Towards Formalizing the Semantics of Lifting Argument Frames When Evaluating Remora Expressions

Ruben Gamboa

Kestrel Institute
Palo Alto, CA

University of Wyoming
Laramie, WY
ruben@uwyo.edu

ACL2 Workshop 2026
Menlo Park, CA

Outline

- Background
- Semantics of Lifting
- Reasoning about Remora Programs: First Steps
- Questions?

Remora

- Remora is a rank-polymorphic functional programming language
- It is based on APL
- It's designed to work well with GPUs

Remora

- Remora is a rank-polymorphic functional programming language
- It is based on APL
- It's designed to work well with GPUs

- No explicit iteration
- No loops, recursion, etc.

Remora

- Remora is a rank-polymorphic functional programming language
- It is based on APL
- It's designed to work well with GPUs

- No explicit iteration
- No loops, recursion, etc.

- Instead, it uses **lifting** to apply functions to arrays

Objectives

- The implementation of Remora relies on fancy compiler transformations
- These transformations should be formally verified

Objectives

- The implementation of Remora relies on fancy compiler transformations
- These transformations should be formally verified

- This requires having a formal model of Remora in ACL2
- Kestrel developed an implementation of Remora, complete with a parser and an interpreter

Objectives

- The implementation of Remora relies on fancy compiler transformations
- These transformations should be formally verified

- This requires having a formal model of Remora in ACL2
- Kestrel developed an implementation of Remora, complete with a parser and an interpreter

- This same framework can be used to reason about programs written in Remora
- That's the motivation for this work

Complications

- The theorems describe how a Remora expression is evaluated by the ACL2 interpreter

Complications

- The theorems describe how a Remora expression is evaluated by the ACL2 interpreter
- Thus, the theorems need to reason about the Remora program **and** the interpreter

Complications

- The theorems describe how a Remora expression is evaluated by the ACL2 interpreter
- Thus, the theorems need to reason about the Remora program **and** the interpreter
- Challenge: The interpreter is complicated

Complications

- The theorems describe how a Remora expression is evaluated by the ACL2 interpreter
- Thus, the theorems need to reason about the Remora program **and** the interpreter
- Challenge: The interpreter is complicated
  - Errors
  - Curried evaluation
  - Lifting

Outline

- Background
- Semantics of Lifting
- Reasoning about Remora Programs: First Steps
- Questions?

Remora Arrays

- All “values” in Remora are arrays
- Arrays can be or rank 0 (scalars), 1 (vectors), 2 (matrices), etc.
- Arrays contain cell elements, which are of traditional scalar types, e.g., numbers

Remora Arrays

- All “values” in Remora are arrays
- Arrays can be or rank 0 (scalars), 1 (vectors), 2 (matrices), etc.
- Arrays contain cell elements, which are of traditional scalar types, e.g., numbers

- The **rank** of an array is the number of dimensions it has
- The **shape** is a list of the sizes of each dimension

Remora Arrays

- All “values” in Remora are arrays
- Arrays can be or rank 0 (scalars), 1 (vectors), 2 (matrices), etc.
- Arrays contain cell elements, which are of traditional scalar types, e.g., numbers

- The **rank** of an array is the number of dimensions it has
- The **shape** is a list of the sizes of each dimension

- E.g., this is a rank 2 array (matrix) of shape (2, 3):

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Binary Operations without Lifting

- A binary operation, e.g., addition, takes two arrays as input and produces a third array as output

- When the array arguments have the same shape, the operation is done pairwise

$$2 + 3 = 5$$

Binary Operations without Lifting

- A binary operation, e.g., addition, takes two arrays as input and produces a third array as output
- When the array arguments have the same shape, the operation is done pairwise

$$2 + 3 = 5$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

Binary Operations without Lifting

- A binary operation, e.g., addition, takes two arrays as input and produces a third array as output
- When the array arguments have the same shape, the operation is done pairwise

$$2 + 3 = 5$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Binary Operations with Lifting

- When the array arguments are compatible, lifting is performed automatically to make the shapes match
- Arrays are compatible if the shape of one is a prefix of the shape of the other

Binary Operations with Lifting

- When the array arguments are compatible, lifting is performed automatically to make the shapes match
- Arrays are compatible if the shape of one is a prefix of the shape of the other

- Compatible arrays

$$2 + \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

- Incompatible arrays

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Binary Operations with Lifting

- An array is lifted by replicating it along the missing dimensions
- E.g., if A is a rank 1 array it is indexed as $A[i]$
- When looking for $A[i][j]$, the value of $A[i]$ is returned regardless of j
- And that's why arrays are compatible if the shape is a prefix

Binary Operations with Lifting

- An array is lifted by replicating it along the missing dimensions
- E.g., if A is a rank 1 array it is indexed as $A[i]$
- When looking for $A[i][j]$, the value of $A[i]$ is returned regardless of j
- And that's why arrays are compatible if the shape is a prefix

$$2 + \begin{bmatrix} 3 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Another Wrinkle: Lifting and Currying

$$\left(+ \begin{bmatrix}1\\2\end{bmatrix} \begin{bmatrix}3&4&5\\6&7&8\end{bmatrix}\right) = \left(+ \left(\begin{bmatrix}1\\2\end{bmatrix} \begin{bmatrix}3&4&5\\6&7&8\end{bmatrix}\right)\right) \quad \text{grouping args for currying}$$

$$= \left(\begin{bmatrix}1+\\2+\end{bmatrix} \left(\begin{bmatrix}3&4&5\\6&7&8\end{bmatrix}\right)\right) \quad \text{curry the first arg}$$

$$= \left(\begin{bmatrix}1+&1+&1+\\2+&2+&2+\end{bmatrix} \left(\begin{bmatrix}3&4&5\\6&7&8\end{bmatrix}\right)\right) \quad \text{lift the operator}$$

$$= \left(\begin{bmatrix}4&5&6\\8&9&10\end{bmatrix} ()\right) \quad \text{apply the operator}$$

$$= \begin{bmatrix}4&5&6\\8&9&10\end{bmatrix} \quad \text{final result}$$

Another Wrinkle: Lifting and Currying

$$
\begin{aligned}
(+\,[1\,2]\,[[3\,4\,5]\;[6\,7\,8]]) &= (+\,([1\,2]\,[[3\,4\,5]\;[6\,7\,8]])) && \text{grouping args for cu} \\
&= ([1+\;\;2+]\;([[3\,4\,5]\;[6\,7\,8]])) && \text{curry the f} \\
&= ([[1+\,1+\,1+]\,[2+\,2+\,2+]]\;([[3\,4\,5]\;[6\,7\,8]])) && \text{lift the op} \\
&= ([[4\,5\,6]\;[8\,9\,10]]\;()) && \text{apply the op} \\
&= [[4\,5\,6]\;[8\,9\,10]] && \text{fina}
\end{aligned}
$$

Outline

- Background
- Semantics of Lifting
- Reasoning about Remora Programs: First Steps
- Questions?

The Very Basics

- Initial goals were very modest: `(+ 2 3) = 5`

The Very Basics

- Initial goals were very modest: `(+ 2 3) = 5`
- Even that takes some work!
- We still have to reason about currying and error handling
- And, the evaluator is mutually recursive

Error Handling

- This is inspired by the idea of **partial correctness**

Error Handling

- This is inspired by the idea of **partial correctness**
- The idea is to propagate errors from one function into the functions it calls

Error Handling

- This is inspired by the idea of **partial correctness**
- The idea is to propagate errors from one function into the functions it calls

```
(defthm
    propagate-not-reserrp-cells-at-depth-in-expr-value-list-when-vector
    (implies (and (equal (expr-value-kind val) :vector)
                  (expr-value-wfp val)
                  (equal depth (len (dims-of-expr-value val)))
                  (not
                   (reserrp (cells-at-depth-in-expr-value val depth))))
             (not
              (reserrp
               (cells-at-depth-in-expr-value-list
                (expr-value-vector->elems val)
                (+ -1 depth)))))
  :hints ...)
```

Reasoning about Eval-App-List

- This is called with an array of (unary) operators and an array of arguments
- This function applies each operator to the corresponding argument
- Notice that **pairwise-curry-binary-fn** is a simple ACL2 function, essentially separate from the interpreter

```
(defthm eval-app-list-when-primop-binary-and-bases
  (implies (and (list-of-primop-int-binary funcells)
                (list-of-base-values argcells)
                (equal (len funcells) (len argcells))
                (not (reserrp (eval-app-list funcells argcells limit)))
           (equal (eval-app-list funcells argcells limit)
                  (pairwise-curry-binary-fn funcells argcells)))
  :hints ...)
```

Reasoning about Eval-App

- This is the main function of the interpreter
- Mutually recursive with **eval-app-list** and many others
- Its main role is to hande the currying of operators

Reasoning about Eval-App I

```
(defthm eval-app-when-funval-is-binary-op
  (implies (and (equal fun-frame (dims-of-expr-value funval))
                (expr-valuep funval)
                (expr-value-case funval :primop)
                (primop-value-case (expr-value-primop->val funval)
                                   :int-binary)
                (equal argval (car argvals))
                (expr-value-wfp argval)
                (expr-valuep argval)
                (expr-is-nested-vectors-of-base-values argval)
                (equal arg-frame (dims-of-expr-value argval))
                (equal pframe arg-frame)
                (equal funcells (lift-expr-value-to-frame
                                 funval fun-frame pframe))
                (equal argcells (lift-expr-value-to-frame
                                 argval arg-frame pframe))
                (not (reserrp (eval-app funval argvals limit)))))
```

Reasoning about Eval-App II

```
          (equal (eval-app funval argvals limit)
                 (eval-app (expr-value-with-nonempty-dims
                            pframe
                            (pairwise-curry-binary-fn funcells
                                                      argcells))
                           (cdr argvals)
                           (1- limit))))
:hints ...)
```

Continuing to the Next Argument

- The result of `eval-app` is a new call to `eval-app` with the function value of the form `(expr-value-with-nonempty-dims ...)`
- This is a regular Remora object
- It's a Remora array, whose elements come from the call to `pairwise-curry-binary-fn`

```
(equal (eval-app funval argvals limit)
       (eval-app (expr-value-with-nonempty-dims
                   pframe
                   (pairwise-curry-binary-fn funcells
                                             argcells))
                   (cdr argvals)
                   (1- limit))))
```

Continuing to the Next Argument

- At the next call to `eval-app`, the function value is of the form `(expr-value-with-nonempty-dims ...)`
- And we need to reason about how it can be merged with the next argument
- Basically, we need to split this term into the ACL2 expression that produced it

Continuing to the Next Argument I

```
(defthm undo-expr-value-with-nonempty-dims-3
  (implies (and (expr-value-listp cells)
                (expr-value-list-wfp cells)
                (pos-listp frame)
                (equal (len cells)
                       (nat-list-product frame))
                (prefixp frame pframe)
                (not (reserrp
                      (cells-at-depth-in-expr-value
                        (expr-value-with-nonempty-dims frame cells)
                        (len frame)))))
                         trerd
```

Continuing to the Next Argument II

```
          (equal (lift-expr-value-to-frame
                  (expr-value-with-nonempty-dims frame cells)
                    frame
                    pframe)
                 (repeat-each (nat-list-product
                               (nthcdr (len frame) pframe))
                              cells)
                 ))
:hints ...)
```

- Notice that the bulk of the lifting operation is done by the function `lift-expr-value-to-frame`
- `eval-app` figures out the right frame for the result and then calls `lift-expr-value-to-frame` to line up the arguments
- This lemma shows what `lift-expr-value-to-frame` does in terms of simple ACL2 functions

Eval-App of a Binary Operator with Two Arguments I

```
(defthm eval-app-when-funval-is-binary-op-fully
    (implies (and (expr-valuep funval)
                  (expr-value-case funval :primop)
                  (primop-value-case (expr-value-primop->val funval)
                                     :int-binary)
                  (expr-value-wfp arg1)
                  (expr-valuep arg1)
                  (expr-value-wfp arg2)
                  (expr-valuep arg2)
                  (expr-is-nested-vectors-of-base-values arg1)
                  (expr-is-nested-vectors-of-base-values arg2)
                  (equal arg1-frame (dims-of-expr-value arg1))
                  (equal arg2-frame (dims-of-expr-value arg2))
                  (equal (list-prefix-join (list arg1-frame arg2-frame)
                         (mv t pframe))
                  (not
                   (reserrp (eval-app funval (list arg1 arg2) limit))))
```

Eval-App of a Binary Operator with Two Arguments II

```
          (equal (eval-app funval (list arg1 arg2) limit)
                 (expr-value-with-nonempty-dims
                  pframe
                  (pairwise-binary-args-fn
                   (repeat-each (nat-list-product pframe)
                                (get-cells-from-expr-value funval))
                   (repeat-each (nat-list-product
                                 (nthcdr (len arg1-frame) pframe))
                                (get-cells-from-expr-value arg1))
                   (repeat-each (nat-list-product
                                 (nthcdr (len arg2-frame) pframe))
                                (get-cells-from-expr-value arg2)))))
:hints ...)
```

Conclusions

- There is value in reasoning about Remora programs
- But the semantics of Remora is quite complicated

Conclusions

- There is value in reasoning about Remora programs
- But the semantics of Remora is quite complicated

- Kestrel has developed a formal model of Remora in ACL2
- This model is complete with a parser and an interpreter
- The model is being used to formalize Remora compiler transformations
- It can also be used to reason about Remora programs

Conclusions

- There is value in reasoning about Remora programs
- But the semantics of Remora is quite complicated

- Kestrel has developed a formal model of Remora in ACL2
- This model is complete with a parser and an interpreter
- The model is being used to formalize Remora compiler transformations
- It can also be used to reason about Remora programs

- A key challenge in reasoning is Remora's lifting semantics
- We saw how we can address this challenge in ACL2

Outline

- Background
- Semantics of Lifting
- Reasoning about Remora Programs: First Steps
- Questions?