Rank Polymorphism Viewed as a Constraint Problem

Justin Slepak
College of Computer Science
Northeastern University
USA
jrslepak@ccs.neu.edu

Panagiotis Manolios
College of Computer Science
Northeastern University
USA
pete@ccs.neu.edu

Olin Shivers
College of Computer Science
Northeastern University
USA
shivers@ccs.neu.edu

Abstract

Rank polymorphism serves as a type of control flow used in array-oriented languages, where functions are automatically lifted to operate on high-dimensional arguments. The iteration space is derived directly from the shape of the data, presenting a challenge to compilation. A type system can characterize data shape, though the level of detail is beyond what can be reasonably expected from entirely human-generated annotations. The task of checking or inferring shapes can be phrased as solving constraints in the theory of the free monoid over the natural numbers, but the constraints involve both universal and existential quantification. Here is a plan of attack for leveraging past work on decision procedures, which has generally focused on the purely existential fragment of the theory.

CCS Concepts • **Theory of computation → Logic and verification**; *Type structures*; • **Software and its engineering → Polymorphism**; **Control structures**; *Functional languages*;

Keywords array-oriented languages, free monoid, word equations, first-order logic, type inference, indexed types

ACM Reference Format:
Justin Slepak, Panagiotis Manolios, and Olin Shivers. 2018. Rank Polymorphism Viewed as a Constraint Problem. In *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'18)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3219753.3219758

1 Introduction

Rank-polymorphic array programming, a paradigm pioneered by Iverson in his language APL [8] and expanded in the successor language J [10], offers an expressive notation for computing on large regular data. In this model, all functions automatically lift to operate on arguments of arbitrarily high dimension. This polymorphism enables a form of code reuse which is particularly convenient for domains such as scientific programming, machine learning, and signal processing. It also encourages programmers to write code in a way that makes data dependence clear. A compiler for such a language would not need the intricate analysis required in conventional scalar languages to identify which operations are legal to reorder or execute in parallel. However, even though the distinction between loop-carried and loop-independent data dependence is made clear in the code, the iteration space itself may be statically unknown when it is generated from arrays whose shape is dynamically computed.

Static characterization of the control structure of rank-polymorphic code is the goal of Remora, a language which generalizes the implicit function lifting of APL and J to handle higher-order and higher-arity functions, using restricted dependent types to describe array shapes. Using an array type indexed by a sequence of natural-number dimensions, Remora's type system identifies how each argument in a lifted function application must potentially be replicated to match the "principal frame" of the function application.

Remora's types are able to express detailed requirements as to argument shape, but in the language's current state, the programmer must supply type annotations partially explaining how arguments actually satisfy those requirements. For example, applying a matrix-inversion function that only accepts non-empty square matrices, *i.e.*, arrays with shape $(n+1) \times (n+1)$, to a 4×4 matrix requires explicitly instantiating the function at $n = 3$. Type-index variables may also range over *sequences* of naturals. The sequence variable $\vec{d}$ in $3 \times \vec{d}$ places no restriction on whatever lesser axes an array may have, but we require the major axis to have length 3.

The expressiveness of Remora's types comes with a cost: they can be so large that the actual program terms are lost amidst the clutter of their type annotations. We'd like some way of automatically inferring these types. The availability of both dimension (natural number) and shape (sequences of naturals) index variables means that the problem of inferring these index arguments amounts to solving a system of word equations which contains both universally and existentially quantified variables.

This paper examines the decision problems involved in both type checking and type inference for Remora from the perspective of constraints expressed in first-order logic. We give an overview of the language and its type system, including the language of type indices and the associated algebraic theory. We then describe the encoding of type checking as a constraint problem, along with how Remora's type checker ensures that actual constraint queries are inexpensive to check. Since the analogous constraint problem for type inference does not admit this simplification strategy, we instead sketch a different plan of attack, which we expect will allow us to take advantage of well-established decision-procedure technology to automatically select index arguments in Remora programs.

2 Typing Rank Polymorphism

We give the grammar for an explicitly-typed core language representing a subset of Remora [18] in Figure 1. For the purposes of this paper, we ignore type polymorphism and dependent sums; dependent products are enough to illustrate the problem at hand.

The type annotation on each formal parameter of a λ specifies the type of the argument *cells*, the fundamental units on which the function operates. When applying a function, each argument array is viewed as a two-level structure—a *frame* containing cells arranged in a particular shape. In essence, the shape of the cell is taken from the trailing dimensions of the argument array, while the shape of the frame is the remaining prefix of the argument's shape.

For example, a norm operator for points in $\mathbb{R}^3$, with cell type `(Arr (Shp 3) Float)`, would view an argument of type `(Arr (Shp 2 4 3) Float)` as a 2×4 matrix frame containing 3-vector cells. When there are multiple arguments involved, lifting rules require that some argument's frame have as prefixes all other frames involved in the function application; this is the *principal frame* of the application. A cross-product operator expecting two arguments with 3-vector cells could therefore be applied to a $2 \times 4 \times 3$ array (2×4 matrix frame) and a 2×3 array (2-vector frame), but using the same matrix alongside a 4×3 array (4-vector frame) as the second argument instead is not permitted. The scalar shape `(Shp)` is a prefix of every shape, so a scalar frame is of course compatible with anything.

Note the syntactic distinction between *expressions* (which represent arrays and computations which produce them) and *atoms* (the ur-elements which make up arrays). Constant array literals, noted as `array` forms, specify their shape, and constituent atoms, listed in "row major" order. The `frame` notation serves a similar purpose for expressions, arranging several array-expression cells (which must all have the same shape) into some specified containing frame; the shape of the resulting array is given by prefixing the shape of the cells with the shape of the frame collecting them. Thus, a `frame` whose cells are all literal arrays (with identical shape) collapses to an `array` form by appending the frame and cell shapes and appending the cells' atomic elements. However, we can also have code of the form `(frame (2)` e_1 e_2`)`, for general expressions e_1 and e_2, without yet knowing what values the two subexpressions will eventually produce—as long as the type system guarantees us they have identical shape. Once evaluation has proceeded far enough to reduce them to concrete values, we have something like

$e \in \mathit{Expr}$ ::=		*Expressions*
	x	*Variable reference*
$\mid$	`(array (`n`...)` $\mathfrak{a}$`...)`	*Array, containing atoms*
$\mid$	`(frame (`n`...)` e`...)`	*Frame of subarray cells*
$\mid$	`(`e_f e_a`...)`	*Term application*
$\mid$	`(i-app` e_f ι_a`...)`	*Index application*
$\mathfrak{a} \in \mathit{Atom}$ ::=		*Atoms*
	$\mathfrak{b}$	*Base value*
$\mid$	$\mathfrak{o}$	*Primitive operator*
$\mid$	`(`λ `((`x τ`)...)` e`)`	*Term abstraction*
$\mid$	`(I`λ `((`x γ`)...)` v`)`	*Index abstraction*
$v \in \mathit{Val}$ ::=		*Array value forms*
	x	
$\mid$	`(array (`n`...)` $\mathfrak{a}$`...)`	
$\tau \in \mathit{Type}$ ::=		*Types*
	B	*Base types*
$\mid$	`(-> (`τ`...)` τ`)`	*Functions*
$\mid$	`(Pi ((`x γ`)...)` τ`)`	*Dependent products*
$\mid$	`(Arr` ι τ`)`	*Arrays*
$\iota \in \mathit{Idx}$ ::=		*Type indices*
	x	*Index variable*
$\mid$	n	*Natural number*
$\mid$	`(Shp` ι`...)`	*Shape*
$\mid$	`(+` ι`...)`	*Adding naturals*
$\mid$	`(++` ι`...)`	*Appending shapes*
$\gamma \in \mathit{Sort}$ ::=		*Index sorts*
	`Dim`	*Dimension: one natural*
	`Shape`	*Shape: sequence of naturals*

Figure 1. Remora's concrete syntax is s-expression based.

```
(frame (2) (array (2 3) 1 2 3
                        4 5 6)
           (array (2 3) 7 8 9
                        0 1 2))
```

This `frame` of `array` literals in turn becomes

```
(array (2 2 3) 1 2 3 4 5 6 7 8 9 0 1 2)
```

Type indices are separated into sorts: `Dim` for an individual dimension and `Shape` for a (possibly empty) sequence of dimensions. Individual dimensions can be added together, and shapes can be appended to one another.

Remora also includes a bit of syntactic sugar for array notation. In the surface syntax, a bare atom $\mathfrak{a}$ appearing in expression position means `(array ()` $\mathfrak{a}$`)`, the scalar array whose sole element is $\mathfrak{a}$. A bracketed list `[`$e_1 \ldots e_n$`]` means `(frame (`n`)` $e_1 \ldots e_n$`)`. A `frame` form containing only values collapses to an `array` literal containing those values' atoms.

Putting these rules together, a bracketed list of atoms `[𝔞...]` is a vector containing those atoms, and we can write a 2×3 literal matrix as

```
[[1  0 1]
 [0 -1 1]]
```

Remora's core control-flow mechanism is *implicit* rank polymorphism. The semantics of function application consider what input types are expected and how those types must grow to match the actual arguments without asking the programmer to write out calls to `map` and `replicate` functions. The type system includes enough detail about array shapes to ensure statically that arguments in a function application are compatible. The matrix above is typed as `(Arr (Shp 2 3) Int)`, where `Arr` is a type constructor consuming a shape and an atom type (in this case, `(Shp 2 3)` and `Int`, respectively).

If we add this matrix to a vector of type `(Arr (Shp 2) Int)`, function application will automatically expand the vector into a `(Arr (Shp 2 3) Int)` matrix by replicating each scalar cell into a 3-vector. In this instance, replication is performed on scalar cells because `+` is fundamentally a scalar operator—its input type for each argument is `(Arr (Shp) Int)`. The arguments are compatible because the smaller frame `(Shp 2)` is a prefix of the larger frame `(Shp 2 3)`, which we denote as `(Shp 2)` $\sqsubseteq$ `(Shp 2 3)`. However, if we try to add this matrix to a vector of type `(Arr (Shp 3) Int)`, the program is rejected as ill-typed: neither `(Shp 3)` $\sqsubseteq$ `(Shp 2 3)` nor `(Shp 2 3)` $\sqsubseteq$ `(Shp 3)` holds, and expanding the scalar cells to vectors will not produce a 2×3 matrix.

In the previous example, operating on scalar cells means that an argument's entire shape is used as its frame shape. For the general case, where cells may not be scalars, we determine the arguments' frame shapes by breaking off their cell shapes from the *right-hand* side. That is, every argument's shape s is made up of f, the frame portion, appended to c, the cell portion: $s = f +\!\!+\, c$. A 3-vector dot product function can be written as

```
(define 3dot
  (λ ((x (Arr (Shp 3) Int))
      (y (Arr (Shp 3) Int)))
    (reduce + (* x y))))
```

If we apply this `3dot` function to our 2×3 matrix and a 3-vector, then those arguments *are* compatible. While `+` uses scalar cells, leaving us with incompatible argument frames `(Shp 2 3)` and `(Shp 3)`, `3dot` uses vector cells, so the argument frames are `(Shp 2)` and `(Shp)`, with `(Shp)` $\sqsubseteq$ `(Shp 2)`.

As another example, applying `3dot` to a $5 \times 4 \times 3$ array and a $5 \times 4 \times 6 \times 3$ array will produce a $5 \times 4 \times 6$ result array (recall: `3dot` produces scalar output cells). This application will apply the body of `3dot` to corresponding pairs of argument cells, but each cell in the first argument will be used 6 times with different cells from the second argument since the left frame must grow that much to match the right frame.

It may help the reader to illustrate the mechanics in terms of specific locations and regions in the arrays. The cell representing positions [0,0,0] through [0,0,2] in the first argument—call it the [0,0] cell—will be paired with each of the [0,0,0] through [0,0,5] cells from the second argument to compute the [0,0,0] through [0,0,5] cells of the result. Then the first argument's [0,1] cell is used with the [0,1,0] through [0,1,5] cells of the second argument and so on. Our original formal semantics [18] illustrates this process by explicitly replicating cells to produce an application form where everything has the same frame. However, for purposes of demonstrating the type-checking and type-inference problems, the essential bit is the rules which determine which shapes are compatible.

The frame-lifting mechanic applies to the function position in the same way as to the argument positions, with the rule that the cell shape for the function position is always scalar. A curried function can be applied to some large frame of argument cells, producing an array of functions which may then be used in function position in another application form. Suppose we have `c+`, a curried addition function:

```
(define c+
  (λ ((x (Arr (Shp) Int)))
    (λ ((y (Arr (Shp) Int)))
      (+ x y))))
```

If we apply `c+` to the vector `[1 2 3]`, we get a vector of unary functions, which we note here as `[(c+ 1) (c+ 2) (c+ 3)]`. Any application of this vector gives it a frame shape of `(Shp 3)` and a cell shape of `(Shp)`. If we apply it to a 3-vector of integers, as in

```
([(c+ 1) (c+ 2) (c+ 3)] [10 20 30])
```

we split the function array into cells `(c+ 1)`, `(c+ 2)`, and `(c+ 3)` and the argument array into cells `10`, `20`, and `30`. Our next step in evaluation is pointwise application, producing a vector of application forms `[((c+ 1) 10) ((c+ 2) 20) ((c+ 3) 30)]`, then stepping to `[(+ 1 10) (+ 2 20) (+ 3 30)]`, and finally resulting in the vector `[11 22 33]`. We could also apply it to a matrix as in

```
((c+ [1 2 3]) [[10 20]
               [30 40]
               [50 60]])
```

or equivalently

```
([(c+ 1) (c+ 2) (c+ 3)] [[10 20]
                         [30 40]
                         [50 60]])
```

This application form has `(Shp 3 2)` as its principal frame, and each function cell is applied to everything in the corresponding row of the matrix, giving a final result of

```
[[11 21]
 [32 42]
 [53 63]]
```

Permitting arrays of functions to appear in function position, treated as having scalar cells, allows first-class functions to integrate cleanly into the language, with curried functions giving the same final result (once fully applied) as their uncurried counterparts.

Recall our earlier matrix-vector addition example, where we added a 2×3 matrix and a 2-vector, effectively treating the second argument as a column vector. These frame shapes were compatible, since `(Shp 2)` $\sqsubseteq$ `(Shp 2 3)`, but we could not add the 2×3 matrix to a 3-vector. We might *want* to add that 3-vector though, treating it as a row vector. In order to add it to each row of our matrix, we need a version of the + function whose expected input type is `(Arr (Shp 3) Int)`, *i.e.*, one which consumes rank-1 cells instead of rank-0 cells. That will give us argument frames `(Shp)` and `(Shp 2)`, which are compatible. We can make this "reranked" + by η-expansion:

```
(λ ((x (Arr (Shp 3) Int))
    (y (Arr (Shp 3) Int)))
  (+ x y))
```

The function body uses the ordinary +, which operates pointwise on the scalar cells within `x` and `y`, producing a 3-vector result. (Computation over aggregates is managed by means of this "reranking" device so frequently in rank-polymorphic languages that Remora, like APL and J, provides a conveniently terse syntactic sugar for doing so.)

A more general-purpose vector addition function should work on vectors of any length. To do this, we need a dependent product type (roughly, a function parameterized over type indices rather than just terms). A dependent product is introduced by the `Iλ` form:

```
(Iλ ((len Dim))
  (λ ((x (Arr (Shp len) Int))
      (y (Arr (Shp len) Int)))
    (+ x y)))
```

Even without concrete shapes for `x` and `y` when they appear in the function body, the type checker can still prove that they must be vectors of the same length, which means that the scalar operator + can lift over them. So this function (which itself is an atom) can be given the type

```
(Pi ((len Dim))
  (Arr (Shp)
       (-> ((Arr (Shp len) Int)
            (Arr (Shp len) Int))
           (Arr (Shp len) Int))))
```

This type says that for any dimension `len`, we have (a scalar array whose sole element is) a function operating on two vectors whose length is `len`; furthermore, it produces a vector of the same length. As a minor aside, an `Iλ` term requires an (array-producing) expression for its body, while we have written a λ form, which is an atom. Desugaring turns the λ form into an `array` containing the function atom.

Dependent products may also abstract over `Shape`s, such as the append function whose type (when used on integer arrays) is

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

This type says that we can stitch two arrays together along their major axes as long as they agree as to the rest of their shapes, and the length of the resulting major axis is as long as the sum of the two arguments' major axes.

3 Type Checking as Constraint Solving

Type checking a Remora program requires inspecting each function call to ensure (1) that the function input type is compatible with the arguments and (2) that the frame shapes of the function and argument arrays are all prefixes of one principal frame. The rule for compatibility between a formal parameter's cell shape and an actual argument's shape is that the parameter's cell shape must be a suffix of the actual shape. The rest of the actual shape is that argument's frame.

As a simple case, consider

```
([+ -] [[10 20 30]
        [40 50 60]]
       2)
```

The `[+ -]` function array has type

```
(Arr (Shp 2)
  (-> ((Arr (Shp) Int)
       (Arr (Shp) Int))
      (Arr (Shp) Int)))
```

and the arguments' types are `(Arr (Shp 2 3) Int)` and `(Arr (Shp) Int)`. Since the function position in an application form expects cells with scalar shape, it is clear that condition 1 is satisfied. We have frame shapes `(Shp 2)`, `(Shp 2 3)`, and `(Shp)`. We resolve condition 2 by noting that `(Shp)` $\sqsubseteq$ `(Shp 2)` $\sqsubseteq$ `(Shp 2 3)` So `(Shp 2 3)` is the principal frame for this function application.

The inclusion of index variables, which may represent shapes or individual dimensions, complicates type checking. Even in a completely explicitly-typed language, where abstraction over and application of types and indices are written out in the program, we still have to determine shape compatibility and identify a principal frame for shapes that contain variables. Fortunately, type checking rank-polymorphic programs via shape-compatibility checking requires that two shapes will be compatible regardless of what concrete shapes and dimensions their variables might represent. Discovering *any* opportunity for the expected shape to differ from actual shape is grounds for rejecting the program as ill-typed.

The formal theoretical underpinning of Remora's type index language (*i.e.*, the theory of lists of natural numbers built up by concatenation) is the free monoid over $\mathbb{N}$. A monoid has an associative operation with an identity element. In Remora, the shape-appending `++` serves as the operation, with the scalar shape `(Shp)` as its identity. An algebraic structure is called *free* if it has no equalities other than those implied by its other axioms. The generators of an algebra are the "primitive" elements from which every element can be built using the algebra's operations. For our purposes, natural numbers serve as generators, since no single dimension can be decomposed into smaller non-scalar shapes. Remora's type system is built to leverage the existing corpus of work on decision procedures for the theory of a free monoid.

The shape criteria for a function application can be expressed as a first-order formula in this theory. Given two indices l and r, the prefix and suffix relations can be expressed by introducing a new existential variable c which completes the prefix (or suffix) producing the larger index. We write $\mathit{prefix}(l, r)$ as $\exists c.l + c = r$ and a $\mathit{suffix}(l, r)$ as $\exists c.c + l = r$. Condition 1 is then a conjunction of suffix assertions. For condition 2, we introduce another existential variable p, representing the principal frame. We know that it must be equal to one of the argument frames (a disjunction of equalities) and prefixed by all of them (a conjunction of prefix assertions). All of these are wrapped in universal quantifiers representing the index variables we have in the environment when type checking this function application.

However, we can be a bit more clever about our use of decision procedures and limit actual queries to a form with only universal quantifiers, for which the decision procedure is trivial. The associativity of the free monoid operation allows a canonical representation of shapes, provided we can determine equality for individual dimensions, which correspond to the monoid's generators. Of course, since the language of individual dimensions is Presburger arithmetic, checking whether an equality is valid is straightforward: ensure that each variable has the same number of appearances on both sides of the equality and that the constant component is also the same on each side.

In addition to checking whether an equality is valid, comparing canonical representations also reveals the validity of prefix and suffix assertions. The shape `(++ c (Shp 4) d)` is definitely prefixed by `(Shp)`, `c`, `(++ c (Shp 4))`, and `(++ c (Shp 4) d)`, but the prefix assertion would be invalid for any shape that does not share one of those canonical forms. This makes it possible to separate the checks for the two conditions. Check condition 1 on its own, and the completing indices which witness the suffix relation are the function and argument frame shapes. Then the witness for p, the principal frame, can be constructed by a linear search through the frames.

4 Type Inference as Constraint Solving

The decision problem becomes more complicated when we set out to relieve the programmer of the task of writing out index arguments. When we apply an index-polymorphic function, we introduce a new existential variable for each index argument we are trying to infer. These are in addition to the universal variables corresponding to the contents of the type-checking environment. We don't have a way to turn this into a series of queries in the easily decided universal fragment of the theory of a free monoid. Instead, we present here a way to construct solver queries by eliminating the *universal* quantifiers. Doing so reduces the type inference constraint problem to the existential fragment of the theory. This fragment is decidable, though the complexity is greater than that of the universal fragment.

The first step is to cast the task of selecting index arguments which make a function's input shapes $\iota\ldots$ compatible with known argument shapes $\kappa\ldots$ as a first-order logic formula. Suppose we have a function on m index arguments and n term arguments, whose type is of the form

```
(Pi ((e : γ) ...)
  (-> ((Arr ι τ) ...)
      τ'))
```

and we are applying it to n arguments whose respective types are `(Arr κ τ) ...`. This formula requires a conjunction of the equalities $(f + \iota = \kappa)\ldots$, each quantified with:

- An existential variable e for each index argument `e` of the corresponding sort γ
- An existential variable f of sort `Shape` to represent each argument's term frame
- A universal variable a for each index variable already bound at sort γ'

The existential variables must be permitted to depend on the universal variables. Then the shape compatibility formula is

$$\forall a : \gamma' \ldots \; \exists e : \gamma \ldots, f : \mathsf{Shape} \ldots \; \bigwedge_{i=1}^{n} (f_i + \iota_i = \kappa_i)$$

A solution for the existential variables $e\ldots$, written in terms of the universal variables $a\ldots$, is the missing information we need in order to elaborate a use of an index-polymorphic function into a form with explicit index application.

We run into an interesting dilemma when a function is polymorphic in cell *rank*, *i.e.*, when a variable ranging over `Shape` rather than `Dim` appears in the function's argument type. Such functions include `reduce`, `append`, and `scan`—an array-oriented language without these functions is a non-starter. In rank-polymorphic languages, such functions conventionally operate along the major axis of each cell. Then reranking the function to use smaller cells and lifting over the resulting frame performs the operation along some other axis. In Remora, these functions must take index arguments which specify the cell shape. Using a constraint of the above form, we end up with multiple solutions. If we append two matrices with the same shape (Shp 3 4), our frame and cell shapes could be resolved as (Shp) and (Shp 3 4), or as (Shp 3) and (Shp 4). The established convention can be respected by *assuming* a scalar frame, *i.e.*, including no existential variable for the argument's frame and treating it as (Shp).

As a concrete example, consider a small function which produces a matrix using arithmetic on some existing arrays and then tacks on a padding row.

```
(Iλ ((r Dim)
     (c Dim))
  (λ ((vec (Arr (Shp r) Int))
      (mat (Arr (Shp r c) Int))
      (pad (Arr (Shp c) Int)))
    (append (+ vec mat) [pad])))
```

Shape compatibility for the + application is simple because both of +'s input types are scalars. We query the validity of

$$\begin{aligned}&\forall\, \mathtt{r} : \mathtt{Dim},\ \mathtt{c} : \mathtt{Dim}\ \exists\, f_0 : \mathtt{Shape}, f_1 : \mathtt{Shape}, f_2 : \mathtt{Shape}\\ &\quad (f_0 + \!\!+ \ \mathtt{(Shp)} = \mathtt{(Shp)})\\ &\ \wedge (f_1 + \!\!+ \ \mathtt{(Shp)} = \mathtt{(Shp\ r)})\\ &\ \wedge (f_2 + \!\!+ \ \mathtt{(Shp)} = \mathtt{(Shp\ r\ c)})\end{aligned}$$

This gives us a scalar frame in function position and argument frames (Shp r) and (Shp r c). The maximum of these, (Shp r c), is the principal frame, which we set around +'s scalar Int output cells to derive the result type (Arr (Shp r c) Int).

The array [pad] is a singleton vector whose sole item is the vector pad, which makes its full shape (Shp 1 c). This is the extra bit we will tack on to the matrix produced by +. We must instantiate append's index arguments m, n, and s; they are respectively the lengths of the two arguments' leading axes and the shape of their elements when viewed as vectors. If viewing them as vectors gives them elements of differing shape, we cannot safely append them, and the program is therefore ill-typed. Following the major-axis convention, we do not need existentials for the argument frame shapes, though we do still have an existential f_0 for the function frame (which will only be resolvable as (Shp)). Our shape-compatibility constraint is

$$\begin{aligned}&\forall\, \mathtt{r} : \mathtt{Dim},\ \mathtt{c} : \mathtt{Dim}\ \exists\, f_0 : \mathtt{Shape}, \mathtt{m} : \mathtt{Dim}, \mathtt{n} : \mathtt{Dim}, \mathtt{s} : \mathtt{Shape}\\ &\quad (f_0 + \!\!+ \ \mathtt{(Shp)} = \mathtt{(Shp)})\\ &\ \wedge (\mathtt{(Shp\ m)} + \!\!+ \ \mathtt{s} = \mathtt{(Shp\ r\ c)})\\ &\ \wedge (\mathtt{(Shp\ n)} + \!\!+ \ \mathtt{s} = \mathtt{(Shp\ 1\ c)})\end{aligned}$$

The solution is f_0 = (Shp), m = r, n = 1, and s = c. Elaborating the function to use explicit index application gives

```
(Iλ ((r Dim)
     (c Dim))
  (λ ((vec (Arr (Shp r) Int))
      (mat (Arr (Shp r c) Int))
      (pad (Arr (Shp c) Int)))
    ((i-app append r 1 c)
     (+ vec mat) [pad])))
```

Note that polymorphism over the *frame* rank is still implicit, so the solution for the frame variables is not written out in the elaborated code.

5 Decidability of Inference Constraints

Unfortunately, the full first-order theory of equality in a free monoid is undecidable. Any endeavor to use decision procedures for constraint solving in an undecidable theory must first carve out a fragment of the theory that is large enough to express useful constraints but small enough to be decidable. In the free monoid theory, we are safe as long as we use only one type of quantifier—both the $\forall^*$ and $\exists^*$ fragments of the theory are decidable, whereas the $\exists^*\forall^*$ and $\forall^*\exists^*$ fragments are not. Moreover, checking validity of an equation in the $\forall^*$ fragment is much simpler than in the $\exists^*$ fragment. This is why we are careful in type checking to use universal-only constraints and then build frame-shape witnesses ourselves rather than the conceptually simpler strategy of throwing the mixed-prefix formula over the wall to a solver. This is enabled in part by separating the check for argument-shape compatibility from the search for the maximum frame. In the inference case, even if we don't existentially quantify frame shapes, we can't avoid using existential variables for the index arguments. Since there must be existential quantifiers in our constraint, we work at eliminating the universal quantifiers instead.

The general strategy for solving an existentially-quantified equation involves considering which segments of the left-hand side could correspond to which segments of the right-hand side. For example, suppose we are trying to equate these two shapes:

```
(++ c (Shp 5) d (Shp 5) d)
(++ d (Shp 5) (Shp 2) d e)
```

We might first try to make the (++ d (Shp 5)) from the first shape cover the (++ (Shp 2) d) segment from the second, or we might conjecture that the (++ c (Shp 5)) from the first is contained within the leftmost d from the second. Makanin's decision procedure [12] constructs a "generalized equation" for each possible alignment of the boundaries between components of the equated terms. A generalized equation encodes alignment-based restrictions, which pares down the solution space. A key feature of this algorithm which we can leverage is that boundaries can only appear inside an existential, not inside a generator. This is the same treatment we must give to universal variables in an equation when solving for its existential variables. If a universal variable is only partially covered by existentials, then some possible assignment for that universal makes the equation false. So we plan to translate our $\forall^*\exists^*$-quantified equation into a formula with only existential quantifiers by treating the universal variables as additional generators.

This plan of attack is applicable to type checking as well as type inference, allowing shape compatibility and principal frame to be checked in a single constraint query. However, we avoid doing so for type checking because it invokes a decision procedure that is much more expensive than necessary. The decision problem for equality in the existential fragment of the free monoid theory is NP-hard, since it is trivial to embed Presburger arithmetic—$\mathbb{N}$ with addition is the free monoid on one generator. By comparison, equality in the universal fragment is decidable in linear time by collapsing nested $+\!\!+$ terms according to associativity (though if we have only one generator, the monoid is commutative, so we must also count uses of each variable).

6 Related Work

The restricted form of dependent typing used in Remora was developed by Xi in Dependent ML [20]. Xi's work on programming with arrays focused on bounds checking on accesses to individual vector elements [21], with an index language based on Presburger arithmetic and constraint generation that takes advantage of the decidability of formulas even with arbitrarily nested alternating quantifiers. The rank-polymorphic programming model de-emphasizes operations on single elements, calling for an index language which describes whole array shapes rather than just individual dimensions.

Iverson designed the programming language APL [8], which is the original rank-polymorphic programming language. A series of design iterations, making the function-lifting rules more and more general, led eventually to its successor J [10]. In J, first-order functions may lift over two arguments, as long as either one's frame shape is a prefix of the other. The emphasis on regular, parallelizable control structure offers great performance opportunities, but these languages were implemented as sequential interpreters. The missed opportunities have encouraged several projects aimed at compiling APL code, such as that of Budd [2]. Bernecky's thesis [1] describes a compiler targeting the SISAL language [14] and motivates the effort with an analysis of interpretive overhead in real-world APL code. Grelck and Scholz [6] demonstrate the performance gains available from translating partially rank-specialized versions of APL functions into Single Assignment C [17]. Elsman and Dybdal [5] devise an intermediate representation for array programs which uses a type system to describe array shapes.

Several aspects of APL proper make static compilation difficult but are not essential to the rank-polymorphic array programming model. While compilation efforts have chosen to work on particular subsets of APL, several other related projects take a more clean-slate approach by building new languages to compile, generally with some combination of explicit function lifting and less detailed types. Jay's FISh [9] uses a static shape judgment to ensure that functions are compatible with their arguments. A procedure's behavior is described using a procedure which operates on shapes; an unacceptably-shaped argument must be detected by executing the shape procedure and determining that it returns an error. Array computation on GPUs is the target of Accelerate [3], a language embedded within Haskell, and Futhark [7], a standalone language. Aiming specifically to preserve implicit lifting, Thatte developed a system of type-directed coercions in an ML-like language [19], which automatically constructs the `map` and related calls to handle a subset of Iverson's lifting rules.

The satisfiability of equations in a free monoid was first proven decidable by Makanin [12], and the complexity was shown by Plandowski to be in PSPACE [16]. Karhumäki *et al.* extended the decidability result to satisfiability of boolean formulas over free monoid equations [11]. It has also been proven that the $\forall^*\exists^*$ fragment of the theory of a free monoid is undecidable, with Durnev showing undecidability for even $\forall^1\exists^3$ without negation [4]. Past work has typically treated monoid generators as opaque objects, but integrating with linear arithmetic on natural numbers might make use of Nelson-Oppen combination [15] or Manolios and Papavasileiou's framework for ILP modulo theories [13].

7 Conclusion

Our overarching goal is to address the historic difficulty in statically analyzing the shape-driven control structure of rank-polymorphic programs. Introducing a type discipline which is flexible enough to accommodate implicit lifting, even in the presence of high arity and first-class functions, was a first step. Since array processing functions may come with detailed conditions on the input shapes they accept, the rest of the analysis task is to find witnesses for arguments' compliance with functions' input requirements. Specifically, we must find these witnesses without simply telling the programmer to identify them. Removing the requirement for `map` and `replicate` calls, serving as use-specific adapter code, is a disappointing payoff if the cost turns out to be detailed, hand-written instantiations of cell-shape components for every call to a shape-polymorphic function.

Now the course is plotted. In order to statically identify the control structure of rank-polymorphic code without requiring the programmer to spell out all the ugly details, we need mechanized processes for generating and then solving constraints of the form described in Section 4. We are now confident that the constraints can be solved by reduction to a decidable fragment of the theory of a free monoid. Feedback from the community on this plan is welcome.

References

[1] Robert Bernecky. 1999. *APEX, the APL parallel executor.* Master's thesis. National Library of Canada= Bibliothèque nationale du Canada. http://hdl.handle.net/1807/11626

[2] Timothy Budd. 2012. *An APL compiler.* Springer Science & Business Media. https://doi.org/10.1007/978-1-4612-3806-5

[3] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11).* ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[4] Valery G Durnev. 1995. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Siberian Mathematical Journal* 36, 5 (1995), 917–929.

[5] Martin Elsman and Martin Dybdal. 2014. Compiling a Subset of APL Into a Typed Intermediate Language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14).* ACM, New York, NY, USA, Article 101, 6 pages. https://doi.org/10.1145/2627373.2627390

[6] Clemens Grelck and Sven-Bodo Scholz. 1998. Accelerating APL Programs with SAC. *SIGAPL APL Quote Quad* 29, 2, 50–57. https://doi.org/10.1145/379277.312719

[7] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[8] Kenneth E. Iverson. 1962. *A programming language.* John Wiley & Sons, Inc., New York, NY, USA.

[9] C. B. Jay. 1998. *The FISh language definition.* Technical Report. University of Technology, Sydney.

[10] Jsoftware, Inc. [n. d.]. Jsoftware: High-performance development platform. ([n. d.]). http://www.jsoftware.com/

[11] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. 2000. The Expressibility of Languages and Relations by Word Equations. *J. ACM* 47, 3 (May 2000), 483–505. https://doi.org/10.1145/337244.337255

[12] Gennady S Makanin. 1977. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics* 32, 2 (1977), 129–198. https://doi.org/10.1070/SM1977v032n02ABEH002376

[13] Panagiotis Manolios and Vasilis Papavasileiou. 2013. ILP Modulo Theories. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044 (CAV 2013).* Springer-Verlag New York, Inc., New York, NY, USA, 662–677. https://doi.org/10.1007/978-3-642-39799-8_44

[14] James McGraw, Stephen Skedzielewski, Stephen Allan, Dale Grit, Rob Oldehoeft, John Glauert, Ivan Dobes, and Paul Hohensee. 1983. *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1. 1.* Technical Report. Lawrence Livermore National Lab., CA (USA).

[15] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245–257. https://doi.org/10.1145/357073.357079

[16] Wojciech Plandowski. 2004. Satisfiability of Word Equations with Constants is in PSPACE. *J. ACM* 51, 3, 483–496. https://doi.org/10.1145/990308.990312

[17] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[18] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *European Symposium on Programming Languages and Systems.* Springer, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

[19] Satish Thatte. 1991. A type system for implicit scaling. *Sci. Comput. Program.* 17, 1-3 (Dec. 1991), 217–245. https://doi.org/10.1016/0167-6423(91)90040-5

[20] Hongwei Xi. 1998. *Dependent types in practical programming.* Ph.D. Dissertation. Pittsburgh, PA, USA. AAI9918624.

[21] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98).* ACM, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732