An array-oriented language with static rank polymorphism

Justin Slepak, Olin Shivers, and Panagiotis Manolios

Northeastern University
{jrslepak,shivers,pete}@ccs.neu.edu

Abstract. The array-computational model pioneered by Iverson's languages APL and J offers a simple and expressive solution to the "von Neumann bottleneck." It includes a form of rank, or dimensional, polymorphism, which renders much of a program's control structure implicit by lifting base operators to higher-dimensional array structures. We present the first formal semantics for this model, along with the first static type system that captures the full power of the core language.

The formal dynamic semantics of our core language, Remora, illuminates several of the murkier corners of the model. This allows us to resolve some of the model's *ad hoc* elements in more general, regular ways. Among these, we can generalise the model from SIMD to MIMD computations, by extending the semantics to permit functions to be lifted to higher-dimensional arrays in the same way as their arguments.

Our static semantics, a dependent type system of carefully restricted power, is capable of describing array computations whose dimensions cannot be determined statically. The type-checking problem is decidable and the type system is accompanied by the usual soundness theorems. Our type system's principal contribution is that it serves to extract the implicit control structure that provides so much of the language's expressive power, making this structure explicitly apparent at compile time.

1 The Promise of Rank Polymorphism

Behind every interesting programming language is an interesting model of computation. For example, the lambda calculus, the relational calculus, and finite-state automata are the computational models that, respectively, make Scheme, SQL and regular expressions interesting programming languages. Iverson's language APL [7], and its successor J [10], are interesting for this very reason. That is, they provide a notational interface to an interesting model of computation: loop-free, recursion-free array processing, a model that is becoming increasingly relevant as we move into an era of parallel computation.

APL and J's array-computation model is important for several reasons. First, the model provides a solution to Backus's "von Neumann bottleneck" [1]. Instead of using iteration or recursion, all operations are automatically aggregate operations. This lifting is the fundamental control flow mechanism. The iteration space associated with array processing is reified as the shape of the arrays being processed. Though the paradigm is not without implementation challenges of its own, it at least holds out the promise of eliminating the heroic measures required by modern compilers (*e.g.*, the construction of program-dependency graphs and their difficult associated decision procedures [20]) to extract parallelism through the serialised program's obfuscatory encoding.

Second, operator lifting provides a form of polymorphism based on operands' *rank*, or dimensionality. An operation defined for arguments of one rank is automatically defined for arguments of any higher rank. They are thus parameterized over the ranks of their inputs. The operator for scalar addition is also used for adding a vector to a matrix, a scalar to a three-dimensional array, and so forth.

Third, despite its great expressive power, the core computation model is sub-Turing. Lifting operations to work on aggregate structures means the control structure is embedded in the data structure. With a finite data structure representing the program's control structure, all iteration is bounded. Thus APL's computational model has the potential to occupy a "sweet spot" in language design: increased analytic power without surrendering significant expressiveness.

1.1 Addressing the Model's Shortcomings

Iverson received a Turing award for the design of APL, and the language is often cited as an example of beautiful design [4]. Yet the language—and its accompanying model of computation—has received little study from the formal-semantics research community. Iverson worked almost entirely isolated from the rest of the programming-language research community, even adopting his own private nomenclature for his *sui generis* language mechanisms. Iverson never developed a formal semantics, or a static type system for his language designs. The beautiful, crystalline structure of the core language accreted non-general *ad hoc* additions. For example, APL's reduction operator is able to correctly handle empty vectors when the function being folded across the vector is a built-in primitive such as addition or min: base cases are provided for these functions. Programmers who wish to reduce empty vectors with programmer-defined functions, however, are out of luck.

We address many of the shortcomings of the model and its associated language. First, we define a core language that expresses the essence of the rank-polymorphic array-processing model, along with a formal semantics for the language. Besides eliminating ambiguity and pinning down the corner cases, developing the formal semantics enabled us to replace some of APL and J's *ad hoc* machinery with regular, general mechanisms. Our treatment of higher-order functions, for example, is much more general; this, in turn, allows us to extend the basic array-lifting model to permit arrays of functions (that is, in the function position of a function application) as well as arrays of arguments. This effectively generalises the language's computational model from SIMD to MIMD.

With the essence of the array-computational model captured by our untyped core language and its dynamic semantics, we then develop Remora, a language whose static type system makes the rank polymorphism of a program term explicit. Our type system is a significant result for four reasons:

Soundness We provide a safety theorem connecting the well-typed term judgement to the dynamic semantics of the language. Our type system guarantees that a well-typed term will never become stuck due to the shape or rank of an array argument failing to meet the requirements of its operator.

Expressiveness It permits typing a term that produces an array whose shape is itself a computed value. Our type system is based on Xi's Dependent ML[18] and tuned to the specific needs of Remora's rank polymorphism.

Decidability Despite its expressive power, the dependent elements of Remora's type system are constrained to make the type-checking problem decidable.

Control structure It exposes the iteration space. Recall that the point of Iverson's rank polymorphism is to permit programmers to write programs using element operators that are automatically lifted to operate across the iteration space of the aggregate computation. This means that Remora's static types make the implicit, unwritten iteration structure of a Remora term explicit. In short, our static semantics provides the key "hook" by which compilers can reason about the structure of the computation.

We have implemented the semantics we present using PLT Redex [6]. Our hope (for future work) is that we can exploit this type information to compile programs written in the rank-polymorphic array computation model efficiently: either by translating the reified iteration-space axes of an array back to a serialised, nested-loop computation, or by parallelising the program.

Note that Remora is not intended as a language comfortable for human programmers to write array computations. It is, rather, an explicitly typed, "essential" core language on which such a language could be based.

2 Background: Array-Oriented Programming

2.1 Iverson's Model

The essence of Iverson's array-oriented programming model, which appeared in APL [7] and was later expanded in its successor J [10], is treating all data as regular, *i.e.*, hyperrectangular, arrays. The individual scalar elements of an array, such as numbers or booleans, are referred to as *atoms*. Every r-dimensional array has a *shape*, which is a vector of length r giving the dimensions of the hyperrectangle in which its atoms are laid out. The value r is called the array's *rank*: for example, a matrix has rank 2, a vector has rank 1, and a scalar is taken to have rank 0. An array can be represented using only its shape and its atoms.

The notation we will use for arrays looks like $[2, 3, 5]_3$, meaning a 3-vector whose atoms are 2, 3, and 5. A rank 0 array will be written $[12]_\bullet$, with $\bullet$ denoting an empty shape vector. We write $[9, 8, 7, 6, 5, 4]_{2,3}$ for a 2×3 matrix, $[2, 4, 6, 8, 1, 3, 5, 7]_{2,2,2}$ for a $2 \times 2 \times 2$ array, and so on. For readability, it is sometimes convenient to write arrays in a matrix-like layout:

$$\begin{bmatrix}9 & 8 & 7\\ 6 & 5 & 4\end{bmatrix}_{2,3}$$

An array may also be written with unevaluated expressions:

$$\big[(-\ 10\ 1)\ (-\ 10\ 2)\ (-\ 10\ 3)\big]_3$$

Rank Polymorphism and Frame/Cell Decomposition An array can be viewed at several different ranks. A 4×3 numeric matrix can be viewed as a 4×3 *frame* of scalar *cells*, a 4-element frame of 3-vector cells, or a scalar frame whose single cell is a 4×3 matrix. More generally, a rank-r array can be viewed $r + 1$ different ways: from a rank r *frame* containing rank 0 *cells* to a rank 0 frame containing a single rank r cell.

Every function has an expected rank for each of its arguments. The expected rank can be a natural number n, indicating that the argument should be viewed as containing rank n cells contained in a frame of unspecified rank. Simple arithmetic functions such as `+` and `log` expect arguments with rank 0, *i.e.*, scalars. Applying a function expecting a rank n input to an array of higher rank n' applies the function to the array's n-cells, collecting the multiple outputs into the remaining $n' - n$ dimensional frame. A function can also have expected rank of ∞; such functions consume an entire array of arbitrarily high rank, so they are never lifted. For example, `length` extracts the first element of an array's shape vector—how long the array is. The programmer may write a function with negative argument rank $-n$. Lifting then breaks arguments into a rank n frame around cells of unspecified rank (the "$-n$-cells"), and then the function's body processes each cell. A function with -1 argument rank which finds its argument's `length` effectively extracts the second dimension instead of the first.

$$*\ [1,2,3]_3\ \ [10]_\bullet \quad\mapsto\quad *\ [1,2,3]_3\ \ [10,\underline{10},\underline{10}]_{\underline{3}} \quad\mapsto\quad [(*\ 1\ 10),(*\ 2\ 10),(*\ 3\ 10)]_3$$

$$+\ [10,20,30]_3\ \begin{bmatrix}1\ 2\\ 3\ 4\\ 5\ 6\end{bmatrix}_{3,2} \mapsto +\ \begin{bmatrix}10\ \underline{10}\\ 20\ \underline{20}\\ 30\ \underline{30}\end{bmatrix}_{3,\underline{2}}\ \begin{bmatrix}1\ 2\\ 3\ 4\\ 5\ 6\end{bmatrix}_{3,2} \quad\mapsto\quad \begin{bmatrix}(+\ 10\ 1)\ (+\ 10\ 2)\\ (+\ 20\ 3)\ (+\ 20\ 4)\\ (+\ 30\ 5)\ (+\ 30\ 6)\end{bmatrix}_{3,2}$$

Fig. 1. Automatic expansion of array arguments

To lift a function of multiple arguments, the frames must be brought into agreement by duplicating the cells of the smaller-framed argument (the new elements are underlined in Figure 1). After this duplication, all arguments' frames are the same; this permits the cell-wise function application. The way argument arrays are expanded to the same frame means that function application is only valid if one argument's frame is a prefix of the other argument's frame. This is the *prefix agreement* rule introduced by J.

Manipulating the Iteration Space Under this implicit lifting, the iteration space is the argument frame rather than a sequence of loop indices. The programmer is not required to consider the shape of the array as would be necessary when operating on a nested vector with nested calls to map. A function written to alter an RGB pixel can be used as-is to make the same transformation on every pixel in an image or video. If the transformation is the same for all three color channels, it can simply be written as a scalar function. Generalizing the lifting to multiple arguments, an interpolation function can be used on a matrix of "low" and "high" points with a vector of estimated points. J also includes several second-order operators for manipulating the iteration space. For example, `reduce` collapses the -1-cells of an array to a single -1-cell using a specified binary operator, such as using `*` to transform $[2, 4, 5]_3$ into $2 * 4 * 5$. The `prefix` and `suffix` operators apply a function to the successive prefixes or suffixes of an array, viewing the array as a list of cells with unspecified rank. The results are then put together as cells in a list. A `sum` function could be applied by `prefix` to $[2, 4, 5]_3$ to compute the running sum, $[2, 6, 11]_3$. Some operations such as convolution make use of a sliding window iteration pattern, using a `window` operator which applies a given function over a sliding window of a given shape and assembles the results in a frame corresponding to possible window positions.

The programmer can use the `rerank` operator to change the argument rank of a function. The vector-matrix sum example in Figure 1 effectively treats the vector as a column by duplicating its 0-cells. If $+$ is reranked to expect a vector argument, the 1-cell (*i.e.*, the entire vector) is duplicated, so it is used as a row vector. This reorientation technique generalizes to higher-ranked arrays.

By reranking `append`, the programmer can stitch together arrays by sequencing them on a chosen axis. For example, applying `append` to two matrices will place the vectors (*i.e.*, rows) of one matrix after those of the other. This requires that they have the same number of columns. It produces a matrix with as many rows as the two arguments combined. If `append` is reranked to 1, then it acts on corresponding pairs of vectors, so the two matrices are required to have the same number of rows. Each scalar in a row corresponds to one column in the matrix. Thus the number of columns in the resulting matrix is the sum of the numbers of columns in the argument matrices. Reranking also allows the programmer to `reduce` along any chosen axis. The argument is split into cells of the chosen rank, each cell is `reduce`d along its major axis, and the results are reassembled in the wrapper function's frame.

Boxes Wrapping an array in a *box* makes it appear scalar, even if it contains a non-scalar array. This makes it possible to safely produce and consume non-regular arrays. Boxes are handled explicitly—a common pattern in J code for operating on boxed data is to compose `box`, the desired operator, and `unbox`.

2.2 Related Work

Originally, APL implicitly lifted scalar functions to aggregate functions via point-wise application, either on a scalar and an aggregate or on two aggregates of the same shape. APL was later enriched with attribution of rank to functions, meaning the rank a function expects its arguments to have. This led to the "frame of cells" view of an array and gave a sensible way to lift functions defined only for aggregates to operate on aggregates of even higher rank. J uses the more general lifting rule, prefix agreement, which allows the aggregate lifting to handle arrays of non-identical shape. J retains APL's distinction between data, first-order functions, and second-order functions. Implicit aggregate lifting is still limited to first-order functions.

The design of J still handles many situations through specially-chosen default behavior. For example, 0 and the space character are designated as "fill" elements and used to pad shape-mismatched cells resulting from an application so that they can all be assembled into the same frame. An unfortunate consequence is that applying the composition of two functions may have a different result from applying one function and then the other.

Thatte [16] described automatic lifting based on using coercion to insert `map`, `transpose`, etc. where needed, but this system is limited to lifting scalar operations. It cannot, for example, automatically construct vector-matrix addition.

Ragan-Kelley *et al.* present Halide [14], a language for graphics processing. In Halide, the computation to do at each pixel is written separately from the strategy for ordering and parallelizing the pixels' instances of that computation. This is a similar idea to Single Assignment C's `WITH`-loops [15]. Halide is, however, designed specifically for image processing pipelines rather than general numeric programming, which limits its lifting to the pixel-to-image case.

Xi's Dependent ML [18] addressed the intractability of static type checking in dependently-typed languages by limiting type indices to a separate, simpler language. This technique makes it possible to check type equivalence without having to check equivalence of program terms, which themselves may include indexed types which must be checked for equivalence, and so on. An index erasure pass converts a well-typed Dependent ML program into an ML program with the same behavior. By adding singleton types for numbers, bounds checking for array accesses can be done by the type system instead of at run time [19].

Like Remora, Trojahner and Grelck's Qube [17] uses a type system based on Dependent ML to statically verify structural constraints in array computation. However, Remora and Qube differ significantly in both their dynamic and static semantics. Qube, strictly speaking, does not address the "von Neumann" bottleneck: programmers still specify their programs down at the scalar-computation level, using expressions that explicitly index elements from arrays. The structure of the loop is also specific to the function being lifted and the array arguments to which it is being applied, whereas Remora's implicit lifting frees the programmer from having to specify this detail.

Qube's type system, then, is a device for guaranteeing dynamic safety, but does not support the implicit lifting that gives APL its noted elegance and concision. Qube's heavy use of explicit array indexing necessitates the use of singleton and range types, which in turn restrict the programmer's ability to write code that depends on user input.

Blelloch *et al.* created NESL [2, 3], which focuses on explicit mapping over nested one-dimensional arrays. Arrays need not be rectangular—they can be *jagged*. It is possible, for example, to have a 2-array whose elements are a 4-array and a 5-array. Instead of naïvely breaking a parallel map into a task for each sub-array, the NESL compiler uses a vectorization transformation to treat nested arrays as flat vectors. This makes it possible to split the aggregate operation at places other than sub-array boundaries, removing the load imbalance that had previously been associated with mapping over jagged arrays. Data Parallel Haskell [5] has adopted this vectorization technique. Haskell's existing list comprehensions are extended into parallel array comprehensions [13]. NESL and DPH are still based on explicit looping which does not uniformly handle arrays of varying rank as APL/J and Remora do.

More recent work by Keller *et al.* [11] shows how to use Haskell's type system to handle operations involving regular arrays in a shape-polymorphic way. Instances of the typeclass of `Shape`s provide functions for extracting the rank and size of an array of that shape as well as for indexing into the array. Functions on arrays can be parameterized over the shape type and can effectively place lower bounds on the ranks of arrays they accept. This system prevents errors caused by underranked arguments but not those caused by mismatch in individual dimensions and does not support the full prefix agreement rule.

Jay and Cockett [9] separated the shape of a data structure from its type. For operations whose result shape is dependent only on argument shape, it is possible to evaluate the shape portion of a program separately from the data portion. Jay puts this to work in FISh [8], where arrays have both shape and element type. Evaluating only the shapes of a program ensures that shape-related errors cannot happen at run time, but requiring operators to determine their output shapes only from their argument shapes is unworkably restrictive. For example, it disallows critical functions such as `iota`, `reshape`, and `readvec`.

3 An Untyped Array Language

In J, functions are not first-class, and automatic lifting is restricted to first-order functions. Lifting a function-producing function would allow the application to produce an array of result functions. For example, in Figure 2, we apply a higher-order function, `curry-add`, to two vectors. The result of the first application is a vector of functions, which we then apply to a vector of numbers. In order to do this, we must extend the lifting rule.

Function application itself can be thought of as an operation with expected ranks—that is, in a function-application expression, both function and argument can be arrays, as shown in the second half of Figure 2. Application requires a rank 0 array of functions and requires the arguments to have ranks expected by those functions. All functions in the array must agree as to their argument ranks. $[(\texttt{curry-add}\ 1),\ (\texttt{curry-add}\ 2)]_2$ is a 2-vector of functions which both expect rank 0 arguments. This gives 2 as the frame for both the function and argument arrays. Now that the function and argument arrays have the same frame, each

$$\left([\texttt{curry-add}]_{\bullet} \begin{bmatrix}1\\2\end{bmatrix}_2\right) \begin{bmatrix}20\\30\end{bmatrix}_2 \mapsto \begin{bmatrix}(\texttt{curry-add}\ 1)\\(\texttt{curry-add}\ 2)\end{bmatrix}_2 \begin{bmatrix}20\\30\end{bmatrix}_2 \mapsto \begin{bmatrix}((\texttt{curry-add}\ 1)\ 20)\\((\texttt{curry-add}\ 2)\ 30)\end{bmatrix}_2$$

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

Fig. 2. Lifting the implicit `apply`

function in the array is applied to corresponding cells in the argument arrays. We then have $[((\texttt{curry-add}\ 1)\ 20),\ ((\texttt{curry-add}\ 2)\ 30)]_2$.

The generalized lifting rule provides a way to express a kind of MIMD computation not expressible in APL: the program can dynamically construct and apply an array of distinct functions. In computing a vector mean, we require both the `sum` and the `length`. We can apply $[\texttt{sum}, \texttt{length}]_2$ to a vector, $[8, 9, 6]_3$. The functions consume vectors, so there is only one argument cell. Duplicating this cell transforms the argument vector into a matrix, $[8, 9, 6, 8, 9, 6]_{2,3}$. Pointwise application then produces a vector of applications, $[(\texttt{sum}\ [8, 9, 6]_3),\ (\texttt{length}\ [8, 9, 6]_3)]_2$.

3.1 Syntax

$$\begin{array}{lcll}
e & ::= & \alpha \mid x \mid (e\ e\ \ldots) \mid (\texttt{unbox}\ (x = e)\ e) & \textit{(exressions)} \\
\alpha & ::= & [l\ \ldots]_{n\ \ldots} \mid (\texttt{box}\ e) & \textit{(arrays)} \\
l & ::= & b \mid f \mid e & \textit{(array elements)} \\
b & & \text{base values} & \\
f & ::= & \pi \mid (\lambda\ [(x\ \rho)\ \ldots]\ e) & \textit{(functions)} \\
\pi & & \text{primitive operators} & \\
\rho & ::= & z \mid \infty & \textit{(argument ranks)} \\
z & \in & \mathbb{Z} \qquad n, m \in \mathbb{N} & \textit{(numbers)} \\
v & ::= & b \mid f \mid [b\ \ldots]_{n\ \ldots} \mid [f\ \ldots]_{n\ \ldots} \mid (\texttt{box}\ v) \mid [(\texttt{box}\ v)\ \ldots]_{m,n\ \ldots} & \textit{(value forms)} \\
E & ::= & \square \mid (v\ \ldots\ E\ e\ \ldots) \mid [v\ \ldots\ E\ l\ \ldots]_{n\ \ldots} \mid (\texttt{box}\ E) & \textit{(evaluation contexts)} \\
 & \mid & (\texttt{unbox}\ (x = E)\ e) &
\end{array}$$

Fig. 3. Syntax, value domain and evaluation contexts of the untyped array language

Figure 3 presents the syntax and semantic domains for our untyped array language. We use $t\ \ldots$ to denote a possibly empty sequence, t_1 through t_k. Thus $t\ t'\ \ldots$ represents a guaranteed-nonempty sequence. We may also use $f(t)\ \ldots$ to represent $f(t_1)$ through $f(t_k)$. Expressions include arrays, variables, application forms, and a `let`-like form for extracting the contents of a box. An array is either a sequence of elements tagged with a sequence of naturals representing its shape or a box containing any expression. Array elements are a broader syntactic class than expressions, including base values (noted as b) and functions. Arrays are allowed to syntactically contain sub-arrays; nested arrays are reduced to non-nested arrays during evaluation. λ-abstractions can only be applied to arrays, so variables can only represent arrays. A function is either a primitive operator (noted as π) or a λ-abstraction.

The value forms are arrays with all elements fully evaluated. This allows them to contain base values or functions but not application forms or variables. A box is a value as long as it has a value for its contents. An array of box values is also a value as long as the array is not itself a scalar (*i.e.*, its shape vector must be nonempty). A scalar array containing a box reduces to the box itself.

The built-in operators include conventional scalar operations, such as `+`, `sqrt`, `AND`, *etc.* These all expect their arguments to have rank 0. The common list operations—`head`, `tail`, `init`, `last`, and `append`—have argument rank ∞ so that they can be used to build and destructure arrays of any rank (by reranking at finite argument rank). The operations for manipulating the iteration space described earlier (`prefix`, `reduce`, *etc.*) have argument rank ∞ for both the function and data arrays they consume, and they can be reranked to any natural or negative rank.

3.2 Semantics

Figure 4 gives the operational semantics, and figure 5 defines metafunctions used by the semantics.

The β rule (analogous to β-reduction in the call-by-value λ-calculus) requires that the function's argument ranks match the ranks of the arrays being passed to it. Similarly, the δ rule applies a scalar containing a built-in operator to arguments which have the operator's expected argument ranks.

The *nat*, *lift*, and *map* rules form the steps involved in lifting function application for function and argument arrays of higher rank. The *nat* rule is used in cases where some functions in an application form have infinite or negative argument rank. Primitives are tagged with the appropriate natural argument ranks so that subsequent uses of *Argrank* $[\![\cdot]\!]$ on this occurrence of the primitive will recognize it as having the natural rank it takes on for this particular application.

The *lift* rule expands the function and argument arrays into the application frame by repeating their cells. In cases where function and argument arrays' frames are not all prefixes of a single frame, we have a shape mismatch—function application cannot proceed, so evaluation is stuck (this would raise a "length error" in J).

After an application has been naturalized and lifted, the *map* rule converts function application in which the function and argument arrays are all over-ranked by the same amount to an array of function applications. In the resulting array, each application will have a scalar in function position, and all arguments will have that function's expected rank. We apply *Cells* to each argument array to produce a list of lists of cells. Transposing the nested list produces a nested list where the first entry contains all of the arguments' first cells, the second entry contains all of the arguments' second cells, and so on. Each of these lists is used as the arguments for the corresponding cell (*i.e.*, single function) of the function array. The reduction step produces an array of application forms whose shape is the frame of the original application form.

After the application forms generated by *map* reduction have been evaluated, we have an array of arrays. The *collapse* rules transform a nested array into a non-nested array. If the inner arrays' shapes differ, we have a shape mismatch, and evaluation is stuck (this would induce J's "filling" behavior mentioned in 2.2, potentially causing unexpected results). For *collapse*$_1$, the resulting array contains the concatenated atoms of the inner arrays. Its shape results from prepending the shape of the outer array onto the shape of the inner arrays. In the case of a scalar array containing a box, *collapse*$_2$ reduces to just the box.

Once a box's contents are evaluated, the *unbox* rule substitutes that value into another expression. A function with an `unbox` form in its body can be used to post-process another operation's result cells to make sure their shapes match.

The Empty-Frame Dilemma We require separate rules, *lift*$_0$ and *map*$_0$, for cases where an application form's principal frame shape contains one or more zeroes. Such a frame contains *no cells*, so the lifted function is not applied at all. With no cells to generate, the result is an empty array, but there is no clear way to choose the shape of the result array. That is, both a $2 \times 0 \times 7 \times 24$ array and a $2 \times 0 \times 365$ array are empty arrays—they both have no elements. But they are not at all the same array. If we are lifting a function across a 2×0 frame of argument cells, how can we determine the shape of the result cells? The resulting array's shape must at least start with the principal frame. The rest of the shape is left to a nondeterministic choice, but a language may choose to make a stronger guarantee about how $m \ldots$ will be chosen.

For example, in J, when a function is lifted to apply over an empty frame, it is probed (at run time) by applying it to a cell whose atoms are all 0 or the space character `' '` to determine the result cell shape (the cell itself is then discarded). Unfortunately, this is not safe with an effectful function or one whose result shapes are input-dependent, and it relies on having a bounded number of data types. It is one of J's more awkward corner cases, one that we will be able to resolve cleanly by means of the type system developed in the next section.

Another option is to always consider the resulting cell shape to be scalar unless some concrete cells are available to show otherwise. Lifted functions are often functions on scalars, and this allows scalar operations to behave as expected on empty arrays. The reduction rules could also be changed to make applying in an empty frame a dynamic error.

3.3 Sample Code

We present here several examples of code in our untyped language. As noted earlier, it is intended as a core, not surface, language.

A well-known case of manipulating the iteration space is `sum`:

$$(\lambda\ [(\mathtt{xs}\ 1)]\ ([\mathtt{reduce}]_{\bullet}\ [+]_{\bullet}\ ([\mathtt{append}]_{\bullet}\ [0]_1\ \mathtt{xs})))$$

Applying term abstraction:
$\big([(\lambda\ [(x\ n)\ \ldots]\ \ e)]_{\bullet}\ \ v\ \ldots\big)$
$\mapsto_{\beta}\ \ e\,[(x \leftarrow v)\ \ldots]$
where $n_j = \mathit{Rank}\,\llbracket v_j \rrbracket$, for each j

Applying primitive operator:
$\big([\pi]_{\bullet}\ \ v\ \ldots\big)$
$\mapsto_{\delta}\ \ \delta(\pi, v\ \ldots)$
where $\langle n\ \ldots\rangle = \mathit{Argrank}\,\llbracket \pi \rrbracket$
$n_j = \mathit{Rank}\,\llbracket v_j \rrbracket$, for each j

Rewriting with natural argument ranks:
$([f\ \ldots]_{n\,\ldots}\ \ v\ \ldots)$
$\mapsto_{nat}\ \ ([f'\ \ldots]_{n\,\ldots}\ \ v\ \ldots)$
where $\mathit{Argrank}\,\llbracket f_j \rrbracket \notin \mathbb{N}^k$ for some j
$f' = \mathit{Naturalize}\,\llbracket f, v\ \ldots \rrbracket$

Pointwise application:
$([f\ \ldots]_{n\,\ldots}\ \ v\ \ldots)$
$\mapsto_{map}\ \ \big[([f]_{\bullet}\ \ \alpha\ \ldots)\ \ldots\big]_{n\,\ldots}$
where $f\ \ldots$ is a nonempty sequence
$\langle n\ \ldots\rangle = \mathit{Argrank}\,\llbracket f_j \rrbracket$, for each j
$0 < k = \mathit{Rank}\,\llbracket v_j \rrbracket - n_j$, for each j
$((\alpha\ \ldots)\ \ldots) = (\mathit{Cells}_n\,\llbracket v \rrbracket\ \ldots)^{\top}$

Empty frame:
$([f\ \ldots]_{n\,\ldots}\ \ v\ \ldots)$
$\mapsto_{\mathit{lift}_0}\ \ [\ ]_{n'\,\ldots\ m\,\ldots}$
where $\langle \rho\ \ldots\rangle = \mathit{Argrank}\,\llbracket f_j \rrbracket$, for each j
$\rho_j \in \mathbb{N}$ for each j
$0 \in n'\ \ldots = \mathit{Max}\llbracket n\ \ldots,$
$\mathit{Frame}_{\rho}\,\llbracket v \rrbracket,\ \ldots \rrbracket$
$\mathit{Rank}\,\llbracket v_j \rrbracket - \rho_j$ not same for all j
$m\ \ldots$ chosen nondeterministically

Empty function:
$([\ ]_{n\,\ldots}\ \ v\ \ldots)$
$\mapsto_{\mathit{map}_0}\ \ []_{n\,\ldots\ \ m\,\ldots}$
where $m\ \ldots$ chosen nondeterministically

Converting nested to non-nested:
$[\alpha\ \ldots]_{n\,\ldots}$
$\mapsto_{\mathit{collapse}_1}\ \ [\mathit{Atoms}\,\llbracket \alpha \rrbracket\ \ldots]_{n\,\ldots\ \mathit{Shape}\llbracket \alpha \rrbracket}$
where no α contains a var or app form
no α is a box
all α have the same shape

Converting scalar of boxes to box:
$[\texttt{box}\ v]_{\bullet}\ \ \mapsto_{\mathit{collapse}_2}\ \ \texttt{box}\ v$

Extracting the contents of a box:
$(\texttt{unbox}\ x = (\texttt{box}\ v)\ \ e)\ \ \mapsto_{unbox}\ \ e[x\ \leftarrow\ v]$

Duplicating cells:
$([f\ \ldots]_{n\,\ldots}\ \ v\ \ldots)$
$\mapsto_{\mathit{lift}}\ \ \big(\mathit{Dup}_{0,n'\,\ldots}\ \big\llbracket [f\ \ldots]_{n\,\ldots} \big\rrbracket\ \ \mathit{Dup}_{\rho,n'\,\ldots\ m'\,\ldots}\ \llbracket v \rrbracket\ \ldots\big)$
where $\langle \rho\ \ldots\rangle = \mathit{Argrank}\,\llbracket f_j \rrbracket$, for each j
$\rho_j \in \mathbb{N}$ for each j
$0 \notin n'\ \ldots = \mathit{Max}\,\llbracket n\ \ldots, \mathit{Frame}_{\rho}\,\llbracket v \rrbracket\ \ldots \rrbracket$
the ρ_j-cells of v_j have shape $m'\ \ldots$
$\mathit{Rank}\,\llbracket v_j \rrbracket - \rho_j$ is not the same for all j

Fig. 4. Small-step operational semantics for an untyped array language

$$\mathit{Rank} : \mathit{Val} \rightharpoonup \mathbb{N}$$
$$\mathit{Rank}\,\llbracket [l\ \ldots]_{n\,\ldots} \rrbracket = \mathit{length}(n\ \ldots)$$

$$\mathit{Argrank} : \mathit{Fun} \rightarrow \mathit{Rank}^*$$
$$\mathit{Argrank}\,\llbracket (\lambda\ [(x\ \rho)\ \ldots]\ e) \rrbracket = \rho\ \ldots$$

$$\mathit{Naturalize} : \mathit{Fun} \times \mathit{Val}^* \rightharpoonup \mathit{Fun}$$
$$\begin{array}{l} \mathit{Naturalize}\,\llbracket (\lambda\ [(x\ \rho)\ \ldots]\ e), v\ \ldots \rrbracket \\ \ = (\lambda\ [(x\ n)\ \ldots]\ e) \\ \text{where } n_i = \rho_i \text{ if } \rho_i \in \mathbb{N} \\ \qquad\quad n_i = \mathit{Rank}\,\llbracket v_i \rrbracket + \rho_i \text{ if } -\rho_i \in \mathbb{N} \\ \qquad\quad n_i = \mathit{Rank}\,\llbracket v_i \rrbracket \text{ if } \rho_i = \infty \end{array}$$

$$\mathit{Frame} : \mathit{Rank} \times \mathit{Val} \rightharpoonup \mathbb{N}^*$$
$$\begin{array}{l} \mathit{Frame}_\rho\,\llbracket [l\ \ldots]_{m\,\ldots\ n\,\ldots} \rrbracket = (m\ \ldots) \\ \text{where } \mathit{length}(n\ \ldots) = \rho \end{array}$$

$$\mathit{Max} : \mathbb{N}^{**} \rightharpoonup \mathbb{N}^*$$
$$\begin{array}{l} \mathit{Max}\,\llbracket (n\ \ldots) \rrbracket = n\ \ldots \\ \mathit{Max}\,\llbracket (n_0\ \ldots), (n_1\ \ldots)\ \ldots, (n_m\ \ldots) \rrbracket \\ \ = (n_0\ \ldots) \\ \text{if } \mathit{Max}\,\llbracket (n_1\ \ldots)\ \ldots, (n_m\ \ldots) \rrbracket \sqsubseteq (n_0\ \ldots) \\ \ = \mathit{Max}\,\llbracket (n_1\ \ldots)\ \ldots, (n_m\ \ldots) \rrbracket \\ \text{if } (n_0\ \ldots) \sqsubseteq (n_1\ \ldots) \end{array}$$

$$\mathit{Dup} : \mathit{Rank} \times \mathbb{N}^* \times \mathit{Val} \rightarrow \mathit{Val}$$
$$\begin{array}{l} \mathit{Dup}_{\rho,\ n\,\ldots\ m\,\ldots}\,\llbracket [l\ \ldots]_{d\,\ldots} \rrbracket \\ \ = [(l'\ \ldots)^k\ \ldots]_{n\,\ldots\ m\,\ldots} \\ \text{where } \mathit{length}(m\ \ldots) = \rho \\ \qquad\quad k = \prod_{j=1}^{\rho} n_j \\ \qquad\quad ((l'\ \ldots)\ \ldots) = \mathit{Cells}_\rho\,\llbracket [l\ \ldots]_{d\,\ldots} \rrbracket \end{array}$$

$$\mathit{Cells} : \mathbb{N} \times \mathit{Val} \rightharpoonup \mathit{Val}^*$$
$$\begin{array}{l} \mathit{Cells}_n\,\llbracket [l_1\ \ldots\ l_m\ l_{m+1}\ \ldots\ l_{2m}\ \ldots l_{p-m+1}\ \ldots\ l_p]_{c\,\ldots\ d\,\ldots} \rrbracket \\ \ = [l_1\ \ldots\ l_m]_{d\,\ldots}\ [l_{m+1}\ \ldots\ l_{2m}]_{d\,\ldots}\ \ldots [l_{p-m+1}\ \ldots\ l_p]_{d\,\ldots}, \\ \text{where } \mathit{length}(d\ \ldots) = n \\ \qquad\quad \prod_{i=1}^{n}(d_i) = m \end{array}$$

Fig. 5. Metafunctions used in array semantics

We can take advantage of automatic lifting for a simple `dotprod` operator:

$$(\lambda\ [(\texttt{xs}\ 1)\ (\texttt{ys}\ 1)]\ ([\texttt{sum}]_\bullet\ ([*]_\bullet\ \texttt{xs}\ \texttt{ys})))$$

We can convolve a signal with a filter by using `dotprod` with the reverse of one argument in a sliding `window` over the other:

$$\begin{array}{l} (\lambda[(\texttt{filter}\ 1)\ (\texttt{signal}\ 1)] \\ \quad ([\texttt{window}]_\bullet\ ([\texttt{length}]_\bullet\ \texttt{filter}) \\ \qquad\qquad\quad [(\lambda\ [(\texttt{seg}\ 1)]\ ([\texttt{dotprod}]_\bullet\ \texttt{seg}\ ([\texttt{reverse}]_\bullet\ \texttt{filter})))]_\bullet\ \texttt{signal})) \end{array}$$

Iverson included many composition forms and operators. However, λ allows the programmer or library implementor to define them. A simple `compose` operator for two unary functions can be defined as:

$$(\lambda\ [(\texttt{f}\ 0)(\texttt{g}\ 0)]\ [(\lambda\ [(\texttt{x}\ \infty)]\ (\texttt{f}\ (\texttt{g}\ \texttt{x})))]_\bullet)$$

J's `fork` form applies two functions (referred to as "tines") to the same input and then applies a third function to their results:

$$(\lambda\ [(\texttt{f}\ 0)(\texttt{g}\ 0)(\texttt{h}\ 0)]\ ([(\lambda\ [(\texttt{x}\ \infty)]\ (\texttt{f}\ (\texttt{g}\ \texttt{x})\ (\texttt{h}\ \texttt{x})))]_\bullet)$$

A simple use of `fork` is computing the arithmetic `mean`:

$$(\lambda\ [(\texttt{xs}\ 1)]\ (([\texttt{fork}]_\bullet\ [/]_\bullet\ [\texttt{sum}]_\bullet\ [\texttt{length}]_\bullet)\ \texttt{xs})$$

The `fork` divides the sum of its input by its length. The outer λ modifies the argument rank of the resulting function, so the function produced by `fork` is only applied to lists.

J also uses a `hook` form (based on the **S** combinator) for applying a binary function to an argument and a transformed version of that same argument.

$$(\lambda\ [(\texttt{f}\ 0)(\texttt{g}\ 0)]\ ([(\lambda[(\texttt{x}\ \infty)]\ (\texttt{f x}\ (\texttt{g x})))]_{\bullet})$$

Without a general recursion operator, `iota` can be used as a limited form of the classical `unfold`, allowing primitive recursion. Using `iota` to write `factorial`:

$$\begin{array}{l}
(\lambda\ [(\texttt{n}\ 0)] \\
\quad (\texttt{unbox}\ (\texttt{xs} = ([\texttt{iota}]_{\bullet}\ [\texttt{n}]_{1})) \\
\quad\quad ([\texttt{reduce}]_{\bullet}\ [*]_{\bullet}\ ([+]_{\bullet}\ [1]_{\bullet}\ (\texttt{append}\ [0]_{1}\ \texttt{xs})))))
\end{array}$$

First, the input scalar is wrapped in a singleton vector and passed to `iota` to produce a boxed vector containing $[0, \ldots, \texttt{n} - 1]$. If $n = 0$, this vector is empty, and later operations would have an empty frame, so we append 0. We then add 1 to get a vector containing $[1, 1, \ldots, \texttt{n}]$. Reducing by $*$ gives $\texttt{n}!$.

We can use `iota` to evaluate a polynomial at a particular point, which uses arguments of differing rank:

$$\begin{array}{l}
(\lambda\ [(\texttt{coeffs}\ 1)\ (\texttt{x}\ 0)] \\
\quad (\texttt{unbox}\ (\texttt{i} = ([\texttt{iota}]_{\bullet}\ ([\texttt{length}]_{\bullet}\ \texttt{coeffs}))) \\
\quad\quad ([\texttt{reduce}]_{\bullet}\ [+]_{\bullet}\ ([*]_{\bullet}\ \texttt{coeffs}([\hat{\ }]_{\bullet}\ \texttt{x i})))))
\end{array}$$

We can also construct an iteration space with `reshape`, which is convenient if we only need a single atom duplicated many times. The following `repeat` operator uses `compose` iterated over a vector containing a single duplicated atom to produce a function which applies that atom a given number of times.

$$\begin{array}{l}
(\lambda\ [(\texttt{f}\ \infty)\ (\texttt{n}\ 0)] \\
\quad (\texttt{unbox}\ (\texttt{fs} = ([\texttt{reshape}]_{\bullet}\ [\texttt{n}]_{1}\ \texttt{f})) \\
\quad\quad ([\texttt{reduce}]_{\bullet}\ [\texttt{compose}]_{\bullet}\ ([\texttt{append}]_{\bullet}\ [\texttt{id}]_{\bullet}\ \texttt{fs}))))
\end{array}$$

Bounded looping with `repeat` can be used for finding the transitive closure of an adjacency matrix. This example uses two additional functions which can be defined in terms of λ. The `dup` function transforms a binary function into a unary one which duplicates its argument and passes two copies to the underlying binary function. We also use `compose'`, a variation on the `compose` function defined above which produces a binary function, passing two arguments of ranks 1 and ∞ to its second input function and the result to its first input function.

$$\begin{array}{l}
(\lambda\ [(\texttt{adj}\ 2)] \\
\quad ((\texttt{repeat}\ (\texttt{hook or}\ (\texttt{dup}\ ([\texttt{compose'}]_{\bullet} \\
\qquad\qquad\qquad\qquad\qquad\qquad [(\lambda\ [(\texttt{xs}\ \infty)]\ (\texttt{reduce or true xs}))]_{\bullet} \\
\qquad\qquad\qquad\qquad\qquad\qquad [(\lambda\ [(\texttt{x}\ 1)\ (\texttt{y}\ \infty)]\ (\texttt{and x y}))]_{\bullet}))) \\
\quad\quad (\texttt{lg}\ (\texttt{length adj})))\ \texttt{adj}))
\end{array}$$

The function constructed by `compose'` applies `and` to each row of its first argument (this treats it as a column) and its entire second argument. The result is a rank 3 array whose matrices are combined using `or` to produce a matrix analogous to the matrix product of the original two arguments. Wrapping this function with `dup` creates a unary function which transforms a matrix into its "boolean product" with itself. The `hook` of `or` and this adjacency matrix transformation is a function which updates an adjacency matrix to allow paths twice as long. Finally, this process is `repeated` (`lg` (`length adj`)) $= log_2(|V|)$ times.

4 Types for Array-Oriented Programming

In order to eliminate shape-mismatch errors, our type system must be capable of tracking arrays' shapes. Dependent typing has been used in the past to implement lists whose types specify their lengths via a natural number index. This generalizes to an array type which is indexed by a list of natural numbers to specify its shape. If types can contain arbitrary term expressions, checking whether two types are equivalent can require checking whether two terms are equivalent. In order to keep type checking tractable, we use the technique of defining a separate language of type indices, demonstrated by Xi *et al.* in Dependent ML [18]. Separating the term and index languages eliminates the mutual dependence between type checking and evaluation. An index language should be powerful enough to express the desired type properties, but also simple enough that checking index equivalence is tractable. In Dependent ML's case, index equivalence is checked via integer linear programming. The constraint domain associated with our index language also includes lists of natural numbers; this combination of theories is still decidable [12].

4.1 Syntax

Figure 6 gives the syntax for Remora. It includes several new expression and element forms. They are introduction and elimination forms for universal types (Tλ and `T-APP`), dependent products (Iλ and `I-APP`), and dependent sums (`PACK` and `UNPACK`). Dependent sums effectively replace boxes from the untyped language. A type or index abstraction or application form can be used as an element, and it is a valid expressions as long as its underlying element is also a valid expression. Multiple type or index abstraction forms in an array can each be given separate type or index arguments to produce functions of the same type. Remora's arrays can have a type annotation rather than just a shape annotation. This ensures that a concrete type can be determined for an empty array. For non-empty arrays (those of the form $[l\ l' \ldots]$), a shape annotation is sufficient, and the type can be reconstructed by inspecting the array elements. It is assumed that similar type annotations for all expression forms will be generated in type checking, but these are not included in the regular program syntax.

Types include base types such as `Num` or `Bool` (noted as B) and arrays of a given shape and element type (noted as $\mathtt{A}_\iota\tau$). An index can be a `Nat` (*n.b.*, different from `Num`), a `Shape` (noted as (S $\iota \ldots$)), or the sum of two indices.

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

Fig. 6. Syntax for Remora

4.2 Static Semantics

The typing, kinding, and sorting rules are given in Figures 7 and 8. Types are ascribed to elements (which can themselves be arrays). Rules for base types are straightforward, but an example rule for numbers is given in Figure 7.

The kind judgment is simply a well-formedness check—all well-formed types are of a single kind. K-Array accepts an array type as well-formed if its underlying type is well formed and its index is a `Shape`. K-Univ binds type variables, and K-DProd and K-DSum bind index variables at specific sorts. A variable introduced in a universal type is only allowed to stand for a non-array type. This is necessary in order to express polymorphic input types like "any scalar," $\texttt{A}_{(\texttt{S})}\texttt{t}$ (with `t` bound by some ∀). Otherwise, $\texttt{A}_{(\texttt{S})}\texttt{t}$ could describe any array type.

S-Shape requires that a shape be built from `Nat`s. Constructing an index with + requires that the summands be `Nat`s, and the result will also be a `Nat`.

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 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 7. Type judgment for Remora

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

$$\frac{}{\Delta;\Theta \vdash B}\ (\text{K-Base}) \qquad \frac{x \in \Delta}{\Delta;\Theta \vdash x}\ (\text{K-Var}) \qquad \frac{\begin{array}{c}\Delta;\Theta \vdash \tau \\ \Theta \vdash \iota :: \texttt{Shape}\end{array}}{\Delta;\Theta \vdash \texttt{A}_{\iota}\tau}\ (\text{K-Array})$$

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

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

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

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

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

Fig. 8. Kind and index sort judgments for Remora

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

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

In the untyped language, `append` has argument rank ∞, but it still requires its arguments to have the same shape except for their first dimensions. 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.

4.3 Dynamic Semantics

The reduction relation is given in Figure 9. It assumes every expression has been annotated with its type (most of these type annotations can be generated mechanically). This run time type information is needed to determine the correct output cell shape for a function application with an empty frame, so type annotations are kept up to date during reduction (they subsume the untyped language's shape tags). We use $x[(y \leftarrow_e z)\ \ldots]$, $x[(y \leftarrow_t z)\ \ldots]$, and $x[(y \leftarrow_i z)\ \ldots]$ for substitution of term, type, and index variables respectively. The untyped language's box and nonscalar array of boxes value forms are replaced with analogous sum and nonscalar array of sums. We replace the evaluation contexts for `box` and `unbox` with analogous contexts for `PACK` and `UNPACK`.

Remora's β, δ, and *collapse* rules are essentially unchanged from the untyped language, so they are not repeated. The implicit lifting is now type-directed, instead of rank-directed. Types include enough information to determine the correct cell shape for any application form, solving the empty-frame dilemma from 3.2 and eliminating the nondeterminism.

Pointwise application:

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

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

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

$$((\alpha\ \ldots)\ \ldots) = ((\mathit{Cells}_\rho\,\llbracket v \rrbracket)\ \ldots)^{\top}$$

Duplicating cells:

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

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

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

$$\iota = \mathit{Max}\,\llbracket (m\ \ldots), (m'\ \ldots)\ \ldots \rrbracket$$

Applying a type abstraction:

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

Applying an index abstraction:

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

Projecting from a dependent sum:

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

Fig. 9. Small-step operational semantics for Remora

$T\beta$ and $I\beta$ substitute types and indices for the appropriate type and index variables. This substitution must be applied to both the body of the type or index abstraction as well as to its type annotation. Explicit type and index application effectively replace *naturalize* steps from the untyped language. Finally, *project* substitutes a dependent sum's witnesses and contents in the body expression.

The sample programs given in section 3.3 are straightforward to express in Remora. Typed code is given in Appendix B. The translation involves adding type and index abstractions and applications and replacing rank annotations with type annotations.

4.4 Type Soundness

We expect a type system which ascribes shapes to arrays to only ascribe shapes that the arrays will actually have once computed.

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

- *There is some* v *such that* $l \mapsto^* v$
- l *diverges*
- *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.

5 Future Work

The transition from a core semantics modeled in PLT Redex to a complete programming system requires a more flexible surface language and a compiler. In moving from the untyped core language to Remora, the added code is mostly type and index applications. Type inference would be necessary in order to make a surface language based on Remora practical. An interesting challenge in this setting is that the different type and index arguments can produce different behavior (*e.g.*, reducing an entire matrix versus reducing its 1-cells).

An implementation of Remora could use type information to inform decisions about how to parallelize aggregate operations. With a cost model for analyzing when different cells in an application frame are likely to take significantly different amounts of time, a compiler could choose between statically breaking up a task and leaving the allocation to a work-stealing run-time system.

Stream-like computation is often convenient for tasks such as signal processing, and it could be expressed by generalizing array types to allow an unbounded dimension. Implicit lifting still has a sensible meaning, as do `foldl`, `scan`, and `window`. This would allow us to extend Iverson's rank-polymorphic control mechanism to Turing-equivalent programs requiring `while`-loop computation (for example, iterating a numeric solver to a given tolerance).

6 Conclusion

We have given a formal reduction semantics for Iverson's rank polymorphism which addresses several shortcomings of the model. Remora generalizes automatic operator lifting to include first-class functions and MIMD computation. Embedding the core ideas of APL and J in a setting based on λ-calculus combines the expressive power of both models. Our type system rules out errors due to mismatching argument shapes and still gives the programmer enough freedom to write code whose result shape cannot be determined until run time.

References

1. Backus, J.: Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. Commun. ACM 21(8), 613–641 (Aug 1978)
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. Brooks, F.P.: The Design of Design: Essays from a Computer Scientist. Addison-Wesley (2010)
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. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, 1st edn. (2009)
7. Iverson, K.E.: A programming language. John Wiley & Sons, Inc., New York, NY, USA (1962)
8. Jay, C.B.: The fish language definition. Tech. rep. (1998)
9. 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)
10. Jsoftware, Inc.: Jsoftware: High-performance development platform, http://www.jsoftware.com/
11. 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)
12. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (Oct 1979)
13. Peyton Jones, S., Leshchinskiy, R., Keller, G., Chakravarty, M.M.: Harnessing the multicores: Nested data parallelism in haskell. In: FSTTCS. vol. 2, pp. 383–414 (2008)
14. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. 31(4), 32:1–32:12 (Jul 2012)
15. Scholz, S.B.: Single assignment c: efficient support for high-level array operations in a functional setting. J. Funct. Program. 13(6), 1005–1059 (Nov 2003)
16. Thatte, S.: A type system for implicit scaling. Sci. Comput. Program. 17(1-3), 217–245 (Dec 1991), http://dx.doi.org/10.1016/0167-6423(91)90040-5
17. Trojahner, K., Grelck, C.: Dependently typed array programs dont go wrong. Journal of Logic and Algebraic Programming 78(7), 643–664 (2009)
18. Xi, H.: Dependent types in practical programming. Ph.D. thesis, Pittsburgh, PA, USA (1998), aAI9918624
19. 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)
20. Zima, H., Chapman, B.: Supercompilers for Parallel and Vector Computers. ACM Press (1990)

Appendix

A Basis Library

The following table lists primitive operators along with their argument ranks (for the untyped language) and types (for Remora). Free type and index variables should be considered as bound by universals and dependent products. For example, the full type of `length` is:

$$\left(\forall\,[\mathtt{t}]\;\left(\mathit{\Pi}\;[(\mathtt{n\ Nat})\;(\mathtt{d\ Shape})]\;\left(\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\right)\right)\right)$$

From `copy` down, all operators produce boxes in the untyped language and dependent sums in Remora. The types of `prefix`, `suffix`, and `window` use rank-2 polymorphism. The function passed to them must be parameterized over its input length in order to be safe to use. Using a family of `window` functions make it safe to use a function which is monomorphic in its argument length, but a single use of `prefix` and `suffix` applies a function to arguments of several lengths.

Function	**Arg. Rank**	**Type**
+, -, *, /, ^, `mod`, `and`, `or`	$0, 0$	$\left(\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\ \ \mathtt{A}_{(\mathtt{S})}\mathtt{Num}\to\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\right)$
`square`, `sqrt`, `exp`, `log`, `not`	0	$\left(\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\to\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\right)$
`head`, `last`	∞	$\left(\mathtt{A}_{(\mathtt{S\ (+\ 1\ n)})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`tail`, `init`	∞	$\left(\mathtt{A}_{(\mathtt{S\ (+\ 1\ n)})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`append`	∞	$\left(\mathtt{A}_{(\mathtt{S\ m})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S\ (+\ m\ n)})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`reverse`	∞	$\left(\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`rotate`	$0, \infty$	$\left(\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\ \ \mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`length`	∞	$\left(\mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\right)$
`foldl`	$0, \infty, \infty$	$\left(\mathtt{A}_{(\mathtt{S})}\ (\mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{\mathtt{e}}\mathtt{u}\to\mathtt{A}_{\mathtt{d}}\mathtt{t})\ \ \mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{e}}\mathtt{u}\to\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`foldr`	$0, \infty, \infty$	$\left(\mathtt{A}_{(\mathtt{S})}\ (\mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{\mathtt{e}}\mathtt{u}\to\mathtt{A}_{\mathtt{e}}\mathtt{u})\ \ \mathtt{A}_{\mathtt{e}}\mathtt{u}\ \ \mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{\mathtt{e}}\mathtt{u}\right)$
`reduce`	$0, \infty$	$\left(\mathtt{A}_{(\mathtt{S})}\ (\mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{\mathtt{d}}\mathtt{t})\ \ \mathtt{A}_{(\mathtt{S\ n})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{\mathtt{d}}\mathtt{t}\right)$
`scan`	$0, \infty$	$\left(\begin{array}{l}\mathtt{A}_{(\mathtt{S})}\ (\mathtt{A}_{\mathtt{d}}\mathtt{t}\ \ \mathtt{A}_{\mathtt{d}}\mathtt{t}\to\mathtt{A}_{\mathtt{d}}\mathtt{t})\\ \mathtt{A}_{(\mathtt{S\ (+\ 1\ n)})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\\ \to\mathtt{A}_{(\mathtt{S\ (+\ 1\ n)})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\end{array}\right)$

Function	Arg. Rank	Type
`readvec`	*none*	$\big(\to \big(\mathit{\Sigma}\,[(\mathtt{m}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\big)\big)$
`copy`	$1, \infty$	$\big(\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\ \ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{A}_{\mathtt{d}}\mathtt{t} \to \big(\mathit{\Sigma}\,[(\mathtt{m}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{A}_{\mathtt{d}}\mathtt{t}\big)\big)$
`iota`	1	$\big(\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num} \to \big(\mathit{\Sigma}\,[(\mathsf{d}\ \mathsf{Shape})]\ \mathtt{A}_{\mathtt{d}}\mathtt{Num}\big)\big)$
`iota’`	0	$\big(\mathtt{A}_{(\mathtt{S})}\mathtt{Num} \to \big(\mathit{\Sigma}\,[(\mathtt{m}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{Num}\big)\big)$
`iota`k	1	$\big(\mathtt{A}_{(\mathtt{S}\ k)}\mathtt{Num} \to \big(\mathit{\Sigma}\,[(\mathtt{n}_1\ \mathtt{Nat}) \cdots (\mathtt{n}_k\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n}_1 \cdots \mathtt{n}_k)}\mathtt{Num}\big)\big)$
`reshape`	$1, \infty$	$\big(\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num}\ \ \mathtt{A}_{\mathtt{d}}\mathtt{t} \to \big(\mathit{\Sigma}\,[(\mathsf{e}\ \mathsf{Shape})]\ \mathtt{A}_{\mathtt{e}}\mathtt{t}\big)\big)$
`reshape’`	$0, \infty$	$\big(\mathtt{A}_{(\mathtt{S})}\mathtt{Num}\ \ \mathtt{A}_{\mathtt{d}}\mathtt{t} \to \big(\mathit{\Sigma}\,[(\mathtt{n}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{t}\big)\big)$
`reshape`k	$1, \infty$	$\left(\begin{array}{l}\mathtt{A}_{(\mathtt{S}\ k)}\mathtt{Num}\ \ \mathtt{A}_{d}t \\ \to \big(\mathit{\Sigma}\,[(\mathtt{n}_1\ \mathtt{Nat}) \cdots (\mathtt{n}_k\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n}_1 \cdots \mathtt{n}_k)}t\big)\end{array}\right)$
`prefix`, `suffix`	$0, \infty$	$\left(\begin{array}{l}\big(\forall\,[\mathtt{t}]\ \big(\mathit{\Pi}\,[(\mathtt{n}\ \mathtt{Nat})(\mathsf{c}\ \mathsf{Shape})]\ \big(\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{A}_{\mathtt{c}}\mathtt{t} \to \mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\big)\big) \\ \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{A}_{\mathtt{d}}\mathtt{v} \\ \to \big(\mathit{\Sigma}\,[(\mathtt{p}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{p})}\mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\end{array}\right)$
`window`	$0, 0, \infty$	$\left(\begin{array}{l}\big(\forall\,[\mathtt{t}]\ \big(\mathit{\Pi}\,[(\mathtt{n}\ \mathtt{Nat})(\mathsf{c}\ \mathsf{Shape})]\ \big(\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{A}_{\mathtt{c}}\mathtt{t} \to \mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\big)\big) \\ \mathtt{A}_{(\mathtt{S})}\mathtt{Num}\quad \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{A}_{\mathtt{d}}\mathtt{v} \\ \to \big(\mathit{\Sigma}\,[(\mathtt{p}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{p})}\mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\end{array}\right)$
`window`k	$0, \infty$	$\left(\begin{array}{l}\mathtt{A}_{(\mathtt{S})}\ \big(\mathtt{A}_{(\mathtt{S}\ k)}\mathtt{A}_{\mathtt{d}}\mathtt{t} \to \mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\quad \mathtt{A}_{(\mathtt{S}\ \mathtt{m})}\mathtt{A}_{\mathtt{d}}\mathtt{t} \\ \to \big(\mathit{\Sigma}\,[(\mathtt{n}\ \mathtt{Nat})]\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{A}_{\mathtt{e}}\mathtt{u}\big)\end{array}\right)$

B Typed Example Code

A typed version of `sum`:

$$\begin{array}{l}(\mathtt{I}\lambda\ [(\mathtt{n}\ \mathtt{Nat})]\ (\lambda\ [(\mathtt{xs}\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num})] \\ \qquad ([(\mathtt{I\text{-}APP}\ (\mathtt{T\text{-}APP}\ \mathtt{reduce}\ \mathtt{Num})\ \mathtt{n}\ \ (\mathtt{S}))]^{(\mathtt{S})} \\ \qquad\ \ [+]^{(\mathtt{S})}\ \ ([(\mathtt{I\text{-}APP}\ (\mathtt{T\text{-}APP}\ \mathtt{append}\ \mathtt{Num})\ 1\ \mathtt{n}\ \ (\mathtt{S}))]^{(\mathtt{S})}\ \ [0]^{(\mathtt{S}\ 1)}\ \mathtt{xs}))))\end{array}$$

First, `reduce` is type-applied to `Num` and index-applied to `n` and the scalar shape. It now requires a binary operator on scalar numbers and a list of $1 + \mathtt{n}$ scalar numbers. The binary operator is $+$. To construct a list of length $1 + \mathtt{n}$, we type-apply `append` to `Num` and then index-apply it to 1, `n`, and the scalar shape, meaning we intend to append two lists whose contents are scalar numbers and whose lengths are 1 and `n`. This means we must provide a list of 1 scalar number and a list of n scalar numbers. $[0]^{(\mathtt{S}\ 1)}$ and `xs` have types $\mathtt{A}_{(\mathtt{S}\ 1)}\mathtt{Num}$ and $\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num}$. These types are respectively equivalent to $\mathtt{A}_{(\mathtt{S}\ 1)}\mathtt{A}_{(\mathtt{S})}\mathtt{Num}$ and $\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{A}_{(\mathtt{S})}\mathtt{Num}$. Thus `append` will produce an array of type $\mathtt{A}_{(+\ 1\ \mathtt{n})}\mathtt{A}_{(\mathtt{S})}\mathtt{Num}$, as required by `reduce`. Finally, `reduce` produces an array of type $\mathtt{A}_{(\mathtt{S})}\mathtt{Num}$, *i.e.*, a scalar number, which is the sum of *xs*. This gives `sum` the type:

$$\big(\mathit{\Pi}\ [(\mathtt{n}\ \mathtt{Nat})]\ \big(\mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num} \to \mathtt{A}_{(\mathtt{S})}\mathtt{Num}\big)\big)$$

We can then use `sum` to write `dotprod`:

$$(\mathtt{I}\lambda\ [(\mathtt{n}\ \mathtt{Nat})]\ (\lambda\ [(\mathtt{xs}\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num})\ (\mathtt{ys}\ \mathtt{A}_{(\mathtt{S}\ \mathtt{n})}\mathtt{Num})]\ ([(\mathtt{I\text{-}APP}\ \mathtt{sum}\ 1)]^{(\mathtt{S})}([*]^{(\mathtt{S})}\mathtt{xs}\ \mathtt{ys}))))$$

The multiplication subexpression lifts $*$ into a frame with shape $(\texttt{S}\ l)$, so the result of the application is typed as $\texttt{A}_{(\texttt{S}\ l)}\texttt{Num}$. With `sum` index-applied to l, it consumes the $\texttt{A}_{(\texttt{S}\ l)}\texttt{Num}$ and produces a $\texttt{A}_{(\texttt{S})}\texttt{Num}$, the dot product of *xs* and *ys*.

Typing `compose` and `fork` just requires abstracting over the input, output, and intermediate shapes and atom types:

$$\begin{array}{l}
(\texttt{T}\lambda\ [\texttt{s}\ \texttt{t}\ \texttt{u}]\ (\texttt{I}\lambda\ [(c\ \texttt{Shape})\ (d\ \texttt{Shape})\ (e\ \texttt{Shape})] \\
\qquad (\lambda\ [(\texttt{f}\ \texttt{A}_{(\texttt{S})}\ (\texttt{A}_{\texttt{d}}\texttt{t} \rightarrow \texttt{A}_{\texttt{e}}\texttt{u})) \\
\qquad\qquad (\texttt{g}\ \texttt{A}_{(\texttt{S})}\ (\texttt{A}_{\texttt{c}}\texttt{s} \rightarrow \texttt{A}_{\texttt{d}}\texttt{t}))] \\
\qquad\qquad (\lambda\ [(\texttt{x}\ \texttt{A}_{\texttt{c}}\texttt{s})]\ (\texttt{f}\ (\texttt{g}\ \texttt{x})))))) \\
(\texttt{T}\lambda\ [\texttt{s}\ \texttt{t}\ \texttt{u}\ \texttt{v}]\ (\texttt{I}\lambda\ [(\texttt{a}\ \texttt{Shape})\ (\texttt{b}\ \texttt{Shape})\ (\texttt{c}\ \texttt{Shape})\ (\texttt{d}\ \texttt{Shape})] \\
\qquad (\lambda\ [(\texttt{f}\ \texttt{A}_{(\texttt{S})}\ (\texttt{A}_{\texttt{b}}\texttt{t}\ \texttt{A}_{\texttt{c}}\texttt{u} \rightarrow \texttt{A}_{\texttt{d}}\texttt{v})) \\
\qquad\quad (\texttt{g}\ \texttt{A}_{(\texttt{S})}\ (\texttt{A}_{\texttt{a}}\texttt{s} \rightarrow \texttt{A}_{\texttt{b}}\texttt{t})) \\
\qquad\quad (\texttt{h}\ \texttt{A}_{(\texttt{S})}\ (\texttt{A}_{\texttt{a}}\texttt{s} \rightarrow \texttt{A}_{\texttt{c}}\texttt{u}))] \\
\qquad\quad (\lambda\ [(\texttt{x}\ \texttt{A}_{\texttt{a}}\texttt{s})]\ (\texttt{f}\ (\texttt{g}\ \texttt{x})\ (\texttt{h}\ \texttt{x})))))))
\end{array}$$

With the right type and index arguments, we can write `mean`:

$$\begin{array}{l}
(\texttt{I}\lambda\ [(\texttt{n}\ \texttt{Nat})]\ (\lambda\ [(\texttt{xs}\ \texttt{A}_{(\texttt{S}\ \texttt{n})}\texttt{Num})] \\
\quad (((\texttt{I-App}\ (\texttt{T-APP}\ \texttt{fork}\ \texttt{Num}\ \texttt{Num}\ \texttt{Num}\ \texttt{Num})\ \ (\texttt{S}\ \texttt{n})\ (\texttt{S})\ (\texttt{S})\ (\texttt{S})) \\
\qquad\quad [/]^{(\texttt{S})}\ \ [(\texttt{I-APP}\ \texttt{sum}\ \texttt{n})]^{(\texttt{S})}\ \ [(\texttt{I-APP}\ \texttt{length}\ \texttt{n}\ \ (\texttt{S}))]^{(\texttt{S})})\ \texttt{xs})))
\end{array}$$

While the untyped `factorial` function transforms the result from `iota` to a scalar before allowing it to escape, `iota`'s range is too broad for a single `reduce` to be guaranteed to produce a scalar. This is evident from its type: `iota` returns a dependent sum that quantifies the entire shape rather than just one dimension. This is more general than needed in most situations. The length of the input to `iota` will be the rank of its result, and it is common to know statically how large a vector is being used. The same applies to `reshape`. Two simple changes can allow `factorial` to return a scalar instead of a dependent sum. An `iota'` function restricted to scalar input could have output type $\big(\mathit{\Sigma}\ [(\texttt{n}\ \texttt{Nat})]\ \texttt{A}_{(\texttt{S}\ \texttt{n})}\texttt{Num}\big)$, *i.e.*, a vector of unknown length. Alternatively, a family of `iota`-like functions each specialized for inputs of length k can return dependent sums with contents of rank k. In `factorial`, unfolding with `iota1` allows the program to proceed as in the untyped language:

$$\begin{array}{l}
(\lambda\ [(\texttt{n}\ \texttt{A}_{(\texttt{S})}\texttt{Num})] \\
\quad (\texttt{UNPACK}\ \langle \texttt{m}|\texttt{xs}\rangle = ([\texttt{iota1}]^{(\texttt{S})}\ \ [\texttt{n}]^{(\texttt{S}\ 1)}\ \ \texttt{xs}) \\
\qquad ([(\texttt{I-APP}\ (\texttt{T-APP}\ \texttt{reduce}\ \texttt{Num})\ \texttt{m}\ \ (\texttt{S}))]^{(\texttt{S})} \\
\qquad\ \ [*]^{(\texttt{S})}\ \ ([+]^{(\texttt{S})}\ [1]^{(\texttt{S})}\ \ ([(\texttt{I-APP}\ (\texttt{T-APP}\ \texttt{append}\ \texttt{Num})\ 1\ \texttt{m}\ \ (\texttt{S}))]^{(\texttt{S})} \\
\qquad\qquad\qquad [0]^{(\texttt{S})}\ \texttt{xs})))))
\end{array}$$

A version using `iota'` instead of `iota1` would be similar. The choice of a particular `iota`*k* can be made explicitly by the programmer or (in a surface language) through type reconstruction. An analogous use of `reshape1` to write `repeat`:

$$
\begin{array}{l}
(\texttt{T}\lambda\ [\texttt{t}]\ (\texttt{I}\lambda\ [(\texttt{d Shape})]\ (\lambda\ [(\texttt{f A}_{\texttt{(S)}}\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t}))\ (\texttt{n A}_{\texttt{(S)}}\texttt{Num})]\\
\quad (\texttt{UNPACK}\ \langle \texttt{m}|\texttt{xs}\rangle = ([(\texttt{I-APP}\ (\texttt{T-APP reshape1}\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{d}t))\ (\texttt{S}))]^{\texttt{(S)}}\ [\texttt{n}]^{\texttt{(S 1)}}\ \texttt{f})\\
\quad ([(\texttt{I-APP}\ (\texttt{T-APP reduce}\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t}))\ \texttt{m}\ (\texttt{S}))]^{\texttt{(S)}}\\
\qquad [(\texttt{I-APP}\ (\texttt{T-APP compose}\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t})\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t})\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t}))\\
\qquad\qquad (\texttt{S})\ (\texttt{S})\ (\texttt{S}))]^{\texttt{(S)}}\\
\qquad ([(\texttt{I-APP}\ (\texttt{T-APP append}\ (\texttt{A}_{\texttt{d}}\texttt{t} \to \texttt{A}_{\texttt{d}}\texttt{t}))\ (\texttt{S 1})\ \texttt{m}\ (\texttt{S}))]^{\texttt{(S)}}\\
\qquad\ [(\texttt{I-APP}\ (\texttt{T-APP id t})\ \texttt{d})]^{\texttt{(S)}}\ \texttt{xs}))))))
\end{array}
$$

C Type Soundness

We show soundness via progress and preservation.

Lemma 1 (Progress). *If $\vdash l : \tau$, then one of:*

- $l = v$
- *There exists some l' such that $l \mapsto l'$*
- *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*

The proof is by structural induction on the type derivation. If the last derivation step is T-Equiv, then the induction hypothesis is the desired result. Otherwise, the last derivation step determines the syntactic form of l.

The interesting cases are T-Array and T-FunApp. For both, the induction hypothesis states that progress applies to the subterms (array elements or function and argument arrays). This means that all subterms are values, or there is a leftmost non-value subterm. In the latter case, the non-value subterm is either reducible or a context around an out-of-domain application. Therefore, the array or function application is itself either reducible or a context around an out-of-domain application.

If all of an array's element terms are non-array values, then the array is itself a value. If they are all array values, T-Array requires that they have the same type, so *collapse* reduction is possible.

If all of a function application's terms are values, we must show that a β-, δ-, *map*-, or *lift*-reduction is possible. A well-typed term has no free variables, so the function and argument arrays' shapes are known. In order for the application to be well-typed, T-FunApp requires that there be a principal frame. This means that the arrays' frames can be fully described by just the frame rank—*i.e.*, how overranked the arrays are. If the principal frame has rank 0 (a scalar frame, no overrank), then β- or δ-reduction can be done (unless the argument arrays are outside of the primitive operator's domain). If the arrays are all overranked by the same amount, then *map* is applicable. Otherwise, the arrays are overranked by different amounts, so *lift*-reduction is possible. □

Lemma 2 (Preservation). *If $\vdash l : \tau$ and $l \mapsto l'$, then $\vdash l' : \tau$.*

This is also shown by induction on type derivations. In each case, it suffices to show that $\vdash l' : \tau'$ for some τ' equivalent to τ because $\vdash l' : \tau$ can then be reached with T-EQUIV. The induction hypothesis gives type preservation for subterms; it remains to show any reduction steps applied to the whole term will preserve its type. For terms matching a given type rule (other than T-EQUIV, whose case is trivial), only a few reduction rules may be applied.

Again, the interesting case is T-FUNAPP, which has several subcases. β-reduction simply substitutes arguments into a function body (it is only allowed in a scalar frame). The type derivation for the body of the λ can have the T-VAR uses for its substituted-out variables replaced by the derivations of the values replacing them. A *map* reduction converts an application form with type $\mathtt{A}_{\iota}\sigma$ to an array of applications. Each application's type is σ (from applying a $(\tau \ldots \to \sigma)$ to $\tau \ldots$), and the array around them has shape ι, so the type of the full expression is preserved. In a well-typed application, there is a principal frame ι for *lift* to target. While *lift*'s copying may change the types of the function and argument arrays, the application frame is unchanged, so the application is still typed with frame shape ι around the function's output type. □

D Implementation

Our PLT Redex model of our array language semantics is available at:
`https://github.com/jrslepak/Remora`