Remora Compiler — A Master's Project

William Stevenson

September 6, 2023

Contents

1 Introduction

Remora is a typed, array-oriented programming language, whose type system and semantics were laid out in the thesis of Justin Slepak [1]. Inspired by the likes of Iverson's APL and J, it allows for implicit lifting of functions over arrays of data, allowing for shape-polymorphic computing. Unlike APL and J, it is a strongly typed language, which is beneficial beyond just catching errors at compile time. The type system is dependent, allowing it to capture information about the shape of arrays. This shape information shows the compiler explicitly where parallelism occurs in the code. The hope is that the compiler can leverage this information to create programs that efficiently utilize a GPU. Remora is also a higher-order language, allowing operations like reduce and scan, which can also be parallelized. With these operators and the implicit lifting of functions over arrays, programmers can express highly parallel programs in a more intuitive and less error prone way. Remora's syntax is s-expression based, having a syntax that resembles Lisp, Scheme, and Racket. For more information on Remora, see [2] for a beginner's guide to the language.

The goal of this Master's Project is to create a compiler for the core of the language that produces GPU code as a stepping stone towards a more general and efficient compiler. Recursion was excluded from the core of the language, as users are expected to use higher-order functions like reduce and fold rather than relying on recursion. Additionally, higher-order usage of functions was restricted to cases where the compiler can determine the function being called statically. This paper is an interim report that discusses the work that was done towards reaching this goal, as well as what remains to be done.

2 Overview of GPU Programming

The first stage of the project involved familiarizing myself with GPUs and GPU programming, as in order to create a compiler that produces GPU code, you must know how to write GPU code. The target language was chosen to be CUDA C++, which is a C++ dialect developed by NVIDIA. CUDA C++ produces binaries that can be run on any NVIDIA CPU, which gives good portability due to the prevalence of them.

2.1 Familiarization

CUDA by Example [3] was a useful starting point for learning about GPU programming, as well as nVidia's documentation of CUDA [4]. GPUs allow a large number of threads to run simultaneously. Threads exist in a hierarchy: kernel, block, warps, and thread (moving from large- to small-scale granularity). A kernel can be thought of as a function that is run on a GPU (aka device in GPU programming) and is kicked of by code executing on the CPU (aka host). When a kernel is kicked off, the host specifies how many blocks and how many threads per block should be created. The device then queues blocks of threads to be run, and the kernel finishes once all blocks finish executing. Multiple blocks can be run at once, but the number of synchronous blocks is limited by device resources. When the GPU executes a block, it runs all of its threads. A block organized its threads into warps, which are discussed more later.

The host and device have separate memory spaces; the device cannot access memory stored on the host, and the host cannot access memory stored on device. There are three forms of memory available to the device: global memory, local memory, and registers. Global memory is memory that exists in device RAM. Any device thread can access global memory at any time, and pointers to locations may be passed around. Global memory can be copied to or from host memory. Also, a kernel cannot allocate global memory — it must be done by the host. Local memory exists on a per-block basis and is much more limited in space that global memory. When a block spawns, it may request a local memory allocation. This memory is shared across all threads in a block, but threads in different blocks do not have access to this memory. Lastly, GPUs may store data in registers, as with CPUs. Due to the large number of threads, the number of registers available on a per-thread basis is low, so the number of registers available can be a bottleneck for how many threads may run simultaneously.

When a block executes its threads, it arranged the threads into warps of size 32. All threads execute in a lockstep, SIMD way. This has two important consequences. First, MIMD parallelization is not suitable for GPUs. Consider the piece of code in Figure 1, in which the first thread of every warp executes a long running function `expensiveFunc`. Since all threads in a warp execute together, as the first thread executes `expensiveFunc`, the other 31 threads need to go along for the ride (although they won't perform any side-effects while doing so). This effectively means that the code slows down by a factor of 32. Second, when all 32 threads access 32 consecutive memory slots in the same step, these memory accesses are "coalesced", meaning that they cost the same as only 1 memory access. Since memory access is frequently a bottleneck for GPUs, such as in a dot-product, vector summation, or matrix transpose, ensuring memory coalesces can frequently be the difference between horribly inefficient code and highly optimized code.

The follow is a list of important takeaways to keep in mind:

- Kernels must be launched by the host, not the device. Although newer versions of CUDA do allow the device to spawn kernels, it is less efficient and best avoided. This means that nested loops are not conducive to parallel GPU execution and need to be "flattened" by the compiler.

Figure 1: Branching on a GPU

```
if (threadId % 32 == 0) {
    expensiveFunc();
}
```

- Warps prevent MIMD execution on GPUs.
- Coalescing is extremely important. The compiler should try its best to produces code that performs coalescing memory accesses.
- Global memory cannot be allocated by the device. This can become an issue when the compiler does not know staticly how much memory needs to be allocated, such as in an `iota` call.

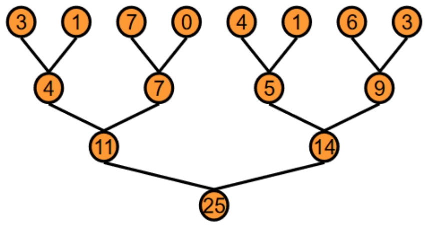
2.2 Experimentation

I also performed some experimentation with writing GPU code. While performing a `map` operation on GPU is straightforward, `reduce` is more complicated. As a result, I benchmarked various reduce algorithm implementations on different sizes of data to determine which would be the best to use for Remora.

In parallel programming, we evaluate algorithm complexity in terms of two concepts: work and span. Work is the total amount of work that needs to be done and is essentially how long the algorithm would take with only one processor. Span is the maximum work that any one thread will perform, so it represents how long the algorithm would take to run with unlimited processors.

`reduce` does not immediately jump out as a parallel operation, and it's not in every case. However, if the combining operation is associative, the operations don't need to be performed in order, allowing for a tree-like reduction (see Figure 2). This allows `reduce` to have a span of $O(f_s(n) \cdot \log n)$ (and work of $O(f_w(n) \cdot n)$), assuming the combination operation has span $O(f_s(n))$ (and work $O(f_w(n))$).

Figure 2: Parallel Reduction



NVIDIA's library Thrust provides a parallel `reduce` operator, but it also requires the combining operation to be commutative, which is not necessary for the before-mentioned asymptotic properties. The Remora standard library will have non-commutative versions of reduce, although commutative versions may be included. Thus, a goal was to implement an efficient `reduce` that does not use commutativity. The best performance reached was only 33% slower than the Thrust version. The reduction in speed is due to commutativity allowing for easy memory coalescing, whereas the non-commutative implementation needs to cache values in local memory in order to achieve coalesces.

Another case considered was performing a reduction along either rows or columns of a matrix, which has 4 cases:

1. Reduce along rows, where row length is large
2. Reduce along rows, where row length is small
3. Reduce along columns, where column length is large
4. Reduce along columns, where column length is small

This investigation assumed that matrices are stored in a row-major format. However, reducing along rows in a row-major format is the same problem as reducing along columns in a column-major format, and vice-versa. Case 1) is trivial — because rows are long and stored contiguously in memory, it is efficient to simply iterate over each row, performing a standard reduction on it. Case 2) is more complex. Initially, it is tempting to assign one thread to each row and iteratively sum elements, but this wouldn't allow for memory coalesces and is thus inefficient. However, I was able to make a version that is about 50% slower than Case 1)'s by again utilizing local memory. It is also straighforward to make case 3) efficient, as individual threads iteratively sum the columns, which produces coalesces. I have not yet explored case 4), but I believe that a similar method to Case 3) can be done.

3 Existing Work

The next phase of the project involved reading the existing literature related to the task of creating a compiler for Remora. Too many papers and articles were read to individually discuss here, but a full list, as well as summarizations of each, is available on a Notion page at https://liam-stevenson.notion.site/583922871f0c4d6a939441e9cca91c05?v=7ef79fca86ee41f58fb16ae2cc704ffd&pvs=4. The below sections discuss in some detail the most significant pieces of work on the topic.

3.1 Guy Blelloch

Guy Blelloch's book *Vector Models for Data-Parallel Computing* [5], which is a revision of his dissertation, shows a general algorithm for flattening any algorithm into one that "flattens" and utilizes all parallelization. Flattening here means getting rid of nested loops. For example, a map whose body is another map operation can be flattened into a single map, as shown in Figure 3. As mentioned before, this is important for GPU programming. However, while Blelloch's algorithm maintains asymptotic runtime characteristics, it is very inefficient in real world applications due to over-parallelizing in some cases, bad constant factors in other cases, and inefficient memory usage [6].

Figure 3: Flattening Nested Maps

```
; The following nested map
(map (λ (ys) (map f ys)) xs)
; can be flattened to
(map f xs)
; Note that this isn't technically correct code, since f would receive arrays.
; However, if the arrays nested in xs are in contiguous memory, this is a valid
; under-the-hood transformation.
```

3.2 Accelerate

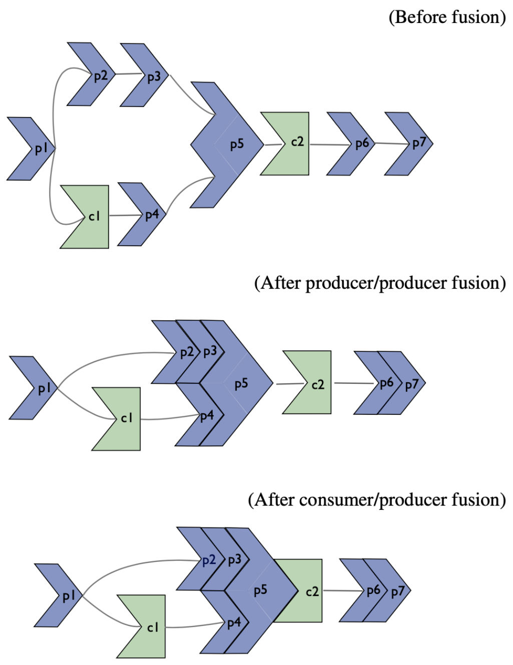
According to its website [7]:

> *Accelerate* is an embedded language of array-based computations for high-performance computing in Haskell. Programs in Accelerate are expressed in the form of parameterised collective operations on regular multidimensional arrays, such as maps, reductions, and permutations. Accelerate then takes these computations and optimises, compiles, and executes them for the chosen target architecture.

Accelerate has some limitations. For one, it does not have `filter` in it. It also is restricted to regular, flat parallelism. As previously mentioned, "flat" means that nested loops are not allowed. "Regular" means that all data being syncronously processed is of the same shape. An example of irregular parallelism would be performing a map over a 2-d array where the inner arrays have different size.

The paper *Optimising purely functional GPU programs* [8] explains how Accelerate performs fusion to eliminate intermediate arrays when performing computations. The paper divides all operations into producers and consumers. In producers, all elements of the output rely at most on one element of the input (map, iota, *etc.*), while consumers are others (reduce, fold, scan, *etc.*). Producer/producer fusion is done, where producers are combined together, and then consumer/producer fusion is done, where consumers acting on producers are combined (see below diagram). It is noted that producer/consumer would also be possible, but that isn't handled in this paper. This fusion is demonstrated in Figure 4.

Figure 4: Parallel Reduction



3.3 Futhark

According to its website [9]:

> Futhark is a small programming language designed to be compiled to efficient parallel code. It is a statically typed, data-parallel, and purely functional array language in the ML family, and comes with a heavily optimising ahead-of-time compiler that presently generates either GPU code via CUDA and OpenCL, or multi-threaded CPU code.

Futhark was created by Troels Henriksen for his PhD thesis [6] and is an ongoing research project. It is the state-of-the-art in compiling a functional language to efficient GPU code. In many cases, it produces GPU code that is competitive with hand-written, optimized GPU code. While it is a powerful language, it does have some limitations. Mainly, since global memory cannot be allocated from the device, sometimes a type-safe program will fail to compile because the compiler wants to allocate memory from within a kernel. Also, like Remora's core, which is the focus of this project, it outlaws recursion.

Unlike Accelerate, Futhark is able to support irregular parallelism through support of segmented arrays. It also supports nested parallelism, which it is able to do through an algorithm called "Incremental Flattening". To flatten a program, Futhark first aggressively inlines function calls, which is possible due to a lack of recursion. It then looks for flattening opportunities, such as a map of a map, a reduce of a map, or a scan of a map. When it finds such cases, the compiler makes a choice. Based on the size of the array (which Futhark may know at compile time, due to having a dependent type system), the compiler chooses to either flatten the parallelism or only keep the inner or outer parallelism and do the other loop iteratively. If the size is not known at compile time, a runtime check is inserted to determine which way to go. This avoids the pitfalls of Guy Blelloch's method.

Futhark also does a number of other optimizations in order to achieve good performance, including fusion, loop tiling, and transposing matrices to ensure coalesced memory accesses.

4 Compiler

The language of choice for implementing the compiler was OCaml. The project is available on GitHub at https://github.com/liam923/remora, which has a README describing how to run it. The project is currently in flux, but this section describes the state of it as of the time of writing (September 6, 2023).

The compiler is split into stages, which are described in the below sections. Each stage's code can be found in `./lib/Stages`, and intermediate data structures can be found in `./lib/IntermediateLanguages`. There are also supporting files in `./lib`.

To see some example outputs from the compiler, see Appendix A.1. For some example programs, it shows the output after each stage.

4.1 Parse

The parsing stage is fairly standard and is split into two sub-stages.

First, the program is parsed into a list of s-expressions. The s-expressions used in Remora are an extension of the standard s-expression syntax found in languages like Lisp, and Figure 5 shows an Ocaml definition of the extended s-expressions (although it is simplified — the real definition also contains fields for tracking the source of tokens). A list has an additional field that says whether the brackets generating the list were parentheses or square brackets. This is important in Remora because unlike in Racket, there is a semantic difference between the two. There is also an additional form called `WithCurlies`. This captures expressions like `foo{a | b}`, which is used for some syntactic sugar that is discussed below. The parsing of s-expressions is performed using Menhir and OCamllex.

Figure 5: Extended S-Expression Definition

```
type braceType =
  | Paren
  | Square

type t =
  | List of
      { braceType : braceType
      ; elements : t list
      }
  | WithCurlies of
      { base : 's t
      ; leftElements : t list
      ; rightElements : t list
      }
  | String of string
  | Integer of int
  | Symbol of string
```

Second, the list of s-expressions is parsed into an AST by performing pattern matching on the s-expressions.

The syntax supported is dictated by two sources: *The Semantics of Rank Polymorphism* [10] and *Introduction to Rank-polymorphic Programming in Remora* [2]. I also added an additional syntactic sugar that makes explicitly-typed polymorphic Remora code more succinct. The syntax defined by these sources is overly verbose, making it hard to read, when defining or using polymorphic functions. Compare the verbose syntax in Figure 6 to the sugared syntax in Figure 7.

The rules governing this sugar are straightforward. There are two use cases: in definitions and in usages. For both, type arguments go before the `|` and index arguments go after. The bar must always be there even if there are just type arguments or just index arguments in order to distinguish the cases. In definitions, the presence of at least one type argument causes a type lambda to be inserted with those type arguments, and the same is true for indices. The index lambda goes on the outside if both

Figure 6: Simple Polymorhpic Definition and Usage — No Sugar

```
; Note the verboseness of defining index and type variables
(define length (i-fn (d @cell-shape) (t-fn (t) (fn ([arr [t d @cell-shape]]) ...))))
; and of instantiating an instance of length
((t-app (i-app length 3 [2]) int) [[1 2] [3 4] [5 6]])
```

Figure 7: Simple Polymorhpic Definition and Usage — With Sugar

```
; Defining index and type variables is similar to many popular languages
(define (length{t | d @cell-shape} [arr [t d @cell-shape]]) ...)
; as is the usage of length
(length{t | d @cell-shape} [[1 2] [3 4] [5 6]])
```

are present. In usages, the system is similar. The presence of at least one type argument causes the expression to be wrapped in a type application, and the same is true for indices. If both are present, the index application goes on the inside, which corresponds to the index lambda being on the outside.

4.1.1 An Aside on Type Verbosity

Types in Remora can quickly become very large in size. For an example, see the type of `reduce` in Figure 8. To help with this issue, the intention is to eventually give Remora a type inference engine so that programmers do not need to write types explicitly. With type inference, the explicitly typed version of Remora would act as an intermediate form to the compiler rather than be user facing. This would render the aforementioned sugar unnecessary, since types would rarely be written. However, since this type inference engine does not yet exist, I put in this sugar so that current examples and tests are more readable, as the `t-app` and `i-app` forms quickly become confusing in moderately-complex code.

Figure 8: Type of `reduce`

```
(Π (d-1 @item-pad @cell-shape) ( (t)
    (→ ((→ ([t @cell-shape] [t @cell-shape]) [t @cell-shape])
        [t (+ d-1 1) @item-pad @cell-shape])
       [t @item-pad @cell-shape])))
```

4.2 Type Check

The type checker is also straightforward since the typing rules are explicitly laid out in *The Semantics of Rank Polymorphism* [10]. However, some conveniences that are described in *Introduction to Rank-polymorphic Programming in Remora* [2] were implemented in the type-checking stage. Namely, implicit lifting of atoms to arrays. Whenever an atomic type `t` appears where an array type is expected, `t` is replaced with `(Arr t [])`. Similarly, whenever an atomic expression `e` appears where an array expression is expected, `e` is replaced with `(array [] e)`.

4.3 Explicitize

This stage of the compiler is so named because it makes explicit all of the maps that Remora implicitly includes at call sites. The result is a form where all function arguments are cells, and all of the implicit shape-lifting is reified in the map forms. The compiler treats the map form as a control syntax, not a primitive operation. It is then free to perform various optimizations on the maps down the line, such as fusion and flattening. An expression like `(+ [1 2] [[3 4 5] [6 7 8]])` is transformed to:

```
(map [2] (λ ([a int] [b [int 3]])
           (map [3] (λ (b2 int)
                      (+ a b2))
                    b))
         [1 2]
         [[3 4 5] [6 7 8]])
```

The map form is worth briefly elaborating on. First, the map is allowed to take multiple arguments. Second, the map takes an argument called the "frame shape" ([2] and [3] in the above example). The frame shape is the dimensions of the argument that are mapped over, allowing the map to map over multiple dimensions at once. As an example, `(+ [[1] [2]] [[3] [4]])` is transformed to:

```
(map [2 1]
     (λ ([a int] [b int])
       (+ a b))
     [[1] [2]]
     [[3] [4]])
```

Note that these transformed programs are actually a simplified form of what is produced. For implementation reasons, and to handle the case of the function being a non-scalar array, the full transformation of `(+ [1 2] [[3 4 5] [6 7 8]])` is:

```
(map [] (λ ([f1 (-> (int int) int)] [a1 [int 2]] [b1 [int 2 3]])
          (map [2] (λ ([a2 int] [b2 [int 3]])
                     (map [3] (λ ([b3 int])
                                (f1 a2 b3))
                          b2))
                   a1
                   b1)
  +
  [1 2]
  [[3 4 5] [6 7 8]])
```

Note that a map with frame shape [] is equivalent to a let. In fact, this compiler pass also transforms all lets into maps with an empty frame shape in order to avoid repeated code in later stages.

4.4 Inline

The inline stage presented the largest challenge to implement. It simultaneously monomorphizes the program while inlining all non-primitive function calls. It is so called because monomorphization can be thought of as inlining type and index applications. The produced code has no term, type, or index applications, besides primitive operations, as well as no term, type, or index lambdas. To reflect this, there are also no `->`, `Forall`, and `Pi` types.

Note that inlining all function calls and type applications *does not* remove all lambdas and type abstractions from the code, as they can remain in positions where they are not called or applied. For example, `(define f (λ ([x int]) x))` remains the same after inlining. However, after inlining, it is guaranteed that no lambda or type abstraction will later be used. Therefore, it is safe to replace them with a dummy value. In this implementation, the compiler replaces them with a `unit` atom. Correspondingly, it replaces `->`, `Forall`, and `Pi` types with the type `Unit`.

At the end of this transformation, there are no remaining type variables. Some index variables, however, do remain. This is because index variables can arise from unbox expressions, and removing these in the general case would require runtime information unavailable at compile time. However, none of the remaining index variables arise from index lambdas, since there are no more index lambdas.

Fully inlining all function calls (besides primitives) is possible because no recursion is allowed, as well as one restriction that was added to the language. This restriction is that the compiler must be able to determine what function is in every function call position, and there cannot be multiple functions in the function call position (this can happen if you have an array of functions). Because of the aggressive inling and lack of recursion, the compiler can determine the function in most common use cases. Further, having an array of different functions is not supported for a second reason, which is that it represents MIMD parallelism. Since MIMD parallelism does not correspond well to GPUs, this project avoids it.

Also, because of full inlining, the issue of monomorphizing higher-rank polymorphism became a non-issue. There was, however, one restriction that needed to be added to the language. This was ML's value restriction, which says that a variable with polymorphic type must have a syntactic value as its value [11]. Remora expands this to also say that polymorphic function arguments must have a syntactic value. The value restriction ensures that monomorphization does not cause duplicated computations.

The following is a hand-wavy explanation of how inlining happens:

The structure of the algorithm is a recursive transformation of the program. As the tree is worked down, a stack keeps track of what type and index applications have been made. When a type and index lambda is hit, the stack is popped, and the popped value is substituted into the body of the lambda.

Additionally, an environment is kept. When a let expression is hit, an entry into the environment is made, where the key is the variable identifier and the entry contains the polymorphic value being bound to the variable. The entry also contains a cache, which will be discussed later.

Furthermore, a substitution set is kept, which is a mapping from variable identifiers to variable identifiers. This will also be discussed later.

In addition to returning an inlined version of an expression, a "function set" is returned. This is the set of all functions that the expression could contain as a value. For a lambda or primitive, this is simply the lambda or primitive. For a frame, it is the union of all elements of the frame. For a variable, this is looked up in the cache.

When a variable reference is hit, it is first substituted based on the substitution set. Then, it is looked up in the environment. If there is no entry, the polymorphic value stored in the environment is inlined using the current application stack. The result is also stored in the cache, along with a newly generated identifier and the function set returned when inlining. A reference to that newly generated identifier is returned. If there is already a cached entry, the identifier in the cache entry is returned. After the body of a let is inlined, the cached is iterated over. For each entry, an new variable is created, with the identifier and value coming from the cache.

When a term application is hit, the value in the function call position is inlined. The returned value is discarded, and the function set is looked at. If it has 0 entries, an error occurred. If it has multiple entries, an error is returned to the user, as this is a functionality that is not currently supported. If it is just one entry, then we're in business. If that entry is a primitive operator, that is dealt with specially. If it is a lambda expression, a let expression is created that binds all the arguments to newly created variables. The body of the let is then the inlined body of the lambda, where the substitution set is extended with entries that map parameters to the newly created variables.

4.5 Simplify

This stage performs a number of standard optimizations on the program. These are:

- Copy propagation
- If variable is used exactly once, inline it
- If a variable's value is a value (a literal or another variable), inline it
- If a variable isn't used, delete it
- Constant folding
- Hoist variable declarations out of loops when possible
- Hoist expressions out of loops when possible
- Remove maps with no arguments and empty frame

Some of these optimizations rely on Remora not allowing side-effects. However, this leads to some issues, because Remora does allow exceptions to be thrown. For example, expressions are hoisted out of loops whenever they do not use any variables defined in the loop, which causes the expression to be evaluated before the loop is called. This is done even if the loop can run 0 times. Thus, it is possible for hoisting to cause programs to raise exceptions when they would not otherwise. The removal of unused variables and inlining of variables used once can cause the reverse issue.

5 Next Steps

The below steps are what I believe need to be taken to achieve the goal of implementing a compiler for Remora's core that produces GPU code:

1. Fusion — Fuse together producers and consumers, similarly to how either Accelerate or Futhark does it. This will avoid creating intermediate arrays where possible.
2. Flattening — Flatten nested parallelism. This will likely look like Futhark's Incremental Flattening system.
3. Convert to IR — After flattening, it is possible to convert the program into GPU-like code. An IR should be developed that represents GPU code. Futhark has such an IR, so it might be a good idea to use its IR and let Futhark take it from there. If possible, this would save a lot of work because it would presumably give us a lot of optimizations for free and produce a lot of different code targets (such as CUDA, OPENGL, and Numpy).
4. Convert to CUDA — The IR should be translated into CUDA code.

Beyond this, there is still a large amount of work to create a compiler that produces efficient CUDA code, and even more to support the full Remora language beyond just its core. Below are what I believe should be the next steps after the core compiler is created. The steps are in the order that I think to be most sensible. If all steps are taken, I believe that Remora would be a complete, efficient language:

1. Loop tiling — Loop tiling can offer large performance gains on the GPU. It is an important technique in creating an optimized matrix multiplication, for example.
2. Transpose matrices — Futhark sometimes transposes matrices under the hood to get more memory coalesces.
3. Optimizations — Perform more complex standard optimizations on the program, like frame-map interchanging.
4. Box analysis — Unbox expressions don't *need* to handle arbitrary indices. In many cases, it's possible to narrow down the possible indices to a small set. An analysis could be performed that finds the possible values and compiles different versions of the unbox expression for those values. Then, a runtime check is performed to determine what path to go down. Since there is no recursion, this analysis should be straightforward to perform.
5. Optimize array operations — Operations like transpose, rotate, and subarray might not need to produced reified arrays.
6. Support MIMD — Add support for MIMD operations by allowing arrays of different functions in the function call position.
7. Support recursion — Supporting recursion would make the language much more powerful, but also would make the compiler much trickier to implement.
8. Add Type Inference — Type inference would make the language much more usable

6 References

[1] J. Slepak, “A typed programming language: The semantics of rank polymorphism.,” Ph.D. dissertation, Northeastern University, 2020. [Online]. Available: https://ccs.neu.edu/~jrslepak/Dissertation.pdf.

[2] O. Shivers, J. Slepak, and P. Manolios, *Introduction to rank-polymorphic programming in remora (draft)*, 2020. arXiv: 1912.13451 [cs.PL].

[3] J. Sanders, E. Kandrot, and J. J. Dongarra, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley/Pearson Education, 2015.

[4] *Cuda toolkit documentation 12.2*, 2023. [Online]. Available: https://docs.nvidia.com/cuda/.

[5] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[6] T. Henriksen, “Design and implementation of the futhark programming language,” English, Ph.D. dissertation, 2017.

[7] *Accelerate*. [Online]. Available: https://www.acceleratehs.org/documentation/users-guide/language.html.

[8] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, “Optimising purely functional gpu programs,” *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 49–60, 2013.

[9] *Why futhark?* [Online]. Available: https://futhark-lang.org/.

[10] J. Slepak, O. Shivers, and P. Manolios, *The semantics of rank polymorphism*, 2019. arXiv: 1907.00509 [cs.PL].

[11] *Value restriction*. [Online]. Available: http://mlton.org/ValueRestriction.

[12] J. Slepak, O. Shivers, and P. Manolios, “An array-oriented language with static rank polymorphism,” in *Programming Languages and Systems*, Z. Shao, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 27–46.

A Example Compilations

The following sections show the output of each compilation stage for some example programs. Note that after type checking, all expressions are also annotated with their type, but types are not included in the printout because they bloat the expression, as in Remora types are frequently very large. Also, the *Explicitize* stage creates a large, hard to follow program because it creates many redundant variables. Because of this, the *Simplify* stage can drastically shrink the program size and increase readability. For this reason, it may be worthwhile skipping directly from the *Type Check* result to the *Simplify* result.

A.1 Basic addition

Original program:

```
(+ 1 2)
```

Result of stage Type Check:

```
(TermApplication
 ((func (Primitive ((func Add))))
  (args
   ((Scalar ((element (Literal (IntLiteral 1)))))
    (Scalar ((element (Literal (IntLiteral 2)))))))))
```

Result of stage Explicitize:

```
(Map
 ((args
   (((binding ((name f) (id 21))) (value (Primitive ((func Add)))))
    ((binding ((name +arg1) (id 19)))
     (value (Scalar ((element (Literal (IntLiteral 1)))))))
    ((binding ((name +arg2) (id 20)))
     (value (Scalar ((element (Literal (IntLiteral 2)))))))))
  (body
   (TermApplication
    ((func (Ref ((id ((name f) (id 21))))))
     (args (((id ((name +arg1) (id 19)))) ((id ((name +arg2) (id 20))))))
     (type' ((element (Literal IntLiteral)) (shape ()))))))
  (frameShape ()) (type' (Arr ((element (Literal IntLiteral)) (shape ()))))))
```

Result of stage Inline and Monomorphize:

```
(IntrinsicCall
 (Map (frameShape ())
  (args
   (((binding ((name f) (id 22)))
     (value
      (Scalar
       ((element (Literal UnitLiteral))
        (type' ((element (Literal UnitLiteral)) (shape ())))))))
    ((binding ((name +arg1) (id 23)))
     (value
      (Scalar
       ((element (Literal (IntLiteral 1)))
        (type' ((element (Literal IntLiteral)) (shape ())))))))
    ((binding ((name +arg2) (id 24)))
     (value
      (Scalar
       ((element (Literal (IntLiteral 2)))
        (type' ((element (Literal IntLiteral)) (shape ()))))))))))
  (body
   (PrimitiveCall
    ((op Add)
     (args
      ((Ref
        ((id ((name +arg1) (id 23)))
         (type' ((element (Literal IntLiteral)) (shape ())))))
       (Ref
        ((id ((name +arg2) (id 24)))
         (type' ((element (Literal IntLiteral)) (shape ())))))))
     (type' ((element (Literal IntLiteral)) (shape ()))))))
  (type' ((element (Literal IntLiteral)) (shape ())))))
```

Result of stage Simplify:

```
(Scalar
 ((element (Literal (IntLiteral 3)))
  (type' ((element (Literal IntLiteral)) (shape ())))))
```

A.2 Addition with Implicit Map

Original program:

```
(+ [1 2] [[3 4 5] [6 7 8]])
```

Result of stage Type Check:

```
(TermApplication
 ((func (Primitive ((func Add))))
  (args
   ((Frame
     ((dimensions (2))
      (elements
       ((Scalar ((element (Literal (IntLiteral 1)))))
        (Scalar ((element (Literal (IntLiteral 2)))))))))
    (Frame
     ((dimensions (2))
      (elements
       ((Frame
         ((dimensions (3))
          (elements
           ((Scalar ((element (Literal (IntLiteral 3)))))
            (Scalar ((element (Literal (IntLiteral 4)))))
            (Scalar ((element (Literal (IntLiteral 5)))))))))
        (Frame
         ((dimensions (3))
          (elements
           ((Scalar ((element (Literal (IntLiteral 6)))))
            (Scalar ((element (Literal (IntLiteral 7)))))
            (Scalar ((element (Literal (IntLiteral 8))))))))))))))))
```

Result of stage Explicitize:

```
(Map
 ((args
   (((binding ((name f) (id 21))) (value (Primitive ((func Add)))))
    ((binding ((name +arg1) (id 19)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar ((element (Literal (IntLiteral 1)))))
          (Scalar ((element (Literal (IntLiteral 2))))))))))))
    ((binding ((name +arg2) (id 20)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Frame
           ((dimensions (3))
            (elements
             ((Scalar ((element (Literal (IntLiteral 3)))))
              (Scalar ((element (Literal (IntLiteral 4)))))
              (Scalar ((element (Literal (IntLiteral 5)))))))))
          (Frame
           ((dimensions (3))
            (elements
             ((Scalar ((element (Literal (IntLiteral 6)))))
              (Scalar ((element (Literal (IntLiteral 7)))))
              (Scalar ((element (Literal (IntLiteral 8))))))))))))))))
  (body
   (Map
    ((args
      (((binding ((name +arg1) (id 22)))
        (value (Ref ((id ((name +arg1) (id 19)))))))
       ((binding ((name +arg2) (id 23)))
        (value (Ref ((id ((name +arg2) (id 20))))))))
     (body
      (Map
       ((args
         (((binding ((name +arg2) (id 24)))
           (value (Ref ((id ((name +arg2) (id 23)))))))))
        (body
         (TermApplication
          ((func (Ref ((id ((name f) (id 21))))))
           (args
            (((id ((name +arg1) (id 22)))) ((id ((name +arg2) (id 24))))))
           (type' ((element (Literal IntLiteral)) (shape ()))))))
        (frameShape ((Add ((const 3) (refs ())))))
        (type'
         (Arr
          ((element (Literal IntLiteral))
           (shape ((Add ((const 3) (refs ())))))))))))
     (frameShape ((Add ((const 2) (refs ())))))
     (type'
      (Arr
       ((element (Literal IntLiteral))
        (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ()))))))))))
  (frameShape ())
  (type'
   (Arr
    ((element (Literal IntLiteral))
     (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ())))))))))
```

Result of stage Inline and Monomorphize:

```
(IntrinsicCall
 (Map (frameShape ())
  (args
   (((binding ((name f) (id 25)))
     (value
      (Scalar
       ((element (Literal UnitLiteral))
        (type' ((element (Literal UnitLiteral)) (shape ())))))))
    ((binding ((name +arg1) (id 26)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element (Literal (IntLiteral 1)))
            (type' ((element (Literal IntLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (IntLiteral 2)))
            (type' ((element (Literal IntLiteral)) (shape ())))))))
        (type'
         ((element (Literal IntLiteral))
          (shape ((Add ((const 2) (refs ()))))))))))
    ((binding ((name +arg2) (id 28)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Frame
           ((dimensions (3))
            (elements
             ((Scalar
               ((element (Literal (IntLiteral 3)))
                (type' ((element (Literal IntLiteral)) (shape ())))))
              (Scalar
               ((element (Literal (IntLiteral 4)))
                (type' ((element (Literal IntLiteral)) (shape ())))))
              (Scalar
               ((element (Literal (IntLiteral 5)))
                (type' ((element (Literal IntLiteral)) (shape ())))))))
            (type'
             ((element (Literal IntLiteral))
              (shape ((Add ((const 3) (refs ())))))))))
          (Frame
           ((dimensions (3))
            (elements
             ((Scalar
               ((element (Literal (IntLiteral 6)))
                (type' ((element (Literal IntLiteral)) (shape ())))))
              (Scalar
               ((element (Literal (IntLiteral 7)))
                (type' ((element (Literal IntLiteral)) (shape ())))))
              (Scalar
               ((element (Literal (IntLiteral 8)))
                (type' ((element (Literal IntLiteral)) (shape ())))))))
            (type'
             ((element (Literal IntLiteral))
              (shape ((Add ((const 3) (refs ())))))))))))
        (type'
         ((element (Literal IntLiteral))
          (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ())))))))))))
  (body
   (IntrinsicCall
    (Map (frameShape ((Add ((const 2) (refs ())))))
     (args
      (((binding ((name +arg1) (id 27)))
        (value
         (Ref
          ((id ((name +arg1) (id 26)))
           (type'
            ((element (Literal IntLiteral))
             (shape ((Add ((const 2) (refs ())))))))))))
       ((binding ((name +arg2) (id 29)))
        (value
         (Ref
          ((id ((name +arg2) (id 28)))
           (type'
            ((element (Literal IntLiteral))
             (shape
              ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ())))))))))))))
     (body
      (IntrinsicCall
       (Map (frameShape ((Add ((const 3) (refs ())))))
        (args
         (((binding ((name +arg2) (id 30)))
           (value
            (Ref
             ((id ((name +arg2) (id 29)))
              (type'
               ((element (Literal IntLiteral))
                (shape ((Add ((const 3) (refs ())))))))))))))
        (body
         (PrimitiveCall
          ((op Add)
           (args
            ((Ref
              ((id ((name +arg1) (id 27)))
               (type' ((element (Literal IntLiteral)) (shape ())))))
             (Ref
              ((id ((name +arg2) (id 30)))
               (type' ((element (Literal IntLiteral)) (shape ()))))))))
           (type' ((element (Literal IntLiteral)) (shape ()))))))
        (type'
         ((element (Literal IntLiteral))
          (shape ((Add ((const 3) (refs ())))))))))))
     (type'
      ((element (Literal IntLiteral))
       (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ())))))))))))
  (type'
   ((element (Literal IntLiteral))
    (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ()))))))))))
```

Result of stage Simplify:

```
(IntrinsicCall
 (Map (frameShape ((Add ((const 2) (refs ())))))
  (args
   (((binding ((name +arg1) (id 27)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element (Literal (IntLiteral 1)))
            (type' ((element (Literal IntLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (IntLiteral 2)))
            (type' ((element (Literal IntLiteral)) (shape ()))))))))
        (type'
         ((element (Literal IntLiteral))
          (shape ((Add ((const 2) (refs ()))))))))))))
    ((binding ((name +arg2) (id 29)))
     (value
      (Frame
       ((dimensions (2 3))
        (elements
         ((Scalar
           ((element (Literal (IntLiteral 3)))
            (type' ((element (Literal IntLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (IntLiteral 4)))
            (type' ((element (Literal IntLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (IntLiteral 5)))
            (type' ((element (Literal IntLiteral)) (shape ()))))
          (Scalar
           ((element (Literal (IntLiteral 6)))
            (type' ((element (Literal IntLiteral)) (shape ()))))
          (Scalar
           ((element (Literal (IntLiteral 7)))
            (type' ((element (Literal IntLiteral)) (shape ()))))
          (Scalar
           ((element (Literal (IntLiteral 8)))
            (type' ((element (Literal IntLiteral)) (shape ()))))))
        (type'
         ((element (Literal IntLiteral))
          (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ())))))))))))
  (body
   (IntrinsicCall
    (Map (frameShape ((Add ((const 3) (refs ())))))
     (args
      (((binding ((name +arg2) (id 30)))
        (value
         (Ref
          ((id ((name +arg2) (id 29)))
           (type'
            ((element (Literal IntLiteral))
             (shape ((Add ((const 3) (refs ())))))))))))))
     (body
      (PrimitiveCall
       ((op Add)
        (args
         ((Ref
           ((id ((name +arg1) (id 27)))
            (type' ((element (Literal IntLiteral)) (shape ())))))
          (Ref
           ((id ((name +arg2) (id 30)))
            (type' ((element (Literal IntLiteral)) (shape ())))))))
        (type' ((element (Literal IntLiteral)) (shape ()))))))
     (type'
      ((element (Literal IntLiteral)) (shape ((Add ((const 3) (refs ())))))))))
  (type'
   ((element (Literal IntLiteral))
    (shape ((Add ((const 2) (refs ()))) (Add ((const 3) (refs ()))))))))
```

A.3 Box and Unbox

Original program:

```
(define words
  (boxes (len) [char len] [2]
    ((3) "hey" )
    ((2) "hi" )))

(unbox words (word len)
  (= 3 (length{char | len []} word)))
```

Result of stage Type Check:

```
Result of stage Type Check:
(Let
 ((binding ((name words) (id 19)))
  (value
   (Frame
    ((dimensions (2))
     (elements
      ((Scalar
        ((element
          (Box
           ((indices ((Dimension ((const 3) (refs ())))))
            (body
             (Frame
              ((dimensions (3))
               (elements
                ((Scalar ((element (Literal (CharacterLiteral h)))))
                 (Scalar ((element (Literal (CharacterLiteral e)))))
                 (Scalar ((element (Literal (CharacterLiteral y))))))))))
            (bodyType
             (Arr
              ((element (Literal CharacterLiteral))
               (shape ((Add ((const 0) (refs ((((name len) (id 20)) 1))))))))))))))
       (Scalar
        ((element
          (Box
           ((indices ((Dimension ((const 2) (refs ())))))
            (body
             (Frame
              ((dimensions (2))
               (elements
                ((Scalar ((element (Literal (CharacterLiteral h)))))
                 (Scalar ((element (Literal (CharacterLiteral i))))))))))
            (bodyType
             (Arr
              ((element (Literal CharacterLiteral))
               (shape ((Add ((const 0) (refs ((((name len) (id 20))
               ↪  1))))))))))))))))))))
  (body
   (Unbox
    ((indexBindings (((name len) (id 21))))
     (valueBinding ((name word) (id 22)))
     (box (Ref ((id ((name words) (id 19))))))
     (body
      (TermApplication
       ((func (Primitive ((func Equal))))
        (args
         ((Scalar ((element (Literal (IntLiteral 3)))))
          (TermApplication
           ((func
             (TypeApplication
              ((tFunc
                (IndexApplication
                 ((iFunc (Primitive ((func Length))))
                  (args
                   ((Dimension ((const 0) (refs ((((name len) (id 21)) 1)))))
                    (Shape ()))))))
               (args ((Atom (Literal CharacterLiteral)))))))
            (args ((Ref ((id ((name word) (id 22))))))))))))))))))
```

Result of stage Explicitize:

```
(Map
 ((args
   (((binding ((name words) (id 19)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element
             (Box
              ((indices ((Dimension ((const 3) (refs ())))))
               (body
                (Frame
                 ((dimensions (3))
                  (elements
                   ((Scalar ((element (Literal (CharacterLiteral h)))))
                    (Scalar ((element (Literal (CharacterLiteral e)))))
                    (Scalar ((element (Literal (CharacterLiteral y))))))))))
               (bodyType
                (Arr
                 ((element (Literal CharacterLiteral))
                  (shape
                   ((Add ((const 0) (refs ((((name len) (id 20)) 1))))))))))))))
          (Scalar
           ((element
             (Box
              ((indices ((Dimension ((const 2) (refs ())))))
               (body
                (Frame
                 ((dimensions (2))
                  (elements
                   ((Scalar ((element (Literal (CharacterLiteral h)))))
                    (Scalar ((element (Literal (CharacterLiteral i))))))))))
               (bodyType
                (Arr
                 ((element (Literal CharacterLiteral))
                  (shape
                   ((Add ((const 0) (refs ((((name len) (id 20))
                   ↪  1)))))))))))))))))))))))
  (body
   (Unbox
    ((indexBindings (((name len) (id 21))))
     (valueBinding ((name word) (id 22)))
     (box (Ref ((id ((name words) (id 19))))))
     (body
      (Map
       ((args
         (((binding ((name f) (id 27))) (value (Primitive ((func Equal)))))
          ((binding ((name =arg1) (id 23)))
           (value (Scalar ((element (Literal (IntLiteral 3)))))))
          ((binding ((name =arg2) (id 26)))
           (value
            (Map
             ((args
               (((binding ((name f) (id 25)))
                 (value
                  (TypeApplication
                   ((tFunc
                     (IndexApplication
                      ((iFunc (Primitive ((func Length))))
                       (args
                        ((Dimension
                          ((const 0) (refs ((((name len) (id 21)) 1)))))
                         (Shape ()))))))
                    (args ((Atom (Literal CharacterLiteral))))))))
                ((binding ((name length-arg) (id 24)))
                 (value (Ref ((id ((name word) (id 22)))))))))
              (body
               (TermApplication
                ((func (Ref ((id ((name f) (id 25))))))
                 (args (((id ((name length-arg) (id 24))))))
                 (type' ((element (Literal IntLiteral)) (shape ()))))))
              (frameShape ())
              (type' (Arr ((element (Literal IntLiteral)) (shape ()))))))))))
        (body
         (TermApplication
          ((func (Ref ((id ((name f) (id 27))))))
           (args
            (((id ((name =arg1) (id 23)))) ((id ((name =arg2) (id 26))))))
           (type' ((element (Literal BooleanLiteral)) (shape ()))))))
        (frameShape ())
        (type' (Arr ((element (Literal BooleanLiteral)) (shape ())))))))))))
  (frameShape ())
  (type'
   (Arr
    ((element (Literal BooleanLiteral))
     (shape ((Add ((const 2) (refs ())))))))))))
```

Result of stage Inline and Monomorphize:

```
(IntrinsicCall
 (Map (frameShape ())
  (args
   (((binding ((name words) (id 31)))
     (value
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element
             (Box
              ((indices ((Dimension ((const 3) (refs ())))))
               (body
                (Frame
                 ((dimensions (3))
                  (elements
                   ((Scalar
                     ((element (Literal (CharacterLiteral h)))
                      (type'
                       ((element (Literal CharacterLiteral)) (shape ())))))
                    (Scalar
                     ((element (Literal (CharacterLiteral e)))
                      (type'
                       ((element (Literal CharacterLiteral)) (shape ())))))
```

```
          (Scalar
           ((element (Literal (CharacterLiteral y)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ()))))))))
        (type'
         ((element (Literal CharacterLiteral))
          (shape ((Add ((const 3) (refs ())))))))))))
     (bodyType
      ((element (Literal CharacterLiteral))
       (shape
        ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))
     (type'
      ((parameters (((binding ((name len) (id 20))) (bound Dim))))
       (body
        ((element (Literal CharacterLiteral))
         (shape
          ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))))
  (type'
   ((element
     (Sigma
      ((parameters (((binding ((name len) (id 20))) (bound Dim))))
       (body
        ((element (Literal CharacterLiteral))
         (shape
          ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
    (shape ())))))
(Scalar
 ((element
   (Box
    ((indices ((Dimension ((const 2) (refs ())))))
     (body
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element (Literal (CharacterLiteral h)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (CharacterLiteral i)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ()))))))))
        (type'
         ((element (Literal CharacterLiteral))
          (shape ((Add ((const 2) (refs ())))))))))))
     (bodyType
      ((element (Literal CharacterLiteral))
       (shape
        ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))
     (type'
      ((parameters (((binding ((name len) (id 20))) (bound Dim))))
       (body
        ((element (Literal CharacterLiteral))
         (shape
          ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))))
  (type'
          ((element
            (Sigma
             ((parameters (((binding ((name len) (id 20))) (bound Dim))))
              (body
               ((element (Literal CharacterLiteral))
                (shape
                 ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
           (shape ())))))))
      (type'
       ((element
         (Sigma
          ((parameters (((binding ((name len) (id 20))) (bound Dim))))
           (body
            ((element (Literal CharacterLiteral))
             (shape ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
        (shape ((Add ((const 2) (refs ())))))))))))))
(body
 (Unbox
  ((indexBindings (((name len) (id 21))))
   (boxBindings
    (((binding ((name word) (id 32)))
      (box
       (Ref
        ((id ((name words) (id 31)))
         (type'
          ((element
            (Sigma
             ((parameters (((binding ((name len) (id 20))) (bound Dim))))
              (body
               ((element (Literal CharacterLiteral))
                (shape
                 ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
           (shape ((Add ((const 2) (refs ())))))))))))))
   (body
    (IntrinsicCall
     (Map (frameShape ())
      (args
       (((binding ((name f) (id 28)))
         (value
          (Scalar
           ((element (Literal UnitLiteral))
            (type' ((element (Literal UnitLiteral)) (shape ())))))))
        ((binding ((name =arg1) (id 29)))
         (value
          (Scalar
           ((element (Literal (IntLiteral 3)))
            (type' ((element (Literal IntLiteral)) (shape ())))))))
        ((binding ((name =arg2) (id 34)))
         (value
          (IntrinsicCall
           (Map (frameShape ())
            (args
             (((binding ((name f) (id 30)))
               (value
                (Scalar
                 ((element (Literal UnitLiteral))
                    (type' ((element (Literal UnitLiteral)) (shape ()))))))))
                ((binding ((name length-arg) (id 33)))
                 (value
                  (Ref
                   ((id ((name word) (id 32)))
                    (type'
                     ((element (Literal CharacterLiteral))
                      (shape
                       ((Add ((const 0) (refs ((((name len) (id 21))
                        ↪  1)))))))))))))))))
              (body
               (IntrinsicCall
                (Length
                 (arg
                  (Ref
                   ((id ((name length-arg) (id 33)))
                    (type'
                     ((element (Literal CharacterLiteral))
                      (shape
                       ((Add ((const 0) (refs ((((name len) (id 21)) 1))))))))))))
                 (t (Literal CharacterLiteral))
                 (d ((const 0) (refs ((((name len) (id 21)) 1)))))
                 (cellShape ())
                 (type' ((element (Literal IntLiteral)) (shape ())))))))
              (type' ((element (Literal IntLiteral)) (shape ()))))))))))
        (body
         (PrimitiveCall
          ((op Equal)
           (args
            ((Ref
              ((id ((name =arg1) (id 29)))
               (type' ((element (Literal IntLiteral)) (shape ())))))
             (Ref
              ((id ((name =arg2) (id 34)))
               (type' ((element (Literal IntLiteral)) (shape ()))))))))
           (type' ((element (Literal BooleanLiteral)) (shape ())))))))
        (type' ((element (Literal BooleanLiteral)) (shape ())))))))
     (type'
      ((element (Literal BooleanLiteral))
       (shape ((Add ((const 2) (refs ())))))))))))
  (type'
   ((element (Literal BooleanLiteral)) (shape ((Add ((const 2) (refs ()))))))))))
```

Result of stage Simplify:

```
(Unbox
 ((indexBindings (((name len) (id 21))))
  (boxBindings
   (((binding ((name word) (id 32)))
     (box
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element
             (Box
              ((indices ((Dimension ((const 3) (refs ())))))
               (body
      (Frame
       ((dimensions (3))
        (elements
         ((Scalar
           ((element (Literal (CharacterLiteral h)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (CharacterLiteral e)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (CharacterLiteral y)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))))
        (type'
         ((element (Literal CharacterLiteral))
          (shape ((Add ((const 3) (refs ())))))))))))
     (bodyType
      ((element (Literal CharacterLiteral))
       (shape
        ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))
     (type'
      ((parameters (((binding ((name len) (id 20))) (bound Dim))))
       (body
        ((element (Literal CharacterLiteral))
         (shape
          ((Add ((const 0) (refs ((((name len) (id 20)) 1))))))))))))))
  (type'
   ((element
     (Sigma
      ((parameters (((binding ((name len) (id 20))) (bound Dim))))
       (body
        ((element (Literal CharacterLiteral))
         (shape
          ((Add ((const 0) (refs ((((name len) (id 20)) 1))))))))))))
    (shape ()))))))
(Scalar
 ((element
   (Box
    ((indices ((Dimension ((const 2) (refs ())))))
     (body
      (Frame
       ((dimensions (2))
        (elements
         ((Scalar
           ((element (Literal (CharacterLiteral h)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))
          (Scalar
           ((element (Literal (CharacterLiteral i)))
            (type'
             ((element (Literal CharacterLiteral)) (shape ())))))))
        (type'
         ((element (Literal CharacterLiteral))
          (shape ((Add ((const 2) (refs ())))))))))))
```

```
             (bodyType
              ((element (Literal CharacterLiteral))
               (shape
                ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))
             (type'
              ((parameters (((binding ((name len) (id 20))) (bound Dim))))
               (body
                ((element (Literal CharacterLiteral))
                 (shape
                  ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))))
          (type'
           ((element
             (Sigma
              ((parameters (((binding ((name len) (id 20))) (bound Dim))))
               (body
                ((element (Literal CharacterLiteral))
                 (shape
                  ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
            (shape ())))))))
      (type'
       ((element
         (Sigma
          ((parameters (((binding ((name len) (id 20))) (bound Dim))))
           (body
            ((element (Literal CharacterLiteral))
             (shape ((Add ((const 0) (refs ((((name len) (id 20)) 1)))))))))))
        (shape ((Add ((const 2) (refs ())))))))))))))
(body
 (PrimitiveCall
  ((op Equal)
   (args
    ((Scalar
      ((element (Literal (IntLiteral 3)))
       (type' ((element (Literal IntLiteral)) (shape ())))))
     (IntrinsicCall
      (Length
       (arg
        (Ref
         ((id ((name word) (id 32)))
          (type'
           ((element (Literal CharacterLiteral))
            (shape ((Add ((const 0) (refs ((((name len) (id 21)) 1)))))))))))
       (t (Literal CharacterLiteral))
       (d ((const 0) (refs ((((name len) (id 21)) 1))))) (cellShape ())
       (type' ((element (Literal IntLiteral)) (shape ()))))))))
   (type' ((element (Literal BooleanLiteral)) (shape ()))))))
(type'
 ((element (Literal BooleanLiteral)) (shape ((Add ((const 2) (refs ())))))))))
```

B Readings

B.1 Vector Models for Data-Parallel Computing [5]

poop