Refined Remora: Constraining Array Shapes

Vadym Matviichuk
Northeastern University
Boston, USA
matviichuk.v@northeastern.edu

Olin Shivers
Northeastern University
Boston, USA
shivers@ccs.neu.edu

Abstract

Remora is a higher-order functional array programming language based on the rank-polymorphic computational model originally developed by Iverson for APL. Unlike APL, Remora has a dependent type system that captures array shapes for use by the parallelizing compiler. Given that the type system is decidably checkable and inferrable, it inevitably must have limits on its expressiveness. In this paper, we extend Remora's type system by allowing programmers to refine array shapes with extra constraints, which are then checked by an SMT solver. We develop the shape-refinement type extensions for Remora; evaluate its utility for refining the types of standard computational kernels, such as convolution, as well as total applications, such as the YOLO vision application; and prove its type soundness.

CCS Concepts: • **Software and its engineering → Parallel programming languages**.

Keywords: array-oriented languages, refinement types

ACM Reference Format:
Vadym Matviichuk and Olin Shivers. 2026. Refined Remora: Constraining Array Shapes. In *Proceedings of the 12th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3815001.3815004

1 Refining Array Shape Types

Array programming languages (*e.g.*, APL [Iverson 1962] and later J [Hui and Iverson 1998]) have introduced a collection-oriented programming paradigm that allows users to express complex array computations concisely. Traditionally, these languages are dynamically typed, and thus hard to analyze and compile. Famously, APL sometimes needs to defer parsing to runtime [Strawn 1977]. Array programming languages, such as Futhark [Henriksen 2017] and Remora [Slepak 2020] attempt to provide more structure to this programming paradigm by introducing types — and the critical element we'd like such type systems to capture statically is the *shapes* of the arrays being processed by the program.

The main trade-off then becomes expressivity versus decidability. A more feature-rich type system is harder to type check, sometimes running into undecidability, and even harder to do type inference in a decidable manner. Remora, for example, has a type system permitting both decidable inference and checking; but decidability comes at the cost of expressivity—the only operations allowed at the type level are dimension addition and shape concatenation. Futhark, by contrast, allows dimensions to be arbitrary integer-typed expressions, only doing static checking in some cases [Henriksen and Elsman 2021].

One of the unique features of Remora is the ability to use and reason about shapes and shape variables. But decidability of such system comes at an expressivity cost. There are many functions, both in Remora's standard library and real-world applications, where the programmer knows constraints on argument and result shapes that cannot be expressed in the type system. For example, consider a convolution function that takes in two arguments: an n-dimensional input array of data, and a filter or kernel array of weights specifying the convolution. The input array, the filter array, and the result array must all have the same *rank*, *i.e.*, dimensionality — this is not too difficult to express in a type system. However, we'd also like to say that each dimension of the output array is given by subtracting the corresponding dimensions of the filter array from the dimension of the data array. That is, the shape of the result array is found by subtracting the shape of the filter array from the shape of the data array, in a point-wise fashion.

As another example, Remora provides a primitive function, `indices-of`, which operates on an array of any shape. Given, say, a 3×5 matrix, it produces a $3 \times 5 \times 2$ array, which can be viewed as a 3×5 matrix of the valid row/column index vectors. The last dimension of the result is the rank of the input shape — a fact we cannot express in the current Remora type system.

These extra properties cannot be expressed in Remora directly. Two of them would have to be abstracted using an existential type for indices, which is simply a (soundness-preserving) loophole in the type system permitting array shape information to be deferred to run time — which is a barrier to the compiler's ability to parallelize the code.

Our solution comes in the form of refinement types for the shape language. Refinement types extend the language

ARRAY '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2710-8/2026/06
https://doi.org/10.1145/3815001.3815004

by annotating types with predicates in a given logical theory. Traditionally, the choice of the logical theory is restricted to decidable ones. For example, LiquidHaskell [Vazou et al. 2014b] uses QF_EUFLIA. However, due to the lack of expressivity, we instead use an undecidable theory for both the refinements and the index language itself, namely QF_ENIA. It is expressive enough for all of the examples mentioned earlier, and terminates on all of the test programs we type checked.

The main contribution of the paper is the formalized semantics of refinements over indices (Section 4). Section 4.1 presents Refined Remora's type system, including a subkinding relation (Section 4.2), an extension to the shape equality checker (Section 4.3), and the lifting of operations pointwise over shapes (Section 4.4). The minor contribution of the paper is an implementation of the type checker in Haskell (Section 5). Evaluation of the implementation is in Section 6.

2 Background

2.1 Remora Tutorial

The core computational model in Remora comes from Iverson's rank-polymorphic "lifting" computational paradigm. This mechanism has some similarities to broadcasting in NumPy and PyTorch, but unlike these systems, it is not provided on a per-function basis. Instead, it is a general mechanism that is available at *every* function call, and it is the core source of Remora's parallel semantics. Remora's static and dynamic semantics are designed and built around this shape-based structure.

In rank-polymorphic languages, every parameter to a function has a base rank, or dimensionality. For example, a dot-product function expects vectors (rank 1), while the sine function expects a scalar value (that is, an array of rank 0). Any function that has been defined to operate on an array of rank r, is polymorphically lifted also to operate on argument arrays of rank $r' > r$. In essence, the extra $r' - r$ leading dimensions of the actual argument provide the "iteration space" for independent applications of the function. Thus, if we apply a vector-magnitude function, which expects inputs of rank 1, to a rank-3 array of shape $7 \times 2 \times 4$, it will be lifted to apply to a 7×2 "frame" of 14 independent 4-vector inputs (or "cells"), collecting the answers into a 7×2 matrix for a result.

For example, consider the function `add1`, which takes a *scalar* (an array with empty shape *i.e.*, a single value) and adds 1 to it. The expression `[1 2 3]` is a syntactic shorthand for writing array literals, in this case `(array (3) 1 2 3)`. This example constructs a 3-long vector. The first set of parenthesis describe the shape of the array, followed by a sequence of atoms. If at any point an atomic element appears in an array context, it is implicitly wrapped: 5 is transformed into `(array () 5)`.

```
add1: A int [] -> A int []
> (add1 5) ; scalar
6
> (add1 [1 2 3]) ; 3-long vector
[2 3 4]
> (add1 [[4 5] [8 9]]) ; 2x2 matrix
[[5 6] [9 10]]
```

Lifting works for any rank. Take the `sum` function that sums a 1-d array of numbers. It can be lifted to work over any n-d array as long as n is at least 1:

```
sum, prod: A int [d] -> A int []
> (sum [1 2 5])
8
> (sum [[3 5] [8 2] [4 7] [3 2]])
; -> (frame (4) (sum [3 5]) (sum [8 2])
;                (sum [4 7]) (sum [3 2]))
[8 10 11 5]
```

The general principle is that the argument is split into a frame and a cell. The *cell* is exactly the rank of the function parameter's shape (a scalar for `add1`, a vector for `sum`), while the *frame* is the rest of the shape. In the `add1` example, the cell is always `[]` and the frame is `[]`, `[3]`, `[2 2]` respectively. For the `sum` example, the cell is `[3]`, `[2]` and the frame is `[]`, `[4]` respectively. How does this principle scale to multiple arguments? The expression in the function position is *also* an array. Function array carries principal frame information between arguments. For clarity, in the future examples we will elide the scalar array type around functions.

```
add: A int [] ->
       A (A int [] -> A int []) []
> ((add 3) 5)
8
> ((add 2) [1 2 3])
[3 4 5]
```

When the shape of the function array and the frame of the argument are the same, stepping and type checking is straightforward. When they differ, one of them has to be replicated, so that the frames match, then split into scalar applications. Section 4.1 and Figure 6 explains this process in detail.

```
> ([sum prod] [[[1 5 4] [8 4 7] [2 3 5]]
               [[2 4 3] [5 1 2] [3 4 1]]])
[[10 19 10] [24 10 12]]
```

The first snippet is computed as follows: `[2]` is the shape of function array, `[2 3]` is the frame of the argument, `[2 3]` is the principal frame, so each function gets replicated to shape `[3]` from shape `[]`. Then the inputs are split such that each function is applied to an array of size `[3]`. Each scalar application is computed, and the results are combined. Here is how the earlier example of `[sum prod]` would step:

More intuitions and examples of this mechanism at work can be found in the Remora tutorial [Shivers et al. 2020].

```
([[sum sum sum] [prod prod prod]]
 [[[1 5 4] [8 4 7] [2 3 5]]
  [[2 4 3] [5 1 2] [3 4 1]]])
->
(frame (2 3)
  (sum [1 5 4])  (sum [8 4 7])  (sum [2 3 5])
  (prod [2 4 3]) (prod [5 1 2]) (prod [3 4 1]))
->*
(array (2 3) 10 19 10 24 10 12)
```

Figure 1. Reduction sequence example.

2.2 Refinement Types

Refinement types are an extension of a type system that allow specifying a subset of values from a given type using a predicate. For example,

$$x : \{v : \texttt{int} \mid 0 \leq v\}$$

specifies that x is a variable that represents a natural number, *i.e.*, a refinement on the integer type. The language of predicates is usually restricted to be decidable, so that the type checker can always accept or reject a program, but never diverge or not give an answer. Traditionally, refinement type systems also allow for a restricted form of dependent types, where predicates can depend on other in-scope, zero-order variables (*i.e.*, not functions). For example, in LiquidHaskell type for + (specialized to integers) is: `x:Int -> y:Int -> {v:Int | v = x + y}`. To make the notation more concise, when we write `Int` in a context where a refined type is expected, we mean `{v:Int | True}`.

3 Why Refinement Types?

Remora's design is centered on the task of providing a static type system that captures shape information in the context of its rank-polymorphic dynamic semantics. But the inevitable consequence of a decidable type system is limits on expressiveness. As a first example, take the append5 program (Figure 2) from Slepak's dissertation [Slepak 2020]. While contrived, it demonstrates how even a simple program can exceed the limits of expressivity in the original restricted system:

```
(define (append5 [a 1] [b 1])
  (+ [1 2 3 4 5]
     (append a b)))
```

Figure 2. Code for append5 function.

The 1 annotations on the `[a 1]` and `[b 1]` parameters specify that the function takes two vector arguments (*i.e.*, of rank 1). In order for this function to run successfully, the append operation must produce a result of length 5 (to match up with the first, five-element argument of the addition function). But Remora's type system does not let us specify that the lengths of the a and b arguments sum to 5.

This example was presented in the dissertation to demonstrate the absence of principal types, but there is no way to express the type of this function in statically typed Remora since there is no way to make the size of b dependent on size of a and restrict the size of a to be less than or equal to 5.

To do this, we can extend the type system, allowing us to *refine* the inputs to require this extra precondition. The type for the refined version of the append5 function is shown in Figure 3. We abandon the language's actual s-expression notation when showing these types in favor of more conventional notation, for ease of presentation. In the given type, Π is the type constructor for functions that abstract over shape and dimension indices, while A is the type constructor that constructs an array type from an element type (*e.g.*, `int`) and an array shape. Dimension `m` is less than or equal to 5,

```
append5 :
Π (m :: {d :: Dim | d <= 5}).
  Π (n :: {d :: Dim | d + m = 5}).
    A int (Shape m) ->
      A int (Shape n) ->
        A int (Shape 5)
```

Figure 3. Type of the refined append5.

and `n` is the difference between `m` and 5. There is now an extra equality in the type system: `m + n = 5`. This equality allows the type checker to verify the function statically. Note that, just as in traditional presentations of refinement types, refinement of `n` can depend on the earlier argument, `m`.

For a more real-world example, consider a strided, two-dimensional convolution, as is frequently used in neural networks. Inputs for our function are an image matrix, and a (smaller) filter matrix, along with a pad and a stride dimension. The function pads the image on all four sides with zeroes (increasing both its height and width by twice the pad amount), takes filter-sized windows of the padded image, stepping the windows by the requested stride amount, and calculates the weighted sum of each window according to the weights given by the filter matrix. In our type for this function, `n` provides the height and width of the image, and `k` likewise gives the size of the filter matrix, Here is a first attempt at giving a type for a convolution with no padding and a stride of 1:

```
convolution :
Π (n :: Dim).
  Π (k :: Dim).
    A f32 (Shape n n) ->
      A f32 (Shape k k) ->
        A f32 ???
```

One problem is that the dimension of the image array is not restricted to be at least the size of filter array's dimension. A more significant issue is that the type system does not allow expressing the shape of the output array: its height and width are both n − k + 1, but subtraction is not part of the type system's index language.

The type is still possible to express, though it is unnatural in a sense that the difference in dimensions is passed instead of the dimensions themselves. This version has fixed both issues:

```
convolution :
Π (n-k :: Dim).
  Π (k :: Dim).
    A f32 (Shape (n-k + k) (n-k + k)) ->
      A f32 (Shape k k)
        A f32 (Shape (n-k + 1) (n-k + 1))
```

The type system *does* allow us to add dimensions, so we can parameterize the type over dimension identifier `n-k`, denoting the difference, which we can then add to `k` everywhere we need to describe a dimension of the image array. This can be done, but it is awkward to torque our abstractions to fit the limitations of the index language. The more serious problem is that this type propagates the pattern of passing differences all the way up to the top-level of the program (this point is further explored in Section 6).

Adding padding is not a problem, since it is just addition. At this point, for the sake of clarity and brevity, we will combine multiple Π declarations into one, as seen in the next code snippet. The padding amount is denoted by dimension variable `p`, giving us the type

```
convolution :
Π (n-k :: Dim, k :: Dim, p :: Dim)
  A f32 (Shape (n-k + k) (n-k + k)) ->
    A f32 (Shape k k) ->
      A f32 (Shape (n-k + p + p + 1)
                   (n-k + p + p + 1))
```

However, there is no way to add stride, as derived dimensional calculations using the stride involve either multiplication or division, which is not supported by Remora's dependent types. The output-size formula with padding and stride (variable `st`) is $((n + 2 * p - k)/st) + 1$.

However, we can express all of these constraints using refinements on dimension and shape indices:

```
convolution :
Π (p :: Dim, @n :: Shp, st :: {d :: Dim | 0 < d},
   @k :: {@s :: Shp | (rank @n) = (rank @s)
                    ∧  @s <= @n + p + p})
  A f32 @n -> A f32 @k ->
  A f32 (1 + (@n - @k + p + p) / st)
```

Figure 4. Type of the refined `convolution`.

(In this dependent type, we use the convention that shape parameters, representing *sequences* of dimensions, are written with an `@` prefix, *e.g.*, the shape of the image array is `@n`.) This refined type requires that (1) the stride `st` must be positive; (2) the rank of the two arrays must match; and (3) the shape `@k` of the filter array must be at most the size of the padded input array (*i.e.*, `@n + p + p`). Without constraint #3, the size calculation for the shape of the result array would perform a divide-by-zero. Note that the relations and operations we use in the static shape refinements are expressed in a "lifted" notation not unlike the lifted computation Remora's dynamic value language provides: when we write `@n + p + p`, the single dimension `p` is point-wise added to every dimension in shape `@n`, twice; similarly, the dimensional relation `<=` is lifted to point-wise comparison across the two shapes in the third line of the type. Note that this type is polymorphic across convolutions of any rank, not just two-dimensional matrices, and the notation doesn't require the device of passing around differences.

As another motivating example, consider the following program written in original Remora. `M` has type `(A int [r c])` for some dimensions `r` and `c`. The inner application of `reduce` is an index application, followed by a regular application. For clarity, we will also write `(f a b)` to mean `((f a) b)`, for both regular and index applications.

```
flatten : Π (r :: Dim, c :: Dim).
            A t [r c] ->
              (Σ (a :: Dim) . A t [a])
M : A int [r c]
max : A int [] -> A int [] -> A int []
(unbox (a x ((flatten r c) M))
  ((reduce ???) max x))
```

Since original Remora does not have multiplication at the type level, this requires an existential Σ type. An existential type provides an abstract way to work with dimensions if they cannot be determined statically. `unbox` takes the result of flattening and binds the abstract dimension `a` to dimension `a`, value inside of type `(A t [a])` to `x`. The problem is that there is no way to write down that reduce because it requires input of size `d + 1` where `d` is the index parameter. The subtraction trick no longer works since that would require modifying or making a wrapper around existing standard library function. There is a solution that does not involve modifying standard library: appending the identity element to make the size `a + 1`. That does not always work since `reduce` only assumes associative operator, but not necessarily commutative or one that has an identity element, such as `max` or `min`. Another option would be to add a version of `flatten` that takes in and returns non-empty arrays. Refinements make this a lot cleaner, since it's possible to remove the existential types entirely.

```
λ (r :: { d :: Dim | 0 < d },
   c :: { d :: Dim | 0 < d })
```

$$
\begin{array}{rcll}
\alpha & \in & \mathit{IVar} & \textit{(index variables)} \\
x & \in & \mathit{Var} & \textit{(variables)} \\
k^B & ::= & \texttt{Shp} \mid \texttt{Dim} & \textit{(kinds)} \\
k & ::= & \{v :: k^B \mid p\} & \textit{(refined kinds)} \\
p & ::= & \texttt{true} \mid p \wedge p \mid p \vee p \mid \iota\, R\, \iota & \textit{(predicates)} \\
\delta & ::= & n \in \mathbb{N} \mid \alpha \mid \delta\ op\ \delta & \textit{(dimensions)} \\
\sigma & ::= & (\texttt{Shp}\ \delta \ldots) \mid \alpha \mid \sigma \mathbin{+\!\!+} \sigma \mid \sigma\ op\ \iota & \textit{(shapes)} \\
 & & \mid \iota\ op\ \sigma & \\
\iota & ::= & \delta \mid \sigma & \textit{(indices)} \\
b & ::= & \texttt{int} \mid \texttt{f32} & \textit{(base types)} \\
\tau_{at} & ::= & b \mid \tau_{arr} \to \tau_{arr} \mid \Pi\ \alpha :: k\ .\ \tau_{arr} & \textit{(atomic types)} \\
\tau_{arr} & ::= & \texttt{A}\ \tau_{at}\ \sigma & \textit{(array types)} \\
a & ::= & z \in \mathbb{Z} \mid \lambda x : \tau_{arr}\ .\ e \mid \lambda \alpha :: k\ .\ e & \textit{(atoms)} \\
v & ::= & (\texttt{array}\ (n \ldots)\ a \ldots) & \textit{(values)} \\
e & ::= & v \mid (\texttt{frame}\ (n \ldots)\ e \ldots) \mid x \mid e\ e \mid e\ \iota & \textit{(terms)} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau_{arr} & \textit{(term context)} \\
I & ::= & \cdot \mid I, \alpha :: k & \textit{(index context)} \\
E & ::= & \square & \textit{(evaluation contexts)} \\
 & & \mid (\texttt{frame}\ (n \ldots)\ v \ldots\ E\ e \ldots) \mid E\ e \mid v\ E \mid E\ \iota &
\end{array}
$$

Figure 5. Refined Remora Grammar.

$\boxed{\Gamma; I \vdash e : \tau_{arr}}$

$$\frac{\forall i \in \{1 \ldots m\}\ .\ \Gamma; I \vdash a_i : \tau \qquad m = \Pi(n \ldots)}{\Gamma; I \vdash (\texttt{array}\ (n \ldots)\ a_1 \ldots a_m) : \texttt{A}\ \tau\ (\texttt{Shp}\ n \ldots)}\ \text{E-Arr}$$

$$\frac{\Gamma, x : \tau_{arr}; I \vdash e : \tau'_{arr}}{\Gamma; I \vdash \lambda x : \tau_{arr}\ .\ e : (\tau_{arr} \to \tau'_{arr})}\ \text{A-Fun}$$

$$\frac{\begin{array}{c}\Gamma; I \vdash e_1 : \texttt{A}\ (\texttt{A}\ \tau_1\ \sigma_1 \to \texttt{A}\ \tau_2\ \sigma_2)\ \sigma_f \\ \Gamma; I \vdash e_2 : \texttt{A}\ \tau_1\ (\sigma_a \mathbin{+\!\!+} \sigma'_1) \\ \sigma_p = \sigma_a \sqcup \sigma_f \qquad I \vdash \sigma_1 = \sigma'_1\end{array}}{\Gamma; I \vdash e_1\ e_2 : \texttt{A}\ \tau_2\ (\sigma_p \mathbin{+\!\!+} \sigma)}\ \text{E-App}$$

$$\frac{\Gamma; I, \alpha :: k \vdash e : \tau_{arr}}{\Gamma; I \vdash \lambda \alpha :: k\ .\ e : \Pi\ \alpha :: k\ .\ \tau_{arr}}\ \text{A-IFun}$$

$$\frac{\begin{array}{c}\Gamma; I \vdash e : \texttt{A}\ (\Pi\ \alpha :: k\ .\ \texttt{A}\ \tau\ \sigma)\ \sigma_f \\ I \vdash \iota :: k' \qquad \Gamma; I \vdash k' <: k\end{array}}{\Gamma; I \vdash e\ \iota : \texttt{A}\ \tau[\iota/\alpha]\ (\sigma_f \mathbin{+\!\!+} \sigma[\iota/\alpha])}\ \text{E-IApp}$$

Figure 6. Selected Refined Remora Typing Judgements.

```
λ (M : (A int [r c]))
  ((reduce (r * c)) max (flatten M))
```

Since we declared `r` and `c` to be greater than zero, the SMT solver can deduce that their product is positive, and therefore the call to `reduce` is safe.

4 Semantics of Refined Remora

4.1 Type System

To simplify and focus the presentation, we drop some extraneous elements of the language that are not central to the issues of this work from the theoretical portion. In particular, we eliminate index existential and universal types, and we assume that functions have a single parameter and index operations are binary.[1]

This section contains detailed explanations of the typing judgements in order to provide more intuition for how the system works and where the refinements come into play.

E-Arr is the introduction form for arrays. The shape of the array is described by a sequence of natural numbers (note that variables and operations are not allowed), and the number of atoms needs to be equal to the product of the natural numbers. The `array` form is *always* a value — a constant array whose elements are either numbers or lambdas. The elements of the `frame` form are all expressions producing arrays of identical shape; after stepping each expression to a value, the arrays are "plugged into" the given frame shape to produce the form's result. The `array` and `frame` forms require the type to be the same for all atoms or expressions respectively, which might include Π types, meaning no variance on refinements.[2]

A-Fun is a standard typing judgement for functions. The only detail of note is that input and output types *have* to be array types since all expressions, and therefore variables, are arrays. The elimination form, E-App is where the lifting actually happens. Figure 1 shows an example of how type checking happens. We type check the function expression to be an array of shape σ_f of functions $\texttt{A}\ \tau_1\ \sigma_1 \to \texttt{A}\ \tau_2\ \sigma_2$, and the argument expression to have type $\texttt{A}\ \tau_1\ (\sigma_a \mathbin{+\!\!+} \sigma'_1)$. For the example, `sum` is a scalar (rank 0 / empty shape) array whose sole element is a function of type `(A int [3]) -> (A int [])`, while the argument has type `(A int [2 3 3])` We must then check the equality of σ_1 and σ'_1. This is one of the places where refinements come into play, as the SMT solver uses the constraints to check equality of shapes (Section 4.3). This means that the argument needs to be *at least* the shape of the parameter. The principal frame is constructed by joining the two frames $\sigma_p = \sigma_f \sqcup \sigma_a$. The join relation chooses the longest common prefix, and it can be undefined: `[2]` and `[3]` do not have a common prefix. The principal frame is the maximal way to lift a given function application, such that each cell has a scalar function and the input array matches

[1] Since Remora is a higher-order language, multi-parameter functions can be encoded simply by currying them; the principal-frame negotiation between the function-position array and the argument-expression array translates to this setting with no problems.

[2] There is an alternative version of the system that uses type joins ($\Gamma; I \vdash a_i : \tau_i$ and then $\tau = \sqcup_i(\tau_i)$ instead of $\Gamma; I \vdash a_i : \tau$). The join would $\wedge$ all the predicates from all functions in the array. This system seems sound, but we did not deem it worth the extra complexity.

$$\boxed{I \vdash \iota <: \iota}$$

$$\frac{\begin{array}{c} i = \texttt{ToSMTEnv}(I, v :: k^B) \\ p_s = \texttt{ToSMTPred}(i, p) \qquad q_s = \texttt{ToSMTPred}(i, q) \\ \texttt{SMT-Query}(i, p_s \implies q_s) \end{array}}{I \vdash \{v :: k^B \mid p\} <: \{v :: k^B \mid q\}} \text{ Sub-SMT}$$

Figure 7. Subkinding Relation

$$\boxed{I \vdash \iota = \iota}$$

$$\frac{\begin{array}{c} i = \texttt{ToSMTEnv}(I, v :: k^B) \\ (x, p) = \texttt{ToSMT}(i, \iota_1) \qquad (y, q) = \texttt{ToSMT}(i, \iota_2) \\ x \neq y \qquad \texttt{SMT-Query}(i, p \wedge q \implies x = y) \end{array}}{I \vdash \iota_1 = \iota_2} \text{ SMT-Idx-Eq}$$

Figure 8. Index Equality.

exactly what the function expects. This is the essence of rank polymorphism [Slepak 2020]. In the example, σ_f is empty, σ_a is `[2 3]`, the prefix part of argument shape. Therefore, the principal shape is `[2 3]`. The result shape is constructed by appending the principal frame to the output type of the function. In the example, that would be `(A int [2 3])`.

A-IFun is once again a function type that looks standard, similar to how universal quantification types are presented. E-IApp is where refinements actually come into play and are checked. We type check the expression e to be an array of shape σ_f of Π functions, and the index ι to have kind $\{w :: k' \mid q\}$. We then check that the argument's kind is a subkind of the function's parameter. The subkinding relation is explained in Section 4.2. Afterwards, we substitute ι for α where it is bound: in τ and σ but not σ_f. Since the shape of the function array is σ_f, that is what defines a principal frame for the index application.

4.2 Subkinding Relation

The subkinding relation can be found in Figure 7: the relation uses a standard technique for subtyping of refinement types: checking if the first predicate implies the second one [Jhala and Vazou 2020, p.12]. We use an SMT solver to check if the implication holds. The important part is to use the same environment to turn both syntactic predicates (p and q) into SMT formulae to ensure that the resulting formula is correct.

4.3 Index Equality

Index equality (more specifically, shape equality) lies at the core of the type-checking algorithm in the application case. The rule itself looks simple: construct a substitution for the index environment, convert each index into a pair of a symbolic value and a formula, and then check if the two values are equal under the two formulae.

To give an idea of how this works, consider the `append5` function in Figure 2, more specifically the shape check on the `+` application (`m` and `n` are dimensions of `a` and `b`).

```
[1 2 3 4 5] : A int (Shp 5)
(append a b) : A int (Shp (m + n))
                m :: {d :: Dim | d <= 5}
                n :: {d :: Dim | d + m = 5}
```

First, construct a substitution. A substitution maps names in the source language to a pair of a symbolic value and a formula representing all the associated constraints. To do that for `m` and `n`, first instantiate `m`. Let `x` be the generated SMT variable. Then evaluate the right-hand side, with `d` mapped to `x`. The resulting formula is `x <= 5`. Do the same process for `n`, but with `x`'s substituted for `m`. Let `y` be the generated variable for `n`. The formula would be `y + x = 5`. Now, the substitution is complete: [`m` ↦ (`x`, `x <= 5`), `n` ↦ (`y`, `y + x = 5`)].

The next step is the equality check. Since the shapes on both sides are singletons, the only proof obligation is to show `m + n = 5`. Evaluate both sides to SMT expressions, along with the formulas that were computed earlier. `5` will evaluate to $(5, \top)$. `m + n` will evaluate to $(x+y, x \leq 5 \wedge y+x = 5)$. The evaluator itself is simple: (1) for each variable return the right-hand side of the substitution, (2) for each constant return that constant and $\top$ (for truth), and (3) for any expression return SMT version of the operator applied to evaluated arguments and combine all the formulas with `&&`. The last step is to check using SMT whether the conjunction of two formulas implies equality. For this example, the formula is $(x \leq 5 \wedge y + x = 5) \implies x + y = 5$. There are also some constraints, namely `x >= 0 && y >= 0`, but they were elided for brevity (but that is the reason for the `d <= 5` constraint for `m`). The SMT solver will return `True` on that formula.

4.4 Shape Operation Lifting

This work extends the set of possible computations on indices. First, the type system now allows extra operations like multiplication, division, subtraction and so on. They allow user to type check more programs that required an existential type before since actual dimension was not expressible. The need for an existential type comes from the fact that there are indices that are not expressible, so they are treated in an abstract way. We also added the `rank` operator that returns the length of the shape as a dimension.

The more interesting feature is operation lifting. In the original Remora semantics the only allowed arithmetic operation was additions on dimensions. In practice, there are often times when it's beneficial to either operate on shapes point-wise (*e.g.*, `@img+@w` from convolution) or lift operation to work on dimensions and shapes at the same time (*e.g.*, `@img + p + p`). Remora makes this kind of lifting ergonomic, so Remora itself was the source for the semantics of operator

$\boxed{I \vdash \iota :: k}$

$$\frac{}{I \vdash n :: \{d :: \texttt{Dim} \mid d = n\}}\ \text{K-DimN} \qquad \frac{I(iv) = k}{I \vdash iv :: k}\ \text{K-Ivar}$$

$$\frac{}{I \vdash (\texttt{Shp}\ \sigma \ldots) :: \{s :: \texttt{Shp} \mid s = (\texttt{Shp}\ \sigma \ldots)\}}\ \text{K-ShpLit}$$

$$\frac{}{I \vdash \delta_1\ op\ \delta_2 :: \{d :: \texttt{Dim} \mid d = \delta_1\ op\ \delta_2\}}\ \text{K-OpDim}$$

$$\frac{\iota_1 = \sigma \vee \iota_2 = \sigma}{I \vdash \iota_1\ op\ \iota_2 :: \{s :: \texttt{Shp} \mid s = \iota_1\ op\ \iota_2\}}\ \text{K-OpShp}$$

$$\frac{I \vdash \iota :: k' \qquad I \vdash k' <: k}{I \vdash \iota :: k}\ \text{K-Subsume}$$

Figure 9. Kinding Relation.

lifting at the index level. The "arrays" of this shape language are dimensions and shapes. Kinds serve as "types" of this shape language: the dimension kind is a scalar natural number, the shape kind is a vector of natural numbers. Figure 9 showcases this: if an operation's arguments are all dimensions, the kind of the result is a dimension, but if there is at least one shape, then the kind of the result is a shape. All operations (except `rank`) operate on scalars, so type checking is simple: only the equality of lengths of vectors (*i.e.*, ranks) needs to be checked.

4.5 Conversion to SMT

The functions that convert Remora indices to SMT are in Figure 10.

An important part of the conversion is the encoding of types. Since, dimensions cannot be negative, the obvious choice is natural numbers. We assume this for the theoretical version of the conversion, although the intermediate results can be negative. Then the shapes becomes lists (or sequences) of natural numbers.

ToSMTEnv function creates an environment, which maps Remora variables to a pair of an SMT literal and the associated constraint. There is one interesting detail here: since the refinement of an index can depend on other in-scope indices, this needs to be done in the right order. Typing judgements introduce variables in a correct order, but a practical implementation will need to construct an ordering for the traversal.

ToSMT function takes in an index substitution map and an index and produces a tuple of an SMT value representing that index and an SMT formula describing all constraints associated with it. For example, if we have $i = [a \mapsto (\text{x}, \text{x} <= 5), b \mapsto$ (y, y >= 6)], calling ToSMT(i, (Shp a b)) will return (SMT-List(x, y), x <= 5 && y >= 6) and the constraints that are needed by the SMT solver to represent the list. The interesting case here is operations over shapes. For the case of a shape and a dimension, partially apply the operator to the dimension, and map the resulting function over the shape. For the case of two shapes, use the operator as the function for map2, which takes two lists and applies the function to the corresponding elements, and check that the two shapes

$\boxed{\texttt{ToSMT}(i, \iota)}$

$\texttt{ToSMT}(i, n) = (\text{n}, \text{True})$

$\texttt{ToSMT}(i, iv) = i(iv)$

$\texttt{ToSMT}(i, (\text{Shp}\ \delta \ldots)) = (\text{SMT-List(sd)}, \text{SMT-All(sc)})$
  where $(\text{sd}, \text{sc}) \ldots = \texttt{ToSMT}(i, d) \ldots$

$\texttt{ToSMT}(i, s_1 \text{ ++ } s_2) = (\text{SMT-Concat(sl1, sl2)}, cr)$
  where $(\text{slj}, \text{scj}) = \texttt{ToSMT}(i, s_j)$
    cr = sc1 && sc2

$\texttt{ToSMT}(i, \delta_1\ op\ \delta_2) = (\text{dr}, \text{sc1 \&\& sc2})$
  where $(\text{sdj}, \text{scj}) = \texttt{ToSMT}(i, \delta_j)$
    dr = sd1 ToSMT(op) sd2

$\texttt{ToSMT}(i, \sigma\ op\ \delta) = (\text{sr}, \text{scl \&\& scd})$
  where (sl, scl) = ToSMT(i, σ)
    (sd, scd) = ToSMT(i, δ)
    sr = SMT-Map((ToSMT(op) sd), sl)

$\texttt{ToSMT}(i, \sigma_1\ op\ \sigma_2) = (\text{sr}, \text{scl1 \&\& scl2 \&\& sclen})$
  where (sl1, scl1) = ToSMT(i, σ_1)
    (sl2, scl2) = ToSMT(i, σ_2)
    sclen = SMT-Length(sl1) = SMT-Length(sl2)
    sr = SMT-Map2(ToSMT(op), sl1, sl2)

$\texttt{ToSMTEnv}(\cdot) = []$

$\texttt{ToSMTEnv}(I, x :: \{v :: k^B | p\}) =$
  $i[x \mapsto (\text{y} : \texttt{ToSMTK}(k^B),$
    $\texttt{ToSMTPred}(i[v \mapsto (\text{y}, \text{True})], p))]$
  where i = ToSMTEnv(I), y fresh in i

$\texttt{ToSMTPred}(i, True) = \text{True}$

$$\texttt{ToSMTPred}(i, p \wedge q) = \begin{array}{l}\texttt{ToSMTPred}(i, p) \\ \text{\&\& } \texttt{ToSMTPred}(i, q)\end{array}$$

$\texttt{ToSMTPred}(i, \iota_1\ R\ \iota_2) = \text{p1 \&\& p2 \&\& (x ToSMTOp(R) y)}$
  where (x, p1) = ToSMT(i, ι_1)
    (y, p2) = ToSMT(i, ι_2)

Figure 10. Conversions to SMT values.

$$\boxed{E[e] \to E[e']}$$

$$E[(\texttt{array}\ ()\ \lambda x : \tau\ .\ e)\ v] \to E[e[v/x]] \qquad (\beta)$$
$$\text{if } \cdot \vdash v : \tau$$

$$E[(\texttt{array}\ (\vec{n})\ \vec{f_i})\ v] \to E[(\texttt{frame}\ (\vec{n})\ \overrightarrow{(f_i\ v_i)})] \qquad (\beta_S)$$
$$\text{where } \cdot \vdash v : \texttt{A}\ \tau\ ((\texttt{Shp}\ \vec{n})\ \texttt{++}\ \sigma)$$
$$\cdot \vdash f_i : \texttt{A}\ \tau\ \sigma \to \tau',$$
$$(\vec{v_i}) = \texttt{Split}(v, \vec{n})$$

$$E[(\texttt{array}\ (\vec{n_f})\ \vec{f})\ (\texttt{array}\ (\vec{n_a}\ \vec{n})\ \vec{a})] \to \qquad (\beta_R)$$
$$E[(\texttt{array}\ (\vec{n_p})\ \vec{f_r})\ (\texttt{array}\ (\vec{n_p}\ \vec{n})\ \vec{a_r})]$$
$$\text{where } (\vec{f_r}) = \texttt{Replicate}(\vec{f}, [], \vec{n_f}, \vec{n_p})$$
$$(\vec{a_r}) = \texttt{Replicate}(\vec{a}, \vec{n}, \vec{n_f}, \vec{n_p})$$
$$\vec{n_p} = (\texttt{Shp}\ \vec{n_p}) \sqcup (\texttt{Shp}\ \vec{n_a})$$
$$\cdot \vdash f : \texttt{A}\ \tau\ (\texttt{Shp}\ \vec{n}) \to \tau'$$

Figure 11. Selected operational semantics rules.

have the same length. ToSMTPred is very similar, but it returns a formula instead of a tuple. The interesting case is the comparison (with R being the comparison operator). The result of interpreting equality of ι_1 and ι_2 is p1 && p2 && x == y where (x, p1) and (y, p2) are the results of converting the corresponding indices. For shapes, the comparison involves checking that the length are equal and that all elements are equal point-wise.

SMT-Query is a function that takes in a substitution and an SMT formula, and checks whether that formula holds. The substitution provides all the names and types that the formula has as free variables. This paper does not contain a definition for this function, as it is meant to be an abstract way to call SMT solver.

4.6 Operational Semantics

Figure 11 presents the operational semantics for lifting a function call in Remora. The reduction rules for the index lambdas are not in the figure since they are exactly the same as the first two rules for regular lambda (since indices cannot be combined into an array), and therefore the only source of lifting is the expression in the function position.

Note that type annotations are required for the reduction relation. Without types, there is no indication of what the function expects, and therefore there is no way to compute the principal frame of the application. Even dynamically typed Remora requires rank annotations on functions.

The first rule (β) is similar to a regular application, except that the lambda is a scalar array. If there is nothing to lift over (*i.e.*, the argument is the expected shape and function array is a scalar), then the only thing left to do is to substitute the argument for the variable x.

The next rule (β_S) is responsible for splitting functions and arguments into a frame of independent applications. This can only happen if the frame (over the cells) of the argument and the function are the same (*i.e.*, the principal frame). `Split`, while missing a formal definition, returns a list of length `product(`$\vec{n}$`)` whose elements are σ-sized chunks of input.

The last rule (β_R) is the replication step. It works as follows:

1. Find the principal frame
2. Replicate each function and each argument cell to the principal frame.
3. Put them back into the array forms, so that `Split` can make them into a frame of applications.

In general an application steps as follows: first replicate, then split, run each scalar application, and combine the results.

4.7 Type Soundness

Type soundness of this system holds via a progress and preservation proof.

Lemma 4.1 (Progress). *If* $\Gamma; I \vdash e : \tau$ *then either* e *is a value or* $\exists e'.e \to e'$.

Lemma 4.2 (Preservation). *If* $\Gamma; I \vdash e : \tau$ *and* $e \to e'$, *then* $\Gamma; I \vdash e' : \tau$.

This paper does not contain full proofs of these lemmas, because the operational semantics are the same as those in the original Remora, and there is only one lemma that would need to change: index substitution.

Lemma 4.3 (Index Substitution). *If* $\Gamma; I_1, \alpha :: k, I_2 \vdash e : \tau, I_1 \vdash \iota :: k', k' <: k$, *then* $\Gamma[\iota/\alpha]; I_1, I_2[\iota/\alpha] \vdash e[\iota/\alpha] : \tau[\iota/\alpha]$.

Proof sketch. By K-Subsume rule for the kinds and the premises, $I \vdash \iota :: k$. Therefore, after the substitution, all the generated formulae are the same, and therefore, the typing derivation tree stays the same. □

4.8 Decidability

The current type system as presented is undecidable in general. The logical theory of the predicate language is QF_ENIA plus a restricted version of the Theory of Sequences, where indexing into sequences is disallowed in the source language. On the other hand, this logic is intended to let programmers express properties about the *shape* of their data, not the actual element values themselves. In typical array codes, these shapes tend to be reasonably simple, to the odds that a theorem prover can successfully discharge proof obligations without user input are good.

There are a couple of other possible options to resolve this issue if timeouts do occur in practice. One of them would be to allow a special annotation to guard an assertion with a run-time check, allowing the compiler to assume the guard holds in downstream code. Since these checks must be done

on a per-aggregate (that is, per-array) basis, rather than a per-element basis, the constant-time cost of the run-time check is amortized over the processing of all of the aggregate's items. We hope to explore this possibility in future work.

Another approach would be a system similar to the `Proof` part of LiquidHaskell [Vazou et al. 2018] or `TP` part of `sbv`, *i.e.*, SMT-assisted theorem proving. This would eliminate all run-time checks, but comes at a cost of doing proof engineering beforehand. This also requires a lot of engineering effort to make such tool both usable and correct.

4.9 Refinement Erasure

The definition of erasure is: if a program type checks at a refinement type, then the erased program will type check at the erased type (for simplicity, erasure replaces all the predicates with `True`). Formally, if $\Gamma; I \vdash e : \tau$, then $\text{erase}(\Gamma); I) \vdash \text{erase}(e) : \text{erase}(\tau)$. This kind of property is expected to hold in this type system (since it holds in traditional refinement type systems). But, it does not hold for a peculiar reason. To see why that is, examine all the typing judgements that involve working with indices, E-IApp and E-App. In the former, subkinding is the only check that involves refinements. Since all predicates in the program became true, the implication is always true, and therefore the typing judgement still holds. The latter is the interesting case. The important check is the equality of the function parameter shape and the non-frame part of the argument's shape. The problem arises because the SMT solver checks equality, and it uses equalities from the constraints that are gone after erasure. To illustrate this, look at the `append5` functions in Figure 3. In the original type system, we can never deduce $\mathtt{m} + \mathtt{n} = 5$. In the refined type system, if $\mathtt{m} \leq 5, \mathtt{m} + \mathtt{n} = 5$, then $\mathtt{m} + \mathtt{n} = 5$ and $\mathtt{m} \geq 0, \mathtt{n} \geq 0$. Erasing predicates removes these equalities, and therefore might make the program not well-typed. This does not arise in traditional type system since they do not check equality of expressions, or indices in Remora's case.

5 Implementation

This type checker is implemented in Haskell, using the `sbv` library to interface with the SMT solver, Z3 in our case. The entire implementation is about 600 lines of code, which follow the semantic definitions closely. The implementation is available at https://github.com/matviichukv/refined-remora-haskell/. The two important parts of this code base are the subkind relation and the equality relation.

An essential part of the implementation is the environment structure. In this implementation, the environment is represented as a record of 4 fields: `term`, corresponding to Γ; `idx`, corresponding to I; and `symDim` and `symShp`, mapping names to symbolic representations of dimensions and shapes respectively. The representation is a tuple of a symbolic variable, and a symbolic formula, representing the constraint associated with that value. For example, $\mathtt{m} :: \{v :: \mathtt{Dim} \mid v = 5\}$, would have the following entry in `symDim`: $\mathtt{m} \mapsto (\mathtt{d}, \mathtt{d} = 5)$.

5.1 Theory of Sequences

Dimensions are interpreted as symbolic integers with a constraint that they have to be non-negative (i.e., a natural number). Shapes are more complicated to deal with. Rosette [Torlak and Bodik 2014] and Grisette [Lu and Bodík 2023] do not allow constructing lists of some indeterminate length n. The usual approach would be to construct lists of all possible lengths up to some bound, so that the program is terminating. Shape variables in Π require lists of symbolic length to be properly represented, so a more expressive embedding is required.

The answer comes in the form of the theory of sequences. The `sbv` library has bindings to use the theory of sequences in Z3. The theory of sequences provides a type in Z3, namely `(Seq t)`, along with operations on them (`length`, `nth`, etc.), and comes bundled with decision procedures to prove propositions involving them. The rest of the implementation follows the definitions in Figure 10.

5.2 Error Reporting

Since all the checks are independent of each other, and instantiations are done on a per-check basis, we can report errors in type checking in a modular way. Also, because it is expressed as a satisfiability problem to SMT, if a given call does not type check, Z3 provides a counter-example which should guide the user as to which constraints are missing or are contradictory. We implemented this in the type checker. As an example, consider the following call from a convolution implementation:

```
windows:
Π (@n :: Shp,
   @w :: {@s :: Shp | (@s <= @n) ∧
                      (rank @s) = (rank @n) },
   st :: {d :: Dim | 0 < d})
  [t @n] -> [t ((1 + ((@n - @w) / st)) ++ @k)]

((windows (+ @n p p) @k st) ...)
```

Assume that the constraint on `st`, `0 < d`, was replaced with `True`. The type checker would provide the following counter-example (with 3 lines of output removed) and an expression where the check failed (since we did not do source location tracking for the prototype). From there figuring out what went wrong in the type checking process is straightforward: since `windows` divides by `st`, evaluating the counter-example results in a divide-by-zero.

```
(IApp (IApp (IApp windows (@n + p + p)) @w) st)
Falsifiable. Counter-example:
  "p"    =  0 :: Integer
  "st"   =  0 :: Integer
```

```
  s"img" = [] :: [Integer]
  s"w"   = [] :: [Integer]
```

6 Evaluation

We annotated Remora's standard library, various utility functions and a real-world application (at a per-kernel level), YOLOv1 (convolutional neural network) with refinements. The main evaluation criterion is whether we can type check the refined versions of the functions without SMT timing out or using existential types.

6.1 Standard Library

Many functions in Remora's standard library have unergonomic types. A common issue is the pattern of encoding the requirement that an array have a non-zero dimension by abstracting the dimension with some index `d`, and then using $d+1$ in the shape of the array. Functions `head`, `tail`, `reduce` and others all use this pattern. Consider a function that does matrix multiplication as an example. For clarity, we will add more code conventions. `[t d1 d2]` means `A t (Shp d1 d2)`. Dimension variables are quantified implicitly by default. The type after all the arguments is the return type of the function.

```
(define (mm (M [f32 m n])
            (N [f32 n o]))
            :  [f32 m o]
  ((lift-dot n o) M N))
```

Next, define a helper function that takes the dot product of one row of `M` with the columns of `N`.

```
(define (lift-dot (v [f32 n])
                  (N [f32 n o]))
                  :  [f32 o])
  ((dot n) v (transpose N)))
```

Lastly, implement function `dot` that takes the dot product.

```
(define (dot (v [f32 n])
             (w [f32 n]))
             :  [f32 [])
  ((reduce ???) + (* v w)))
```

The problem arises from the type of `reduce`. It accepts a dimension `d`, and an array of size `(+ d 1)`. But the only dimension variable available is `m` and there is no subtraction operator in original Remora. So the only solution is to replace dimension `m` with index `m-1`, then specifying inputs to be of size `(+ m-1 1)`. In this specific case, we could append `0` to the start, or use a different version of `reduce`, one that accepts an initial element, but that is not always possible. For example, functions `max` and `min` do not have an identity element.

```
(define (dot (v [f32 (+ 1 n-1)])
             (w [f32 (+ 1 n-1)]))
             :  [f32])
  ((reduce n-1) + (* v w)))
```

The real problem comes from the fact that this pattern propagates: `lift-dot` needs to pass `m-1`, so it has to do the same trick. This pattern then repeats all the way up to the top level:

```
(define (lift-dot (v [f32 (+ n-1 1)])
                  (M [f32 (+ n-1 1) o]))
                  :  [f32 o])
  ((dot n-1) v (transpose M)))

(define (mm (A [f32 m (+ n-1 1)])
            (B [f32 (+ n-1 1) o]))
            :  [f32 m o]
  ((lift-dot n-1 o) A B))
```

Using refinement shapes gets around this problem by directly constraining the dimension be greater than zero.

```
reduce: Π (r :: {d :: Dim | d > 0}, @s :: Shp)
          ([t @s] -> [t @s] -> [t @s]) ->
            [t r @s] -> [t @s]
```

A similar propagation happens here, where each `m` parameter now needs a greater-than-zero constraint. This still has benefits over the original pattern: (1) the constraint is specified once per argument, (2) the constraints can sometimes combine, such as $(1 < d) \wedge (2 < d)$ is just $2 < d$, and (3) input shapes are just `m` instead of `(m-1 + 1)` which we argue is the more natural way to write this down.

Figure 12 shows the refined matrix multiplication implementation.

```
(define (matmul ((m Dim) (o Dim)
                 (n { d :: Dim | (< 0 d) }))
                ((M [f32 m n])
                 (N [f32 n o])))
                 :  [f32 m o]
  ((lift-dot n o) M N)))

(define (lift-dot ((n { d :: Dim | (< 0 d) })
                   (o Dim))
                  ((v [f32 n])
                   (N [f32 n o])))
                   :  [f32 o]
  ((dot n) v ((transpose n o) M)))

(define (dot ((n { d :: Dim | (< 0 d) }))
             ((v [f32 n])
              (w [f32 n])))
              :  [f32]
  ((reduce n) + (* v w)))
```

Figure 12. Code for refined matrix multiply.

There are a number of other standard library function that have a better type signature now; we have been able to verify all of them. Here are some examples:

```
flatten:
Π (m :: Dim, n :: Dim, @c :: Shp)
  [t (++ [m n] @c)] ->
    [t (++ [(m * n)] @c)]

unravel:
Π (m :: Dim, n :: Dim, @c :: Shp)
  [t (++ [(* m n)] @c)] ->
    [t (++ [m n] @c)]

indices-of:
Π (@s :: Shp) [int (++ @s [(rank @s)])])

// Defined over a shape instead of a dim:
reduce:
Π (@c :: Shp, @fr :: { @s : Shp | (< 0 @s) })
  ([t @c] -> [t @c] -> [t @c]) ->
    [t (++ @fr @c)] ->
      [t @c]
```

6.2 Machine Learning

YOLOv1. YOLOv1 [Redmon et al. 2016] is a well-known convolutional neural network. It is one of the motivations for this work, since the Remora implementation contained many awkward patterns and array shapes that could not be specified with the standard Remora types, forcing the use of existential types. There are some interesting refinements we found for convolution: (1) the rank of the input and the filter arrays must be the same (2) the filter's size must be at most the size of the padded image, (3) the stride must be non-zero (because of division), and (4) depending on use case, we might want the padded input to be divisible by the stride to avoid dropping any data.

Maximum pooling has a very similar refined type, since at its core it is a computation very similar to convolution (*i.e.*, partition the input into windows and do some per-window operation). Thus, all the refinements for convolutions that do not involve weights apply to max pooling layers as well. The one extra refinement required here is `@k > 0` since `max` does not have an identity element. There is one more refinement that might be useful here — divisibility, which can be expressed as `d = (d / k) * k`. We use integer division (since our domain are integers), so `d = (d / k) * k` is the same as `d mod k = 0`. This constraint ensures no data is dropped from mismatched dimensions. Other layers do not have interesting invariants to turn into constraints, but they do use operations such as multiplication in the type.

Other computational kernels. The other interesting kernels we tried are ViT (Vision Transformer) Patch Embedding and softmax. Patch embedding is similar to convolution and max pooling, due to its use of strided windows. The main difference is that it requires the input array's shape to be cleanly divisible into a number of patches and that it does not do any sort of padding. So stride is not a parameter, but a computed value $n/patch$. Softmax is simpler, only requiring the input to have at least 2 elements.

6.3 Performance

We have tested the type checker on a number of test cases, as described in the previous sections. For full list, look at the artifact for this paper, in the `examples/` directory.

For all real world examples, such as convolution, max pooling, *etc.*, the type checker terminates instantaneously. As we mentioned before, the new type system is not decidable. There are many simple programs on which the SMT solver would not terminate, although most of the ones we have encountered seem contrived. One such example is the following program, (in the s-expression format the actual compiler uses as an internal syntax).

```
(env ; variables available in the expr
  (idx @aa { @s :: Shp | (= @s 1) })
  (idx @ab { @s :: Shp | (and (= @s 1)
                              (= (rank @s)
                                 (rank @aa)))})
  (term v (A Float @aa)))
(result (A Float @ab))
v ; expression to type check
```

This is a simple file format we use for test cases. An `env` is a list of pairs of either terms and types (such as `v` and `(A Float @aa)`) or indices and their kinds (such as `@aa` and `{@s : Shp|(= @s 1)}`). The expression should have the type specified in `result` field. The only check is the equality of shapes `@aa` and `@ab`. `@aa` is a list of natural numbers of some length where each element is equal to one. `@ab` is also a list of natural numbers that is the same length as `@aa`, and where each element is also equal to one. The two lists are the same: the length is the same, and all the elements are equal to one.

Z3 times out on this simple program. Why? There are five constraints in this example: (1, 2) each element in both lists is non-negative, (3, 4) each element in both lists is equal to one, (5) the length of the two lists are equal. Constraints 1-4 get encoded as a `foldr and true ...`, and in the case of the `sbv` library, this is further transformed into a recursive function. We are not sure why this is happening, but Z3 cannot reason well about equality of lists constructed through recursive functions. Our best guess, currently, is that Z3 is trying to brute force the solution, and it is not working since the length is arbitrary. This is supported by the fact that if the length of `@aa` is capped at 5, Z3 terminates instantly, and as the cap on length increases, the execution time becomes longer (0.1ms for 5, 0.8ms for 15, 6.6s for 25, 25.5s for 35, 161s for 55, 4590s for 100). This can be avoided by constraining the two shapes either to be equal to each other, or, better still, using only one of the shapes.

7 Related Work

Array Programming Languages. The rank-polymorphic array-programming model originated with Iverson's APL [Iverson 1962], and its successor J. These languages are dynamically typed, so checks on array shapes must be implemented as run-time tests. The language R is similarly dynamic; there have been efforts to develop a dependent type system for R [Wrenn et al. 2023]. Julia is a popular language for scientific computation, but there has not been an effort to add refinement types to it (either through a library, or language/compiler extensions). Array libraries, such as NumPy, verify invariants by doing runtime checks in the absence of shape types. This approach is more fragile since developer errors aren't caught at compile time. Haskell's Accelerate library [Chakravarty et al. 2011] only statically tracks the rank of array shapes, not the actual dimensions, due to absence of dependent types. Futhark [Henriksen 2017] has a much richer language for dimensions, allowing arbitrary dimension computation in the type with two stipulations: it has `i64` type and the type checker will try to do the checks at compile-time but will produce runtime checks if it cannot [Henriksen and Elsman 2021]. The difference is that Futhark can only talk about dimension variables and not shape variables. Another language in this general space is Dex [Paszke et al. 2021]. Dex uses a dependent type system to express array shapes and dimensions, but its system does not admit *any* type polymorphism, so one cannot write functions over arbitrary dimension/shape.

Refinement Types. Refinement types are a long-standing area of research. Early work is due to Freeman [Freeman and Pfenning 1991], the goal of which was to allow recognizing certain instances of a recursive type (such as singleton lists as a subtype of all lists). This used purely built-in type mechanisms for checking and did not use any SMT solvers. LiquidHaskell [Vazou et al. 2014a] is a language extension adding "liquid" types to Haskell. Liquid types (Logically Qualified Data Types) is a design that admits limited dependent and refinement types, but in a restricted way such that local inference is still decidable. It does not have an n-dimensional array type, so the only way to directly encode them is through nested lists which limits the programmer to arrays of specific rank.

PyTorch and NumPy. PyTorch and NumPy are popular Python libraries for building machine-learning and other array-processing applications. There are three main differences between these libraries and the Remora language. The first one is that Remora is statically typed. The main benefit here is that, in Remora, array shapes drive parallelism, so the shape types expose parallelism to the compiler; In Python, this must be deferred to runtime. Also, Python's asserts are for a single input, not a proof for all possible inputs: this mechanism is on the wrong side of the "von Neumann bottleneck." PyTorch has more flexibility with dimensions with a feature called Dynamic Shapes. It uses symbolic values for dimensions to parameterize a network to work over different input sizes or parameters. This allows the system to show that certain constraints hold for every possible assignment to the integers — but it does not allow us to work with arbitrary shapes. The second issue is that broadcasting is not automatically done in the Python libraries. While all the library primitives lift as we would expect them to, user-defined functions needs to be vectorized manually, either through a `vectorize` function call or a `vectorize` annotation. Lastly, the mechanism for lifting and the mechanism for broadcasting are different, although they serve a similar purpose.

8 Future Work

Refinement Inference. Type inference is a popular technique that decreases burden on users by inferring (some of) the type annotations. There are multiple published methods on inferring refinements [Cosman and Jhala 2017; Pavlinovic et al. 2021]. We did not implement inference for this system for a number of reasons. The main one is that we would need to do data-flow analysis to do the inference properly, a non-trivial task outside the (current) scope of this work. Beyond this extra complexity, however, we do not envision any problems with inference, since for the most part, the language is simple, and lifting can be expressed with `map` (which is how Futhark implements lifting), which can be inferred with the system mentioned earlier.

SMT encoding. The non-terminating example (Section 6.3) can be proved using a simple proof by induction. The solver's failure is almost certainly due to suboptimal SMT encoding. We would like to do a more thorough exploration of SMT encodings to improve both performance and expressivity.

9 Conclusion

We started this work based on two beliefs:

- The mechanism of refinement types is well-suited to the task of describing array shapes and related constraints in an extensible way.
- The computational difficulties of proving theorems specifically about the array shapes in real-world programs, using modern decision systems, should be tractable.

As shown by the practical and theoretical results, those beliefs seem to hold when applied to practical programs.

Acknowledgments

This work was supported by NSF and the DARPA MOCHA program.

Appendix A Semantics for Refined Remora

A.1 Type System

$\boxed{\Gamma; I \vdash e : \tau_{arr}}$

$$\frac{\forall i \in \{1 \ldots m\} \,.\, \Gamma; I \vdash a_i : \tau \qquad m = \Pi(n \ldots)}{\Gamma; I \vdash (\texttt{array}\ (n \ldots)\ a_1 \ldots a_m) : \texttt{A}\ \tau\ (\texttt{Shp}\ n \ldots)}\ \text{E-Arr}$$

$$\frac{\forall i \in \{1 \ldots m\} \,.\, \Gamma; I \vdash e_i : \texttt{A}\ \tau\ \sigma \qquad m = \Pi(n \ldots)}{\Gamma; I \vdash (\texttt{frame}\ \ (n \ldots)\ e_1 \ldots e_m) : \texttt{A}\ \tau\ ((\texttt{Shp}\ n \ldots) \texttt{++}\ \sigma)}\ \text{E-Fr}$$

$$\frac{x : \tau \in \Gamma}{\Gamma; I \vdash x : \tau}\ \text{E-Var}$$

$$\frac{\Gamma, x : \tau_{arr}; I \vdash e : \tau'_{arr}}{\Gamma; I \vdash \lambda x : \tau_{arr} \,.\, e : (\tau_{arr} \to \tau'_{arr})}\ \text{A-Fun}$$

$$\frac{\begin{array}{c}\Gamma; I \vdash e_1 : \texttt{A}\ (\texttt{A}\ \tau_1\ \sigma_1 \to \texttt{A}\ \tau_2\ \sigma_2)\ \sigma_f \\ \Gamma; I \vdash e_2 : \texttt{A}\ \tau_1\ (\sigma_a \texttt{++}\ \sigma'_1) \\ \sigma_p = \sigma_a \sqcup \sigma_f \qquad I \vdash \sigma_1 = \sigma'_1\end{array}}{\Gamma; I \vdash e_1\ e_2 : \texttt{A}\ \tau_2\ (\sigma_p \texttt{++}\ \sigma)}\ \text{E-App}$$

$$\frac{\Gamma; I, \alpha :: k \vdash e : \tau_{arr}}{\Gamma; I \vdash \lambda \alpha :: k \,.\, e : \Pi\ \alpha :: k \,.\, \tau_{arr}}\ \text{A-IFun}$$

$$\frac{\begin{array}{c}\Gamma; I \vdash e : \texttt{A}\ (\Pi\ \alpha :: k \,.\, \texttt{A}\ \tau\ \sigma)\ \sigma_f \\ I \vdash \iota :: k' \qquad \Gamma; I \vdash k' <: k\end{array}}{\Gamma; I \vdash e\ \iota : \texttt{A}\ \tau[\iota/\alpha]\ (\sigma_f \texttt{++}\ \sigma[\iota/\alpha])}\ \text{E-IApp}$$

A.2 Operational Semantics

$\boxed{E[e] \to E[e']}$

$$E[(\texttt{frame}\ \ (\vec{n})\ (\texttt{array}\ (\vec{m})\ \vec{v_a}) \ldots)] \to E[(\texttt{array}\ (\vec{n}\,\vec{m})\ \vec{v_a} \ldots)] \qquad (\beta_f)$$

$$E[(\texttt{array}\ ()\ \lambda x : \tau \,.\, e)\ v] \to E[e[v/x]] \quad \text{if } \cdot \vdash v : \tau \qquad (\beta)$$

$$E[(\texttt{array}\ (\vec{n})\ \vec{f_i})\ v] \to E[(\texttt{frame}\ \ (\vec{n})\ \overrightarrow{((\texttt{array}\ ()\ f_i)\ v_i)})] \quad \text{where } \cdot \vdash v : \texttt{A}\ \tau\ ((\texttt{Shp}\ \vec{n}) \texttt{++}\ \sigma),\ \cdot \vdash f_i : \texttt{A}\ \tau\ \sigma \to \tau',\ (\vec{v_i}) = \texttt{Split}(v, \vec{n}) \qquad (\beta_S)$$

$$E[(\texttt{array}\ (\vec{n_f})\ \vec{f})\ (\texttt{array}\ (\vec{n_a}\ \vec{n})\ \vec{a})] \to E[(\texttt{array}\ (\vec{n_p})\ \vec{f_r})\ (\texttt{array}\ (\vec{n_p}\ \vec{n})\ \vec{a_r})] \quad \text{where } (\vec{f_r}) = \texttt{Replicate}(\vec{f}, [], \vec{n_f}, \vec{n_p}),\ (\vec{a_r}) = \texttt{Replicate}(\vec{a}, \vec{n}, \vec{n_f}, \vec{n_p}),\ \vec{n_p} = (\texttt{Shp}\ \vec{n_p}) \sqcup (\texttt{Shp}\ \vec{n_a}),\ \cdot \vdash f : \texttt{A}\ \tau\ (\texttt{Shp}\ \vec{n}) \to \tau' \qquad (\beta_R)$$

$$E[(\texttt{array}\ ()\ \lambda \alpha :: k \,.\, e)\ \iota] \to E[e[\iota/\alpha]] \quad \text{if } \cdot \vdash \iota <: k \qquad (\beta^i)$$

$$E[(\texttt{array}\ (\vec{n})\ \vec{f})\ \iota] \to E[(\texttt{frame}\ \ (\vec{n})\ \overrightarrow{((\texttt{array}\ ()\ f)\ \iota)})] \qquad (\beta^i_S)$$

References

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)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1926354.1926358

Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 26:1–26:27. doi:10.1145/3110270

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. *SIGPLAN Not.* 26, 6 (May 1991), 268–277. doi:10.1145/113446.113468

Troels Henriksen. 2017. *Design and Implementation of theFuthark Programming Language.* Ph. D. Dissertation.

Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ACM, Virtual Canada, 1–14. doi:10.1145/3460944.3464310

Roger K. W. Hui and Kenneth E. Iverson. 1998. *J: Dictionary.* Iverson Software.

Kenneth E. Iverson. 1962. *A Programming Language.* Wiley, New York. Literaturangaben.

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. arXiv:2010.07763 [cs] doi:10.48550/arXiv.2010.07763

Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Reproduction Package for Article "Grisette: Symbolic Compilation as a Functional Programming Library"* 7, POPL (Jan. 2023), 16:455–16:487. doi:10.1145/3571209

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–29. doi:10.1145/3473593

Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data Flow Refinement Type Inference. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021), 19:1–19:31. doi:10.1145/3434300

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640 [cs] doi:10.48550/arXiv.1506.02640

Olin Shivers, Justin Slepak, and Panagiotis Manolios. 2020. Introduction to Rank-polymorphic Programming in Remora (Draft). arXiv:1912.13451 [cs] doi:10.48550/arXiv.1912.13451

Justin Slepak. 2020. *A Typed Programming Language: The Semantics of Rank Polymorphism*. Ph. D. Dissertation. Northeastern University, United States – Massachusetts.

George O. Strawn. 1977. Does APL Really Need Run-Time Parsing? *Software: Practice and Experience* 7, 2 (1977), 193–200. doi:10.1002/spe.4380070207

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541. doi:10.1145/2594291.2594340

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 39–51. doi:10.1145/2633357.2633366

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014b. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–31. doi:10.1145/3158141

John Wrenn, Anjali Pal, Alexa VanHattum, and Shriram Krishnamurthi. 2023. Dependently Typing R Vectors, Arrays, and Matrices. https://arxiv.org/abs/2304.04265v1. doi:10.48550/arXiv.2304.04265

Received 2026-04-06; accepted 2026-04-29