end{array}} \quad (\text{T-IApp})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash e : \tau\,[(x \leftarrow \iota) \ \ldots] \\ \mathit{\Theta} \vdash \iota_j :: \gamma_j \quad \text{for each } j\end{array}}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash (\mathtt{PACK}\ \iota \ \ldots\ \ e) : (\mathit{\Sigma}\,[(x\ \gamma) \ \ldots]\,\tau)} \quad (\text{T-Pack})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash e : (\mathit{\Sigma}\ [(x\ \gamma) \ \ldots]\,\sigma) \\ \mathit{\Gamma}, y : \sigma; \mathit{\Delta}; \mathit{\Theta}, (x :: \gamma) \ \ldots \vdash e' : \tau \\ \mathit{\Delta}; \mathit{\Theta} \vdash \tau\end{array}}{\mathit{\Gamma}; \mathit{\Delta}; \mathit{\Theta} \vdash \big(\mathtt{UNPACK}\ (\langle x \ \ldots |y\rangle = e)\ e'\big) : \tau} \quad (\text{T-Unpack})$$

Fig. 7. Type judgment for Remora

$\boxed{\Delta;\Theta \vdash \tau}$

$$\frac{}{\Delta;\Theta \vdash B}\ (\text{K-Base}) \qquad \frac{x \in \Delta}{\Delta;\Theta \vdash x}\ (\text{K-Var}) \qquad \frac{\Delta;\Theta \vdash \tau \quad \Theta \vdash \iota :: \mathtt{Shape}}{\Delta;\Theta \vdash \mathtt{A}_\iota\tau}\ (\text{K-Array})$$

$$\frac{\Delta;\Theta \vdash \tau_j \text{ for each } j \qquad \Delta;\Theta \vdash \sigma}{\Delta;\Theta \vdash (\tau\ \ldots \to \sigma)}\ (\text{K-Fun}) \qquad \frac{\Delta;\Theta,(x :: \gamma)\ \ldots \vdash \tau}{\Delta;\Theta \vdash (\Pi\ [(x\ \gamma)\ \ldots]\ \tau)}\ (\text{K-DProd})$$

$$\frac{\Delta;\Theta,(x :: \gamma)\ \ldots \vdash \tau}{\Delta;\Theta \vdash (\Sigma\ [(x\ \gamma)\ \ldots]\ \tau)}\ (\text{K-DSum}) \qquad \frac{\Delta, x\ \ldots ;\Theta \vdash \tau}{\Delta;\Theta \vdash (\forall\ [x\ \ldots]\ \tau)}\ (\text{K-Univ})$$

$\boxed{\Theta \vdash \iota :: \gamma}$

$$\frac{n \in \mathbb{N}}{\Theta \vdash n :: \mathtt{Nat}}\ (\text{S-Nat}) \qquad \frac{(x :: \gamma) \in \Theta}{\Theta \vdash x :: \gamma}\ (\text{S-Var}) \qquad \frac{\Theta \vdash \iota_j :: \mathtt{Nat} \text{ for each } j}{\Theta \vdash [\ \iota\ \ldots] :: \mathtt{Shape}}\ (\text{S-Shape})$$

$$\frac{\Theta \vdash \iota :: \mathtt{Nat} \qquad \Theta \vdash \kappa :: \mathtt{Nat}}{\Theta \vdash (+\ \iota\ \kappa) :: \mathtt{Nat}}\ (\text{S-Plus})$$

Fig. 8. Kind and index sort judgments for Remora

frame into which the results of the lifted function will be assembled. If the set of frames has no maximum, then the function-application term is ill-typed.

In type application (T-TApp), the type arguments are required to be atomic element types, not array types. This permits functions with polymorphic argument types like "any scalar," written as $\mathtt{A}_{[]}\mathtt{t}$, where `t` is a ∀-bound type variable. Without this restriction, `t` might stand for anything, so $\mathtt{A}_{[]}\mathtt{t}$ could describe any array type.

The remaining type rules are standard. Destructing a dependent sum must not allow the hidden shape information to leak out, so T-Unpack requires that the result type be well-formed *without* relying on the index variables that were bound while checking the body. A specific but straightforward rule is needed for each base type: T-Num is given as an example.

The type-equivalence relation ≅ is a congruence based on relating nested array types and non-nested array types. An array of type $\mathtt{A}_{[m\,\ldots]}(\mathtt{A}_{[n\,\ldots]}\tau)$ is equivalent to an array of type $\mathtt{A}_{[m\,\ldots\,n\,\ldots]}\tau$. This is the transformation which will be made by a *collapse* step at run time and suggests the fully-collapsed version of a type as its canonical form. The reverse is analogous to breaking an array into a frame of cells. This type equivalence allows us to express restrictions on a part of a function argument's shape. For example, `append` has type:

$$\begin{aligned}&\forall[t]\ \Pi\,[(\mathtt{m\ Nat})\ (\mathtt{n\ Nat})\ (\mathtt{d\ Shape})]\\ &\qquad \big(\mathtt{A}_{[\mathtt{m}]}\,(\mathtt{A}_{\mathtt{d}}\,\mathtt{t})\big)\ \big(\mathtt{A}_{[\mathtt{n}]}\,(\mathtt{A}_{\mathtt{d}}\,\mathtt{t})\big) \to \big(\mathtt{A}_{[(+\ \mathtt{m}\ \mathtt{n})]}\,(\mathtt{A}_{\mathtt{d}}\,\mathtt{t})\big)\end{aligned}$$

Any two array types which have the same atom type and whose shapes differ only in the first dimension can be described using `append`'s argument types. Checking equivalence can be done using a solver for the quantifier-free fragment of the theory of lists of natural numbers.

Sorting rules classify type indices as `Nat`s or `Shape`s. Index-level addition is only permitted for `Nat`s—it does not lift to `Shape`s. The kinding rules check whether types are well-formed (in effect, there is only one kind). K-Array accepts an array type as long as its element type is well-formed and its index is a `Shape`. The element type may itself be another array type. K-Univ brings type variables into the environment, and K-DProd and K-DSum bind index variables at their specified sorts.

The type system will only ascribe to an array the shape it will actually have when computed:

Theorem 1 (Type soundness). *If $\vdash l : \tau$, then one of:*

- *There is some v such that $l \mapsto^* v$;*
- *l diverges; or*
- *There exist some $E, \pi, v \ldots$ such that $l \mapsto^* E[((\pi\ v \ldots))]$, where $\vdash \pi : \sigma \ldots \to \sigma'$, and $\vdash v_i : \sigma_i$ for each i*

That is, a well-typed program completes, diverges, or produces an error due to partial primitive operations, such as division by zero. In other words, this means that shape errors cannot occur. Array-indexing errors are impossible, as there is no indexing operator: Remora programs operate on aggregates rather than extracting and operating on individual elements.

5 Planned work

The remaining work falls into two broad categories: reducing the programmer's annotation burden through type inference; and compilation to efficient, explicitly-looping code.

Remora currently has a type checker, based on the formal typing rules presented in Section 4. It still needs type inference, to provide a more human-friendly interface to the core language. Remora types can carry a lot of detail, and full annotations are too much to expect of a programmer.

Even for dependent types with restrictions as in Dependent ML, standard type inference requires generating a single constraint describing the whole program. If that constraint is satisfiable, then the program is typable. Unfortunately, the generated constraint may contain deeply nested alternating quantifiers, which makes it too uninformative for program elaboration. While being unable to elaborate into a fully-annotated program is somewhat dissatisfying for ML, it is a deal-breaker for Remora, where the choice of type indices can affect the behavior of a program.

Remora also lacks a compiler. My thesis work is intended as a first step into this area: writing a serial compiler will connect the language-design work to issues that concern all possible target machines. A back-end for parallel hardware must still fall back on a sequential code generator. Real hardware has a limited amount of parallelism available, so with sufficiently large data to process, the program will be forced to scatter computation more coarsely than in Remora's formal semantics. Both C [19] and LLVM [20] are possible target languages with a clear path to generating explicitly parallel code. C code can be retargetted to a multi-core machine using OpenMP [4], and LLVM can be directed towards GPGPU compilation using the NVPTX [1] back-end. However, developing a collection of techniques for automatically parallelizing Remora code is beyond the scope of this dissertation. An automatically parallelizing compiler will require more detailed cost models of various target machines in order to reason about how possible output programs would perform, and the performance evaluation for such a project is necessarily much more involved. This remains a clear opportunity for future work, building on the work of this dissertation.

The formal semantics provides clear guidance on how Remora programs ought to behave when run, but it offers only a very roundabout way of reasoning about the validity of anything but the most naïve object code. Efficient compilation demands the freedom to rewrite the user's program, at the source level and in some lower-level representation. Instead of the mechanistic reduction rules for Remora, which encode a very specific abstract machine or evaluation strategy, source-level transformations would be more easily justified by a general calculus for rank-polymorphic array programming, analogous to using the full theory of λ calculus to reason about call-by-value Scheme programs. A low-level internal representation must allow the compiler to express decisions about performance-influencing aspects which cannot be directly stated in source code. While some rudimentary loop fusion is possible at source level, such as turning $(\texttt{add1}\ (\texttt{sqrt}\ [1\ 2\ 3]))$ into $\big((\lambda\ ((x : \texttt{A}_{[]}\texttt{Int}))\ (\texttt{add1}\ (\texttt{sqrt x})))\ [1\ 2\ 3]\big)$, unfolding operations, such as `iota` and `reshape`, cannot be merged into the folds that consume them in a language where the iteration space is always reified. No looping construct is available to express the fused unfold-fold. Furthermore, the details of array allocation and layout are strictly implicit in Remora, but efficiency depends on not creating actual superfluous copies of replicated array cells or allocating a new array for every `rotate` or `reverse`—these intermediate values should instead be "rotated" out of spatial dimensions onto the time axis.

In this section, I describe the work to be done to address these four problems:

– Type inference and elaboration with restricted dependent types
– A decision procedure for array shapes
– A general array calculus
– A low-level representation of arrays and operations on them

5.1 Type inference

As a stepping stone towards type inference for Remora, I have been developing a type inference and elaboration algorithm for Dependent ML which permits local reasoning about type indices. Instead of turning a source program into a single global constraint with deeply nested quantifiers, every application of a dependently-typed function generates a local constraint to be solved before type checking continues. This reduces the asymptotic cost by effectively limiting the quantifier nesting depth—even for just natural numbers with addition, the cost of constraint solving grows doubly exponentially with the quantifier depth. The tradeoff for this cost reduction is that it over-eagerly prunes off some possible solutions to the global constraint. Each function application's index arguments must be chosen before later applications are considered, so the type checker has a chance of painting itself into a corner.

However, typical code should not have many cases where index arguments are ambiguous. A DML function which consumes a vector would generally require an index argument representing the vector's length (as in `foldr`), or possibly all but a constant portion of its length (such as `cdr`, which takes an index argument `n` and a vector with length `n+1`). A problematic example is a function that takes indices `m` and `n` and then a vector with length `m+n`. Seeing the vector argument's type tells us its length, but not how that length should be broken up. If the function then returns a type that uses `n`, we risk choosing a decomposition that keeps the rest of the program from typechecking. In Remora, it is sensible to require the programmer to disambiguate these unusual cases with explicit annotations. Type indices can affect the dynamic semantics of a program, so the ambiguity is not just about how to certify that the program is well-behaved but about what exactly that behavior is. Consider a constant function with a shape argument: $\left(\mathtt{I}\lambda\,[s : \mathtt{Shape}]\left(\lambda\ (x : \mathtt{A}_s\mathtt{Int})\ [0]_{[]}\right)\right)$. When we apply it to an argument, any suffix of the argument's shape could be the intended cell shape `s`, and any prefix could be the application's intended frame shape. The programmer can effectively specify what `s` should be by using an explicit index application or annotating the function with the intended monomorphic type.

My inference algorithm is based on bidirectional type checking [25]. The high-level idea behind bidirectional type checking is that thinking through a type derivation involves some type rules where the resulting type is known and just being confirmed by its sub-derivations and some where no specific result type is expected. For example, in a function application, once we know the function's type, we also know what type to expect the argument term to have. These two modes of use, "synthesis" and "checking," represent information about a term's type flowing up and down the syntax tree, respectively. The core idea in bidirectional checking is to make this distinction explicit in the typing rules themselves, with mutually recursive judgment forms for synthesis and checking. The synthesis judgment can be considered a function from environments and terms to types, while the checking judgment is still used as a predicate on evironment-term-type triples. Here is a standard rule for typing function application:

$$\frac{\begin{array}{c}\Gamma \vdash e_f \uparrow \sigma \to \tau \\ \Gamma \vdash e_a \downarrow \sigma\end{array}}{\Gamma \vdash (e_f\ e_a) \uparrow \tau} \quad (\textsc{Syn-App})$$

It says that we can synthesize ($\uparrow$) a type τ for the application form by first synthesizing a type $\sigma \to \tau$ for the function position and then checking ($\downarrow$) the argument position against σ, the synthesized argument type. Typically, a rule for explicit annotations allows a synthesis goal to become a checking goal:

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau} \quad (\textsc{Syn-Ann})$$

Similarly, a subtyping rule allows a checking goal to become a synthesis goal:

$$\frac{\mathit{\Gamma} \vdash e \uparrow \sigma \qquad \mathit{\Gamma} \vdash \sigma <: \tau}{\mathit{\Gamma} \vdash e \downarrow \tau} \quad (\textsc{Chk-Sub})$$

Dunfield and Krishnaswami [10] expanded on bidirectional type checking to better address polymorphism. A limitation of the simple application rule above is that if synthesis discovers that the function e_f is polymorphic, we have no clear σ to check e_a against. Requiring that synthesis produce a monomorphic type is unfeasible because we are forced to choose that type without knowing anything about the actual argument the monomorphized function will consume. They introduce a third "application judgment" form, which essentially asks, "When we apply something of this possibly polymorphic type to this particular argument term, what type is the result?" The application judgment form is used for choosing the type arguments at a polymorphic function's call site, so that every function can stay polymorphic until it is actually used. In order to make this possible, the environment structure is allowed to contain specially designated existential type variables, standing for type arguments that may not have been chosen yet. These variables can be freely added to the environment by an application rule and instantiated later by a subtyping rule. When a polymorphic function type has had all of its quantifiers peeled off, with corresponding existential type variables added to the environment, the final step is to ask what type we get from applying a monomorphic function that has some unsolved variables inside it. Since we can get our hands on actual input and output types, we can make sure that the argument term matches the input type, using the checking judgment.

In order to accommodate checking against a type with unsolved variables in it, the subtyping relation[4] is based on the specificity of the types being compared. The rules for subtyping also update the environment to map existential type variables to their solutions. By updating these variables, the subtyping judgment serves as a constraint solver. The question it answers is not whether one type is a subtype of another, but what must be true in order for the one type to be a subtype of the other. Suppose our polymorphic function, with its quantifiers removed, asks for an $\hat{\alpha} \to \hat{\beta}$. When we apply it to a `String` $\to$ `Int`, the subtype check asks how to instantiate $\hat{\alpha}$ and $\hat{\beta}$ to make $\hat{\alpha} \to \hat{\beta} <:$ `String` $\to$ `Int` hold. The subtype judgment answers with $\hat{\alpha} \mapsto$ `String` and $\hat{\beta} \mapsto$ `Int`. Delaying the selection of type arguments is critical for the application judgment to work. Immediately choosing a type argument when a quantifier is removed requires too much global knowledge.

I have extended this from parametric polymorphism to handle restricted dependent types and produce an elaborated version of the program, where type and index arguments are passed explicitly. Where Dunfield and Krishnaswami's subtyping solver uses unification-based rules for instantiating type variables, I instantiate type-index variables using Inez, an ILP modulo theories solver [22]. The environment structure accumulates $\mathit{\Pi}$-bound index variables and existential index variables generated by the application judgment. The rule which invokes Inez is the subtype instantiation rule for indexed type families, which only accepts one type as a subtype of the other if corresponding index arguments are equal. The general form for the constraint which queries Inez for index equality is

$$\forall (\hat{m} \ldots) \,.\, (\phi \Rightarrow \exists (\hat{n} \ldots) \,.\, \iota = \kappa) \,.$$

The variables $\hat{m} \ldots$ are the environment's $\mathit{\Pi}$-bound variables and solved existential variables, and $\hat{n} \ldots$ are the unsolved existential variables. Previously discovered facts about type indices can influence this decision. If we already know that $\hat{x} + 2 = \hat{y}$ and $2\hat{x} = \hat{z}$, then it is safe to accept $\hat{z} + 4 = 2\hat{y}$. These facts are included as the conjunction ϕ. Algebraic manipulation involved in adapting this constraint to Inez produces symbolic existential witnesses in terms of the universal variables. If the entire formula is valid, these witnesses can be added into the environment as solutions to the corresponding existential index variables and used as explicit index arguments in the elaborated program.

[4] Recall, this lets us use a synthesis result to satisfy a checking goal.

5.2 Theory of array shapes

Currently, the DML inference algorithm uses natural numbers as type indices, with addition as the only index-level computation. This makes Presburger arithmetic the theory of type indices. The transition from DML to Remora will require a solver that can reason about array shapes, both for type checking and type inference.

An array shape is a list of natural numbers, representing the array's size in each dimension. A type checker must be able to decide whether two shapes are equal in order to check type equality. It must also be able to decide whether one shape is a prefix of the other in order to check whether two arrays' shapes are compatible as co-arguments in a lifted function application.

The relevant mathematical structure for this problem is a free monoid. A monoid is an algebra with one associative binary operation, which has an identity element. In the case of a free monoid, no two elements are equal unless their equality is implied by associativity and elision of the identity element. This is exactly the behavior of lists built using an `append` operator: `append`ing an empty list onto anything leaves the original list unchanged, and `append` itself is associative. Associativity means any (variable-free) expression denoting an element of the monoid can have its syntactic nesting structure "forgotten," producing a sequence of the generators that make up this particular monoid element. Intuitively, the algebra of lists is a *free* monoid because no additional structure can be forgotten. For the algebra of array shapes, the generators are lists containing single dimensions, *i.e.*, vector shapes. This reflects the notion that any array type can be uniquely viewed as a regular, but possibly very deep, nesting of vector types.

I am considering two possible strategies for axiomatizing the theory of array shapes for use in type-inferring Remora. In one strategy, the monoid theory is kept isolated, and reasoning techniques from the formal methods world are used to handle its interaction with the theory of single dimensions (which is Presburger arithmetic). This should make it easier to scale down to the quantifier-free fragment used in checking type equality and searching for a possible frame shape: a pre-existing decision procedure for monoids can be used as a background solver with Inez.

Alternatively, both the free-monoid axioms and the Presburger axioms can be merged into a single theory meant to have its own unified decision procedure. Since ILP modulo theories does not deal directly with quantifiers, this unified version may be easier to scale up to the quantification needed for selecting implicit index arguments in type inference.

Free monoid alone

$$\Sigma = \langle \Box, +\!\!+, \sqsubseteq, = \rangle$$

The monoid includes one function symbol, $+\!\!+$ (list append), which is associative. The constant symbol $\Box$ denotes the monoid's unique identity element (the empty list). The predicate $\sqsubseteq$ is the prefix relation on lists:

$$a \sqsubseteq b \iff \exists c.\, b = a +\!\!+ c$$

This theory is agnostic about the monoid's generators. Decision procedures exist for several fragments of this theory, typically phrased for a finite alphabet of generators, but, using all of $\mathcal{N}$ as the generators can be encoded with the alphabet $\{0, 1\}$ by writing each $n \in \mathcal{N}$ as 1^n and using 0 as a separator. So a decision procedure for the free monoid with two generators suffices. For quantifier-free reasoning, this can be linked to Inez, leaving the ILP solver to deal with integer arithmetic. The usual monoid axioms

$$a +\!\!+ (b +\!\!+ c) = (a +\!\!+ b) +\!\!+ c$$

$$\Box +\!\!+ a = a +\!\!+ \Box = a$$

are augmented with an "editor axiom", which states that if two concatenations are equal, then there is some completing sublist which is the overlap between the larger argument in each sum (demonstrated in Figure 9):

$$a +\!\!+ b = c +\!\!+ d \implies \exists w.\, (a +\!\!+ w = c \wedge w +\!\!+ d = b) \vee (c +\!\!+ w = a \wedge w +\!\!+ d = b)$$

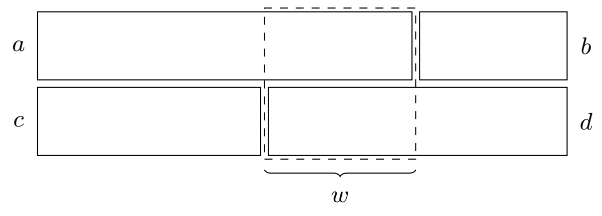

Fig. 9. Editor axiom, visualized: w is the overlapping portion of a and d.

Whole combined theory

$$\Sigma = \langle n \in \mathbb{N}, \square, +, [\cdot], +\!\!+, \sqsubseteq, = \rangle$$

This adds the usual addition and constant symbols from Presburger arithmetic and a function $[\cdot]$ for constructing singleton lists. This works most smoothly as a logic with two sorts for naturals and lists of naturals. If necessary, it can be made one-sorted by having list-consuming functions (and predicates) coerce numbers into singleton lists and number-consuming functions coerce lists to their first elements, as in Furia's presentation [11].

5.3 Using the theory of lists

In Remora, the array type is indexed with a shape, which is itself a sequence of natural numbers. The index language includes addition (only permitted on naturals) and appending (only permitted on shapes):

$$\iota, \kappa ::= n \mid x \mid [\, \iota \ldots] \mid (+\ \iota\ \kappa) \mid (+\!\!+\ \iota\ \kappa) \qquad \textit{(type indices)}$$

Type inference includes selecting appropriate index arguments to dependent functions. For example, we may have a vector-addition Remora function `vec+` with type $\Pi\,[(l : \texttt{Nat})]\, \texttt{A}_{[l]}\texttt{Int}, \texttt{A}_{[l]}\texttt{Int} \to \texttt{A}_{[l]}\texttt{Int}$. If we apply it to two arrays with types $\texttt{A}_{[14]}\texttt{Int}$ and $\texttt{A}_{[3,(+\ n\ 2)]}\texttt{Int}$, we need to choose an appropriate value of l and suitable frame shapes for the function and each argument. Presumably, we have n in our environment as a `Nat`, as well as some facts about it which we might use to conclude that $n + 2$ can be equal to 14. Naively following the local DML style of inference would generate a constraint of the form

$$\forall [n : \texttt{Nat}].(\phi \implies \exists [l : \texttt{Nat}; f_0, f_1, f_2, f_\top : \texttt{Shp}].\psi)$$

The premise ϕ contains the equalities we already know about indices (having stored them in the type-checking environment). The conclusion ψ should specify that the frame shapes f_0, f_1, f_2 must all have $f_\top$ as their join under the prefix relation and that the i^{th} argument's actual shape must be decomposable into $f_i +\!\!+ [l]$. Each decomposability requirement is a simple equality, *e.g.*, $[3, (n + 2)] = f_1 +\!\!+ [l]$ (which would imply $f_1 = [3]$ and $l = n + 2$). The prefix requirement (a component of ψ) can be written as:

$$\begin{gathered} f_0 \sqsubseteq f_\top \land f_1 \sqsubseteq f_\top \land f_1 \sqsubseteq f_\top \\ \land\, (f_0 = f_\top \lor f_1 = f_\top \lor f_2 = f_\top) \end{gathered}$$

A solution to the above constraint would choose all of the index arguments and frame shapes needed for an application form. Both ϕ and ψ are conjunctions of equalities and prefix constraints.

Unfortunately, the full $\forall^*\exists^*$ fragment of the theory of lists is undecidable. It appears frame shapes can be handled in isolation by quantifier-free reasoning (in the explicitly typed version of Remora, the type checker already does this). Although the decidability of the $\forall^*\exists$ fragment is unknown, avoiding generating existential variables for frame shapes is not a viable solution. The programmer can always have a function take too many shape arguments (in fact, three shape arguments is already too many, as $\forall\exists^3$ is undecidable). The largest known-decidable fragments are the $\exists^*$ and the $\forall^*$ fragments. Two general avenues for dealing with this are worth considering.

First, the generated constraints may still reside in a decidable sub-fragment within the $\forall^*\exists^*$ fragment of the theory. With freely available existential quantifiers, the prefix constraints can be written as equalities, so the constraint matrix is transformable to a Horn formula. Decidability specifically for the Horn fragment of the $\forall^*\exists^*$ theory of lists does not appear to have been studied, but it is known to simplify the validity problem in other logics.

Second, the $\exists^*$ fragment may be useful for characterizing when an incomplete type inference algorithm is guaranteed to work, versus when it can only provide "best effort" on an undecidable problem: If an implicit index argument does not refer to any shape variables bound in the environment, we are effectively working in the existential fragment of the list theory (alternatively, we are asking satisfiability questions about the quantifier-free theory). This is always the case for `Nat` arguments, so only functions which are polymorphic in their cell shape can run afoul of this. Since the satisfiability question is decidable, it is possible to recognize when a constraint fits within it. If hiding all shape variables (and facts about them) in the environment produces a constraint satisfiable with quantifier-free reasoning about lists, type inference can use that satisfying assignment. Otherwise, the inferencer would then try using a broader (but incomplete) solver or ask for more hints from the programmer.

There is no perfectly suitable off-the-shelf decision procedure, so I will need to develop one which can integrate with the type inference strategy described earlier. The exact bounds of decidability are still unknown, so I will need to spend some time exploring the space of decision procedures for list theories before I can choose a specific plan of attack. Decidability of the theory of lists corresponds to determining whether suitable index arguments exist when applying an index-polymorphic function to some arrays; if no large enough fragment of the theory is decidable, the inferencer will have to rely on programer-supplied annotations in some situations.

5.4 Array calculus

The semantics of Remora given in Figure 6 follows a strict, machine-like operation. Although this is useful for specifying a programming language, a more general (but presumably confluent) calculus for the rank-polymorphic programming model would help develop and justify the sort of code transformations a compiler ought to perform. Instead of requiring fully-evaluated argument arrays to split into cells, a "full β-reduction" rule should be able to fire whenever enough frame structure is visible to permit frame/cell decomposition of the argument terms. As soon as we have the ability to refer to individual cells of an argument array, even as non-normalized program terms, we should be able to substitute them into the appropriate function body. Evaluation in this manner is like distributing the work of computing one aggregate function before the actual input data is available. In effect, computing that input data also becomes a distributed task. This is good for actual parallel hardware: for example, a program which keeps issuing short tasks to the GPU and retrieving their results, only to send a re-partitioned array back to the GPU later, can spend most of its running time just waiting on the PCI bus. This program would be much better off letting a result keep living in device memory so it can be used again by later computation. The theory of such a calculus should identify opportunities to evaluate under to this "scatter early, gather late" strategy, even fusing operations with differing frame shapes.

This also opens the door for reasoning about computing with delayed arrays. Delaying individual atoms is too fine-grained for reasonable cost, but the ability to delay large cells within an array may be useful, such as in an internal representation that uses futures to express parallelization decisions.

Where Remora's abstract-machine rules in Figure 6 are carefully set up to have only one applicable reduction rule at a time, a general array calculus will need a confluence theorem to ensure that a compiler which uses this theory will not change program behavior. The type-soundness proof will also have to be adjusted to accommodate newer, looser evaluation rules.

5.5 Representation of arrays

Many of Remora's primitive or implicit operations on arrays don't really do anything to the array elements themselves; they just rearrange them in some regular way. Representing an array at run time as a record with a buffer and some information on indexing rules makes operations like transposition and rotation cheap (even free in some cases). Lifting a representation of this “view shift” on the index space into the language used for internal representation should allow the compiler more freedom to take small (*i.e.*, easily justifiable) steps in transforming a program. The theoretical basis seems solid enough—the Agda model of rank polymorphism that Pierre-Evariste Dagand and I worked on at INRIA represents arrays as functions from (valid) index vectors to array elements. Lots of operator-specific reasoning can be replaced with reasoning about affine functions on array indices. They are closed under composition (and can be kept in simple, closed form by a compiler), let shape manipulation caused by lifting disappear in trivial loop invariants, make lots of facts about how shape transformations interact clearly visible, *etc.*

A low-level IR for dealing with arrays can represent an array value as one of

– $(\texttt{buffer}\ ptr)$
– $(\texttt{view}\ e_{idx}\ e_{array})$

A `buffer` uses a store location as a flat array of some scalar element type, which should be indexed with a single natural. A `view` presents an underlying array (computed by e_{array}) as another array by having the index tuple transformed by the index function (given by e_{idx}). To build a rank-m array from a rank-n array, the function must consume m-tuples and produce n-tuples. With annotations for input and output types, this operation might be partially usable in the explicitly typed version of Remora, but it looks like it will be more broadly useful in a later IR where type indices get reified at the term level. Suppose we have at address `p` in memory the 2×3 array [[1 2 3] [4 5 6]]. This indexing pattern is represented with:

$$e_1 = (\texttt{view}\ (\lambda\ \langle x_0, x_1\rangle . 3x_0 + x_1)\ (\texttt{buffer}\ \texttt{p}))$$

Rotating it on its major axis can be written as:

$$e_2 = (\texttt{view}\ (\lambda\ \langle x_0, x_1\rangle . \langle (x_0 + 1)\ \texttt{mod}\ 3, x_1\rangle)\ e_1)$$

The rotated array could be reversed along its minor axis:

$$e_3 = (\texttt{view}\ (\lambda\ \langle x_0, x_1\rangle . \langle x_0, 2 - x_1\rangle)\ e_2)$$

Then we can replicate the 1-cells, to produce a $2 \times 4 \times 3$ array, by dropping the index element which selects which row in a plane to use:

$$e_4 = (\texttt{view}\ (\lambda\ \langle x_0, x_1, x_2\rangle . \langle x_0, x_2\rangle)\ e_3)$$

This three-dimensional array can be transposed on its outer two axes:

$$e_5 = (\texttt{view}\ (\lambda\ \langle x_0, x_1, x_2\rangle . \langle x_1, x_0, x_2\rangle)\ e_4)$$

Nested `view`s can be merged by composing their indexing functions.

I spent the summer of 2015 at Nvidia, working on a compiler for Remora, based on an IR with explicit mapping of functions and replication of arrays. Translation into this IR relies heavily on type information to determine which sub-arrays of each arguments must be replicated how many times and the iteration space of each `map`. Note that the `view` form permits cell replication with no actual copying of underlying data. A possible IR which uses `view` might be:

$$\begin{aligned} e ::= {} & (e\ e\ \ldots) \mid (\texttt{VEC}\ n \mid e\ \ldots) \mid (\texttt{MAP}\ e_{frm}\ e_{fn}\ e_{arg}\ \ldots \mid e_{shp}) \mid (\texttt{REP}\ e\ e_{frm} \Rightarrow\ e_{exp}) \qquad \textit{(expressions)} \\ & \mid (\texttt{view}\ e_{idx}\ e_{arr}) \mid \langle e\ \ldots\rangle \mid (\texttt{LET}\ \langle x\ \ldots\rangle = e\ \texttt{in}\ e') \mid \texttt{LET}\ x = e\ \texttt{IN}\ e' \mid (\lambda\ (x\ \ldots)\ e) \mid x \mid b \end{aligned}$$

At this level, `VEC` serves as the basic array notation. It would be replaced by `buffer` in an IR with explicit allocation and sharing.

6 Schedule

Here is a rough schedule for the work remaining to complete my dissertation; it has me defending in April, 2019.

- Type inference: 12 months
    - Decision procedure for theory of lists
    - Grow Dependent ML inference up to handle Remora
    - Integrate decision procedure
- Compilation: 6 months
    - Array calculus and metatheory
    - IR design
    - Code generation
- Evaluation: 1 month
- Writing: 6 months

7 Related work

First, I summarize work already mentioned in prior sections:

- Dependent ML
- Bidirectional type checking
- Integer linear programming modulo theories
- Theory of a free monoid

I follow with an overview of related work in designing array-oriented programming languages.

7.1 Dependent ML

Dependent ML [26] introduced a dependent type system with limits on the values to which types can be applied. Rather than making the full term language available for use as type indices, Dependent ML uses a separate, simpler index language. This makes it possible to check equivalence of types without having to check equivalence of program terms (which may themselves include indexed types which must be checked for equivalence, and so on). This type system refines ML's original type system, and an erasure pass converts a Dependent ML program into an ML program with the same behavior. The development of Dependent ML is largely independent of the details of the chosen language of type indices. While later work on Dependent ML introduced singleton types for numbers [27], so that array-bounds checks could be performed by the type checker, Remora pursues a design where arrays are not consumed by accessing individual elements by position.

7.2 Bidirectional type checking

Local bidirectional type checking grew as a technique for handling type systems with features which proved hostile to complete, global inference. Rather than assuming a completely un-annotated program, bidirectional algorithms leverage programmer-supplied type annotations, which are often present for human readability. Pierce and Turner [25] explicitly focus on eliminating "common and silly" annotations, which explain tedious bookkeeping steps to the compiler without providing useful insight to a programmer reading the code. Explicitly naming the type arguments whenever applying a polymorphic function is burdensome, but stating the type of a polymorphic function itself is tolerable or even desirable. The fundamental technique is to split the typing judgment into two judgments, one for checking types when a particular type for a term is expected and one for synthesizing a type for a term. Invoking the checking judgment rather than synthesis allows information about types to propagate, *e.g.*, a function-application form need only synthesize a type for the function position, which then identifies a particular type to check for in argument position. Typically, a checking rule for subtyping and a synthesis rule for explicitly annotated terms allows inference to switch between checking and synthesis when needed. If a term must be checked at a particular type, that obligation can be satisfied by synthesizing any subtype of that desired type. When the programmer has provided a type annotation, we can synthesize that type for the annotated term, as long as it checks at that type.

Dunfield and Krishnaswami [10] improve the handling of higher-rank and impredicative polymorphism. They add a third judgment specifically for selecting type arguments when applying polymorphic functions. Earlier bidirectional type systems handled subtyping by progressively refining upper and lower bounds on an unsolved type. This strategy is replaced with a unification-like judgment, which builds up the syntactic form of an unsolved type. The main distinction from unification is in handling scope so that an existential type variable can only be resolved to a type in which all type variables are bound at the point the existential was introduced. While Dunfield and Krishnaswami do not give an elaboration algorithm, this restriction on how type variables may be resolved ensures that their solutions do not rule out elaboration. Since the subtyping relation itself is roughly, "τ is at least as polymorphic as σ," the system can handle a function's demand for a polymorphic argument (*i.e.*, higher-rank polymorphism). The combination of a third judgment and the new subtyping solver also makes this type checking system more flexible about selecting polymorphic types as type arguments.

7.3 ILP modulo theories

Remora's index language combines Presburger arithmetic with lists. Rather than combining standalone decision procedures for these two theories using the Nelson-Oppen method [23], I plan to leverage recent work by Manolios and Papavasileiou on ILP Modulo Theories [22]. Where an SMT solver augments a SAT solver with the ability to query a separate solver for some background theory, to allow checking satisfiability of a formula which uses functions and predicates from that theory, IMT does the same with an ILP solver. SMT uses a conventional DPLL strategy [8] with extra rules for interacting with the theory solver [24]. These extra rules allow the solver to learn by asking the background theory solver whether some collection of formulas (treated like literals by the original DPLL rules) entails some other formula. IMT similarly modifies the branch-and-cut strategy, such that a subproblem may be ruled infeasible by a black-box theory solver.

In Remora, both frame agreement and index instantiation are cases of solving equality constraints on natural numbers and lists of naturals, where terms representing individual numbers may be built up as linear combinations. ILP already handles equality on linear combinations of naturals, and IMT offers a way to integrate reasoning about lists with an off-the-shelf ILP solver.

7.4 Free monoid

Furia [11] gives a formulation of the theory of a free monoid and summarizes several decidability results. A monoid is an associative algebra with an identity element. The theory's signature includes an equality predicate $=$ and concatenation operator $+\!\!+$ as well as a collection of unspecified unary predicates. One of these predicates can be used to identify the monoid's identity element $\square$. The structure described by the theory is then $(A^*, +\!\!+, \square)$, where A is a set of "generator" elements of the monoid. When A has a single element (free monoid on one generator), the theory of concatenation is effectively Presburger arithmetic: $+\!\!+$ is addition, and $\square$ is zero. The free monoid generated by $\mathbb{Z}$, sequences of integers, can be encoded in a free monoid with four generators. Each integer is written out in unary by repeating generator a_1, with positive and negative sign represented using generators a_2 and a_3, and a_4 is used as a delimiter. Then a positive integer k is written as $a_4 a_2 {a_1}^k a_4$, and a negative integer $-k$ is written as $a_4 a_3 {a_1}^k a_4$. Any concatenation of these encoded integers is uniquely decodable. A similar strategy encodes the free monoid on $\mathbb{N}$, but removing the need for an explicit sign means only two generators (a_1 and a_4 above) will suffice.

The axioms are:

- Associativity: $a +\!\!+ (b +\!\!+ c) = (a +\!\!+ b) +\!\!+ c$
- Identity: $\square +\!\!+ a = a +\!\!+ \square = a$

– Editor: $a +\!\!+\, b = c +\!\!+\, d \implies \exists w.\, (a +\!\!+\, w = c \wedge w +\!\!+\, d = b) \vee (c +\!\!+\, w = a \wedge w +\!\!+\, d = b)$

Whenever two concatenations produce the same result, the "editor" axiom effectively identifies an overlapping region between the larger argument of each concatenation. Remora also calls for a binary predicate $\sqsubseteq$, which determines whether one list is a prefix of another. This relation is definable within the $\exists$-fragment of the theory: $a \sqsubseteq b \iff \exists c.\, b = a +\!\!+\, c$.

We do not have a function for indexed reference to sequence elements, and this is not expressible even when the theory is extended with Presburger arithmetic (due to the lack of an induction principle for numbers). The quantifier-free fragment also cannot express `first` and `rest` functions, though the $\exists$-fragment can.

This theory differs from conventional theories of arrays primarily in that terms are not histories of reads and writes. Early work on the theory of arrays also lacked an extensionality principle (*i.e.*, that two arrays are equal if all corresponding elements are equal). Later work did eventually integrate Presburger arithmetic, both for array indices and array elements.

7.5 Other language design work

Rank-polymorphic computation originated with APL [14]. At first, APL only lifted scalar functions to either one aggregate argument and one scalar or two aggregate arguments of the same shape. Extending APL to accommodate functions which naturally operate on aggregates as their fundamental, non-lifted case brought up the notion of a function's expected cell rank, with the remainder of an argument's shape, *i.e.*, the frame, driving iteration. This changed the shape compatibility rule to require one scalar frame or two identical frames. By the time Iverson designed J [17] as a successor to APL, the lifting rule had expanded to allow non-equal frames as long as one was a prefix of the other. Remora further generalizes the lifting rule by permitting lifting for higher-order functions and allowing arbitrarily many arguments, as long as there is one argument frame which is prefixed by all of the others.

In NESL [2], the fundamental data structure is a one-dimensional sequence. Parallelism is available via a `map`-like comprehension form. NESL improves on its predecessors by allowing the operation performed in the comprehension to itself include parallel comprehensions. Instead of conceptually distributing the work of processing each sequence element to a strictly sequential functional unit, NESL eliminates the distinction between code that is a piece of a parallel kernel and code that is not. Nesting parallel operations is the important bit of flexibility needed to properly support nested parallel sequences. However, the pervasiveness of ragged data in NESL code (*e.g.*, a sequence of differing-length sequences), prohibits an APL-like shape-based implicit control construct. NESL's data model imposes a "vector of vectors of ... vectors" view, rather than leaving the choice of views to the function being applied (and possibly lifted). Instead, code which operates on aggregate data must still depend on that data's nesting structure.

The implementation [3] is based on transforming nested parallelism into flat parallelism. A nested parallel operation is still ultimately made up of flat sequence comprehensions, with its nesting structure reflecting the nesting structure of the data. NESL concatenates these leaf comprehensions into one large flat comprehension, which can then be split up into chunks as appropriate for the available compute resources. This vectorization transform was ported to Haskell with Data Parallel Haskell [5]. The programming model is a good conceptual fit for Haskell, which already has a comprehension syntax for lists. DPH adds its own similar notation, which is then desugared into vectorizable function applications. The chunk splitting is performed in an intermediate representation, so that parallel operations can be fused without accidentally ruling out a desirable task chunking.

Other projects have also brought parallelism over regular arrays into Haskell. In Repa [18], operations on arrays can quantify over the shapes of their arguments, though a `map` or `zipWith` function must be called explicitly and used on functions which process scalars. Unlike in J, a rank-2 array is only a matrix of scalars. The type of matrices is not equal to the type of vectors-of-vectors, and the frame and cell decomposition cannot be chosen by the function being applied. Instead, an explicit, rank-specific transformation is needed to enable a vector-consuming operation to work on matrices.

A similar programming interface is provided by Accelerate [6], a domain-specific language embedded in Haskell. Computation written in Accelerate itself is transformed into CUDA kernels to run on a GPU, and several glue functions enable interaction between Accelerate code and native Haskell code (which runs on the CPU). Arrays stored in the GPU's device memory are distinguished at the type level from arrays stored in host memory, as are scalar data residing within an array in device memory. CUDA kernels cannot themselves kick off other CUDA parallel jobs, so nested parallelism is not available. To enforce this restriction, functions provided by Accelerate only take GPU arrays (for first-order arguments) and functions on GPU scalars (for higher-order arguments). Data can be injected into an Accelerate program with `use`, effectively making an explicit transfer to device memory. Once an Accelerate computation has been constructed, it is invoked with `CUDA.run`, which generates and executes CUDA device code then transfers the result array back to host memory. Explicit data movement and lack of nested parallelism make Accelerate a "lower-level" language than Remora but a potentially interesting compilation target for rank-polymorphic array programming.

Jay and Cockett developed FISh [15, 16], a language based on explicitly mapping operations over regular arrays, whose static semantics uses two separate judgments. Rather than including a conventional array type, the type judgment in FISh only concerns itself with the types of array elements. Array shapes are checked by a second judgment, independent of their actual elements. This judgment effectively evaluates a FISh program just for data shape without actually computing any array elements. FISh thus separates well-typedness (corresponding to sensible use of base data) from well-shapedness (corresponding to sensible control flow). However, the shape rules are too restrictive for practical use: being unable to have shapes that depend on dynamically computed data rules out operations like `filter`, `read-vector`, and `iota`.

Nvidia designed NOVA [7] as a statically typed, lisp-like language for explicitly parallel GPU computation. GPU operations are based on vectors (not higher-rank arrays), with nested parallelism implemented via NESL-like flattening. For example, a function which uses a parallel `reduce` cannot itself be used in another parallel `reduce`. NOVA is intended as a lower-level intermediate language for compiling parallel DSLs. Parallelizable kernels are distinguished from ordinary functions, and all aggregate operations, including `map`, are explicitly stated. NOVA is also built to handle foreign functions easily, to facilitate integrating a DSL with its host language. In discussions at Nvidia, we found it awkward to express some computations which look like flat parallelism in Remora but appear as nested parallelism in NOVA, such as vector-matrix addition. NOVA does not have a `replicate` or similar operation or a `map`-like operation that allows each argument to be traversed differently. This impedance mismatch led to choosing LambdaJIT [21] as a more suitable target for an eventual parallel compilation project. LambdaJIT extends LLVM's IR with operations which set up and launch CUDA kernels.

The Futhark project [12] explores compiling for GPUs. Futhark is a GPU-targetted language focused on replacing common C and Fortran code patterns with second-order operators. The basis library includes `map`, `reduce`, `replicate`, `transpose`, *etc.*, and the compiler is built for reasoning about how these operators behave. This allows very aggressive fusion of aggregate computations [13]. Like NOVA, Futhark is intended not as novel language design but for developing GPU compilation techniques and as a compilation target for higher-level data-parallel languages.

References

1. User guide for nvptx back-end (2016), `http://llvm.org/docs/NVPTXUsage.html`
2. Blelloch, G.: NESL: A nested data-parallel language (version 3.1). Tech. rep. (1995)
3. Blelloch, G., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. Journal of Parallel and Distributed Computing 21, 102–111 (1994)
4. Board, O.A.R.: Openmp application program interface version 4.0 (2013), `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
5. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: In DAMP 2007: Workshop on Declarative Aspects of Multicore Programming. ACM Press (2007)
6. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. pp. 3–14. DAMP '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1926354.1926358`
7. Collins, A., Grewe, D., Grover, V., Lee, S., Susnea, A.: Nova: A functional language for data parallelism. In: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Program-

ming. pp. 8:8–8:13. ARRAY'14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2627373.2627375

8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (Jul 1962), http://doi.acm.org/10.1145/368273.368557
9. Dolan, S.: Fun with semirings: A functional pearl on the abuse of linear algebra. SIGPLAN Not. 48(9), 101–110 (Sep 2013), http://doi.acm.org/10.1145/2544174.2500613
10. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. SIGPLAN Not. 48(9), 429–442 (Sep 2013), http://doi.acm.org/10.1145/2544174.2500582
11. Furia, C.A.: What's Decidable about Sequences?, pp. 128–142. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15643-4_11
12. Henriksen, T., Elsman, M., Oancea, C.E.: Size slicing: A hybrid approach to size inference in futhark. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 31–42. FHPC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2636228.2636238
13. Henriksen, T., Oancea, C.E.: A t2 graph-reduction approach to fusion. In: Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 47–58. FHPC '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2502323.2502328
14. Iverson, K.E.: A programming language. John Wiley & Sons, Inc., New York, NY, USA (1962)
15. Jay, C.B.: The fish language definition. Tech. rep. (1998)
16. Jay, C.B., Cockett, J.: Shapely types and shape polymorphism. In: Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming. pp. 302–316. Springer Verlag (1994)
17. Jsoftware, Inc.: Jsoftware: High-performance development platform, http://www.jsoftware.com/
18. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. pp. 261–272. ICFP '10, ACM, New York, NY, USA (2010)
19. Kernighan, B.W.: The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edn. (1988)
20. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (Dec 2002), *See* http://llvm.cs.uiuc.edu.
21. Lutz, T., Grover, V.: Lambdajit: A dynamic compiler for heterogeneous optimizations of stl algorithms. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 99–108. FHPC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2636228.2636233
22. Manolios, P., Papavasileiou, V.: ILP modulo theories. In: Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044. pp. 662–677. CAV 2013, Springer-Verlag New York, Inc., New York, NY, USA (2013), http://dx.doi.org/10.1007/978-3-642-39799-8_44
23. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (Oct 1979)
24. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). J. ACM 53(6), 937–977 (Nov 2006), http://doi.acm.org/10.1145/1217856.1217859
25. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. 22(1), 1–44 (Jan 2000), http://doi.acm.org/10.1145/345099.345100
26. Xi, H.: Dependent types in practical programming. Ph.D. thesis, Pittsburgh, PA, USA (1998), aAI9918624
27. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. pp. 249–257. PLDI '98, ACM, New York, NY, USA (1998)

A Examples of progressive code transformation

Here are examples of how some compiler for a rank-polymorphic programming language might reason through the compilation process for small example programs. These programs when compiled naively generate large intermediate structures, but equivalent code written by a human in von Neumann style does not. Each compilation sequence was generated by hand—that is, I played the role of a "sufficiently smart compiler." However, each code transformation step is justified by facts which a compiler should easily recognize about the code, which argues that producing good object code from an actual compiler is a realistic goal.

A.1 Convolution

Convolution computes a weighted moving average of samples in a signal. The finite impulse response filter is used as a vector of weights. Here, `f` and `s` are implicit index arguments indicating the lengths of the filter and signal respectively. `iota%` is a variant on the `iota` function which takes an index-level shape argument rather than a term-level vector of dimensions. We use the resulting vector, `[0 1 ...(f-1)]`, as the shift amounts for `rotate`. This produces a matrix where each row is the input `signal` rotated by a different amount. Multiplying this matrix by the vector of filter weights means each row is multiplied by the corresponding weight. Finally, we sum along the major axis to get the convolved signal.

```
(define (convolve (weights : Num [f])
                  (signal : Num [s]) : Num [s])
  (reduce + 0 (* weights (rotate (iota% [f]) signal))))
```

Rewrite the function body to use explicit `MAP` and `REPLICATE` operations, which are taken from the IR I used at Nvidia:

```
(reduce + 0
        (MAP [f s] *
             ;; Grow each scalar cell by [s]
             (REPLICATE weights [] [s])
             (MAP [f] rotate
                  (iota% [f])
                  ;; Grow each [s] cell by [f]
                  (REPLICATE signal [s] [f]))))
```

Rewrite the replications as view shifts. The `weights` vector becomes a matrix by replicating each scalar into a vector, so the view shift uses only the row index and ignores the column. The `signal` expands by replicating the entire vector, making the row index irrelevant:

```
(reduce + 0
        (MAP [f s] *
             (view (λ (x0 x1) x0) weights)
             (MAP [f] rotate
                  (iota% [f])
                  (view (λ (x0 x1) x1) signal))))
```

The `rotate` operation is also equivalent to a view shift:

```
(reduce + 0
        (MAP [f s] *
             (view (λ (x0 x1) x0) weights)
             (MAP [f] (λ (rot arr)
                        (view (λ (x) (mod (+ x rot) (length arr)))
                              arr))
                  (iota% [f])
                  (view (λ (x0 x1) x1) signal))))
```

The view-shifted signal is a `[f]`-frame of `[s]`-cells, according to the `MAP` that consumes it. Since all of the cells are the same (specifically, they are all `signal`), we can pull that cell into the mapped function's body instead of passing it as an argument:

```
(reduce + 0
        (MAP [f s] *
             (view (λ (x0 x1) x0) weights)
             (MAP [f] (λ (rot)
                        (view (λ (x) (mod (+ x rot)
                                          (length signal)))
                              signal))
                  (iota% [f]))))
```

The `length` function just pulls index-level information into the term level. We can replace `(length signal)` with `signal`'s actual length:

```
(reduce + 0
        (MAP [f s] *
             (view (λ (x0 x1) x0) weights)
             (MAP [f] (λ (rot)
                        (view (λ (x) (mod (+ x rot) s))
                              signal))
                  (iota% [f]))))
```

The inner MAP produces a matrix, which is then consumed by the outer MAP and never used again. Eliminating this temporary matrix requires fusing the inner and outer MAPs. Fusing maps with the same frame is more straightforward, so we'll move towards writing the view shift on weights and the inner MAP to work over [f s], using an indexing operation. Any array can become a MAP by using iota% to construct an array of loop iteration indices. To MAP over a higher-rank frame, use multiple iota% arrays to build the pieces of the iteration vector:

```
(reduce + 0
        (MAP [f s] *
             (MAP [f s] (λ (f* s*)
                          (IDX [f* s*]
                               (view (λ (x0 x1) x0) weights)))
                  (view (λ (x0 x1) x0) (iota% [f]))
                  (view (λ (x0 x1) x1) (iota% [s])))
             (MAP [f s] (λ (rot i)
                          (IDX [rot i]
                               (view (λ (x) (mod (+ x rot) s))
                                     signal))
                  (view (λ (x0 x1) x0) (iota% [f]))
                  (view (λ (x0 x1) x1) (iota% [s]))))))
```

Now that the inner and outer MAPs all use the same frame shape, we can fuse them:

```
(reduce + 0
        (MAP [f s]
             (λ (f* s* rot i)
               (* (IDX [f* s*] (view (λ (x0 x1) x0) weights))
                  (IDX [rot i] (view (λ (x) (mod (+ x rot) s))
                                     signal)))
             (view (λ (x0 x1) x0) (iota% [f]))
             (view (λ (x0 x1) x1) (iota% [s]))
             (view (λ (x0 x1) x0) (iota% [f]))
             (view (λ (x0 x1) x1) (iota% [s])))))
```

Two of the argument arrays are identical (thus redundant):

```
(reduce + 0
        (MAP [f s]
             (λ (f* s*)
               (* (IDX [f* s*] (view (λ (x0 x1) x0) weights))
                  (IDX [f* s*] (view (λ (x) (mod (+ x f*) s))
                                     signal)))
             (view (λ (x0 x1) x0) (iota% [f]))
             (view (λ (x0 x1) x1) (iota% [s])))))
```

We turn the `MAP` into a comprehension closer to a `FOR` loop, with loop indices made explicit:

```
(reduce + 0
        (FOR (f0 [f])
             (FOR (s0 [s])
                  ((λ (f* s*)
                     (* (IDX [f* s*]
                             (view (λ (x0 x1) x0)
                                   weights))
                        (IDX [f* s*]
                             (view (λ (x) (mod (+ x f*) s))
                                   signal))))
                   (IDX [f0 s0]
                        (view (λ (x0 x1) x0)
                              (iota% [f])))
                   (IDX [f0 s0]
                        (view (λ (x0 x1) x1)
                              (iota% [s])))))))
```

Indexing into a view-shifted array means applying the shift function to the index:

```
(reduce + 0
        (FOR (f0 [f])
             (FOR (s0 [s])
                  ((λ (f* s*)
                     (* (IDX f* weights)
                        (IDX (mod (+ s* f*) s) signal))
                     (IDX f0 (iota% [f]))
                     (IDX s0 (iota% [s])))))))
```

Indexing into `iota`'s result vector produces the index itself:

```
(reduce + 0
        (FOR (f0 [f])
             (FOR (s0 [s])
                  ((λ (f* s*)
                     (* (IDX f* weights)
                        (IDX (mod (+ s* f*) s) signal))
                     f0
                     s0)))))
```

β-reduce the loop body:

```
(reduce + 0
        (FOR (f0 [f])
             (FOR (s0 [s])
                  (* (IDX f0 weights)
                     (IDX (mod (+ s0 f0) s) signal)))))
```

A `reduce` around a `FOR` loop can have the loop iterations update a mutable accumulator array. We know exactly how much to allocate based on the result type, `Num [s]`:

```
(let (result (make-vector s 0)) ; init length s vector of 0s
  (FOR (f0 [f])
       (FOR (s0 [s])
            (vector-set!
             result f0
             (+ (IDX s result)
                (* (IDX f0 weights)
                   (IDX (mod (+ s0 f0) s) signal)))))))
```

Because of rotation, the last $\texttt{f} - 1$ output samples depend on the first input sample. This complicates the transformation into an in-place update of `signal`: we still need to save $\texttt{f} - 1$ extra numbers in order to keep the earliest samples usable at the end of the computation.

A.2 Matrix multiplication

Matrix multiplication starts by taking each row of the $\texttt{L} \times \texttt{M}$ left operand and multiplying it by the entire $\texttt{M} \times \texttt{N}$ right operant (*i.e.*, by every column). This produces an $\texttt{L} \times \texttt{M} \times \texttt{N}$ intermediate array. Summing along the second axis gives the $\texttt{L} \times \texttt{N}$ matrix product.

```
(define (m* (a : Num [L M])
            (b : Num [M N]))
  (#r(0 1 2)reduce
     ;; The reducing function is vector-sum
     (λ ((v : Num [N])
         (w : Num [N])) (+ v w))
     ;; Base case is a length N vector of all 0
     (reshape* [N] 0)
     (#r(1 2)* a b)))
```

Function reranking is shorthand for a sort of type-altering η-expansion. In `reduce`'s case, the rank-3 third argument (with type `Num [L M N]`) is to be split into its rank-2 pieces (with type `Num [M N]`). The first and second arguments to `reduce` are being used at their actual rank. The `a` argument to `*`, which has rank 2 and type `Num [L M]` should be split by our new η-expanded `*` into rank-1 (*i.e.*, `Num [M]`) pieces:

```
((λ ((op   : (-> (Num [N]) (Num [N]) (Num [N])))
     (zero : Num [N])
     (arr  : Num [M N]))
   (reduce op zero arr))
   (λ ((v : Num [N])
       (w : Num [N]))
     (+ v w))
   (reshape* [N] 0)
   ((λ ((a0 : Num [M])
        (b0 : Num [M N]))
      (* a0 b0))
    a b))
```

Rewrite using explicit MAP and REPLICATE operations:

```
(MAP [L] (λ ((op   : (-> (Num [N]) (Num [N]) (Num [N])))
             (zero : Num [N])
             (arr  : Num [M N]))
           (reduce op zero arr))
     ;; Grow [] to [L] by replicating scalar cells
     (REPLICATE (λ ((v : Num [N])
                    (w : Num [N]))
                  (MAP [N] + v w))
                [] [L])
     ;; Grow [N] to [L N] by replicating vector cells
     (REPLICATE (reshape* [N] 0) [N] [L])
     (MAP [L] (λ (()(a0 : Num [M])
                  (b0 : Num [M N])
                (MAP [M N] *
                     ;; Grow [M] to [M N] by replicating
                     ;; scalar cells
                     (REPLICATE a0 [] [N])
                     b0))
          ;; This argument set the frame shape, so no
          ;; replication is needed
          a
          ;; Grow [M N] to [L M N] by replicating
          ;; the sole matrix cell
          (REPLICATE b [M N] [L])))
```

Use view shifts for replication:

```
(MAP [L] (λ ((op   : (-> (Num [N]) (Num [N]) (Num [N])))
             (zero : Num [N])
             (arr  : Num [M N]))
           (reduce op zero arr))
     (view (λ (x0) [0])
           (λ (()(v : Num [N])
               (w : Num [N])
             (MAP [N] + v w)))
     (view (λ (x0 x1) [x1]) (reshape* [N] 0))
     (MAP [L] (λ ((a0 : Num [M])
                  (b0 : Num [M N]))
                (MAP [M N] *
                     (view (λ (x0 x1) x0) a0)
                     b0))
          a
          (view (λ (x0 x1 x2) [x1 x2]) b)))
```

Reshaping can also be written as a view shift:

```
(MAP [L] (λ ((op   : (-> (Num [N]) (Num [N]) (Num [N])))
             (zero : Num [N])
             (arr  : Num [M N]))
           (reduce op zero arr))
     (view (λ (x0) [0])
           (λ ((v : Num [N])
               (w : Num [N]))
             (MAP [N] + v w)))
     (view (λ (x0 x1) [x1]) (view (λ (x0) 0) 0))
     (MAP [L] (λ ((a0 : Num [M])
                  (b0 : Num [M N]))
                (MAP [M N] *
                     (view (λ (x0 x1) x0) a0)
                     b0))
          a
          (view (λ (x0 x1 x2) [x1 x2]) b)))
```

We now have a nested view shift on the `0` argument, so we can compose shifting functions:

```
(MAP [L] (λ (()(op   : (-> (Num [N]) (Num [N]) (Num [N])))
             (zero : Num [N])
             (arr  : Num [M N])
           (reduce op zero arr))
     (view (λ (x0) [0])
           (λ ((v : Num [N])
               (w : Num [N]))
             (MAP [N] + v w)))
     (view (λ (x0 x1) [0]) 0)
     (MAP [L] (λ ((a0 : Num [M])
                  (b0 : Num [M N]))
                (MAP [M N] *
                     (view (λ (x0 x1) x0) a0)
                     b0))
          a
          (view (λ (x0 x1 x2) [x1 x2]) b)))
```

The view-shifted b is invariant during the inner MAP, so we cann pull b itself into the mapping function:

```
(MAP [L] (λ ((op   : (-> (Num [N]) (Num [N]) (Num [N]))
             (zero : Num [N])
             (arr  : Num [M N]))
           (reduce op zero arr))
     (view (λ (x0) [0])
           (λ ((v : Num [N])
               (w : Num [N]))
             (MAP [N] + v w)))
     (view (λ (x0 x1) [0]) 0)
     (MAP [L] (λ ((a0 : Num [M]))
                (MAP [M N] *
                     (view (λ (x0 x1) x0) a0)
                     b))
          a))
```

Similarly, the vector-adding function and zero vector are invariants of the outer MAP:

```
(MAP [L] (λ ((arr  : Num [M N]))
           (reduce (λ ((v : Num [N])
                       (w : Num [N]))
                     (MAP [N] + v w))
                   (view (λ (x0 x1) [0]) 0)
                   arr))
     (MAP [L] (λ ((a0 : Num [M]))
                (MAP [M N] *
                     (view (λ (x0 x1) x0) a0)
                     b))
          a))
```

We now have a MAP over [L] whose argument is also a MAP over [L], so we can fuse them by composing the mapping functions:

```
(MAP [L] (λ ((a0 : Num [M]))
           (reduce (λ ((v : Num [N])
                       (w : Num [N]))
                     (MAP [N] + v w))
                   (view (λ (x0 x1) [0]) 0)
                   (MAP [M N] *
                        (view (λ (x0 x1) x0) a0)
                        b)))
     a)
```

Rewriting the MAP over [M N] as a nested loop will make it easier to merge the outer loop with the surrounding reduce, which only operates along the [M] part of that frame:

```
(MAP [L] (λ ((a0 : Num [M]))
           (reduce (λ ((v : Num [N])
                       (w : Num [N]))
                     (MAP [N] + v w))
                   (view (λ (x0 x1) [0]) 0)
                   (FOR (m [M])
                        (FOR (n [N])
                             (* (IDX [m n]
                                     (view (λ (x0 x1) x0) a0))
                                (IDX [m n] b))))))
     a)
```

Apply the view shift on indexing a0:

```
(MAP [L] (λ (()(a0 : Num [M])
           (reduce (λ ((v : Num [N])
                       (w : Num [N]))
                     (MAP [N] + v w))
                   (view (λ (x0 x1) [0]) 0)
                   (FOR (m [M])
                        (FOR (n [N])
                             (* (IDX [m] a0)
                                (IDX [m n] b))))))
     a)
```

Now we can merge the reduce with its immediately nested FOR loop by explicitly allocating and mutating a result vector. The allocation must initialize it to a vector that is equal to the view-shifted 0:

```
(MAP [L] (λ ((a0 : Num [M]))
           (let ((resvec (make-vector N 0)))
             (begin
               (FOR! (m [M])
                     (set! resvec
                           (MAP [N] +
                                resvec
                                (FOR (n [N])
                                     (* (IDX [m] a0)
                                        (IDX [m n] b))))))
               resvec)))
     a)
```

Instead of building a whole new vector to replace the old one, it would be much better to update the old one in-place, one scalar at a time:

```
(MAP [L] (λ ((a0 : Num [M]))
           (let ((resvec (make-vector N 0)))
             (begin
               (FOR! (m [M])
                     (FOR! (n [N])
                           (vector-set! resvec
                                        n
                                        (+ (IDX [n] resvec)
                                           (* (IDX [m] a0)
                                              (IDX [m n] b))))))
               resvec)))
     a)
```

For more convenient reading, we'll use a shorthand for nested FOR loops:

```
(MAP [L] (λ ((a0 : Num [M]))
           (let ((resvec (make-vector N 0)))
             (begin
               (FOR*! ((m [M])
                       (n [N]))
                      (vector-set! resvec
                                   n
                                   (+ (IDX [n] resvec)
                                      (* (IDX [m] a0)
                                         (IDX [m n] b)))))
               resvec)))
     a)
```

Rewrite the outer MAP as a FOR comprehension. Each iteration produces one vector from a, which is built by having a view shift turn a vector index into a matrix index which touches the corresponding vector cell:

```
(FOR (l [L])
     ((λ ((a0 : Num [M]))
        (let ((resvec (make-vector N 0)))
          (begin
            (FOR*! ((m [M])
                    (n [N]))
                   (vector-set! resvec
                                n
                                (+ (IDX [n] resvec)
                                   (* (IDX [m] a0)
                                      (IDX [m n] b)))))
            resvec)))
      (view (λ (x0) [l x0]) a)))
```

β-reduce the loop body:

```
(FOR (l [L])
     (let ((resvec (make-vector N 0)))
       (begin
         (FOR*! ((m [M])
                 (n [N]))
                (vector-set! resvec
                             n
                             (+ (IDX [n] resvec)
                                (* (IDX [m]
                                        (view (λ (x0) [l x0]) a))
                                   (IDX [m n] b)))))
         resvec)))
```

Collapse the indexed view shift on a:

```
(FOR (l [L])
     (let ((resvec (make-vector N 0)))
       (begin
         (FOR*! ((m [M])
                 (n [N]))
                (vector-set! resvec
                             n
                             (+ (IDX [n] resvec)
                                (* (IDX [l m] a)
                                   (IDX [m n] b)))))
         resvec)))
```

Instead of building up several separately-allocated vectors and then stitching them together, allocate contiguous space for all of them before starting the outermost loop:

```
(let ((resmat (view (λ (x0 x1) [(+ (* x0 N) x1)])
                    (make-vector (* L N) 0))))
  (begin
    (FOR! (l [L])
          (let ((resvec (view (λ (x0) [l x0]) resmat)))
            (FOR*! ((m [M])
                    (n [N]))
                   (vector-set! resvec
                                n
                                (+ (IDX [n] resvec)
                                   (* (IDX [l m] a)
                                      (IDX [m n] b)))))))
    resmat))
```

Constant-propagate `resvec` into the inner loop body:

```
(let ((resmat (view (λ (x0 x1) [(+ (* x0 N) x1)])
                    (make-vector (* L N) 0))))
  (begin
    (FOR! (l [L])
          (FOR*! ((m [M])
                  (n [N]))
                 (vector-set! (view (λ (x0) [l x0]) resmat)
                              n
                              (+ (IDX [n] (view (λ (x0) [l x0])
                                                  resmat))
                                 (* (IDX [l m] a)
                                    (IDX [m n] b))))))
    resmat))
```

Apply the view shift for indexing `resmat`, changing both the `IDX` form that reads from `resmat` and the `vector-set!` form that writes to it:

```
(let ((resmat (view (λ (x0 x1) [(+ (* x0 N) x1)])
                    (make-vector (* L N) 0))))
  (begin
    (FOR! (l [L])
         (FOR*! ((m [M])
                 (n [N]))
                (vector-set! resmat
                             [l n]
                             (+ (IDX [l n] resmat)
                                (* (IDX [l m] a)
                                   (IDX [m n] b))))))
    resmat))
```

Again, reduce rightward drift for the reader's convenience:

```
(let ((resmat (view (λ (x0 x1) [(+ (* x0 N) x1)])
                    (make-vector (* L N) 0))))
  (begin
    (FOR*! ((l [L])
            (m [M])
            (n [N]))
           (vector-set! resmat
                        [l n]
                        (+ (IDX [l n] resmat)
                           (* (IDX [l m] a)
                              (IDX [m n] b)))))
    resmat))
```