Remora Compiler — A Master's Project

William Stevenson

September 12, 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 Introduction to Remora

This section provides a quick overview of Remora. For a much better, more detailed introduction that is quite readable, see *An Introduction to Rank-polymorphic Programming in Remora* [2].

2.1 Rank-Polymorphism

Remora is rank-polymorphic, a programming model invented by Kenneth Iverson when he created APL [3]. In this model, all values are arrays. When you assign a scalar value like `7` to a variable `x`, `x`

is thought of as a scalar array — that is, an array of rank 0. `x` could instead have a value of `[1 2]`, making it an array of rank 1, or a value of `[[1 2] [3 4]]`, making it an array of rank 2, *etc.*

In Remora, function parameters are specified to be of an expected rank, and it produces a value that also has an expected rank. These are the function's "cells", and the expected ranks are statically included as part of the function's definition. For example, `+` has arguments of rank 0 and a return value of rank 0. `dot-product`, on the other hand, has arguments of rank 1 and a return value of rank 0, since is expects two arrays and returns a scalar array.

Functions arguments are allowed to have any rank greater than or equal to the rank of the parameter they correspond to. If the argument rank is greater than the parameter rank, the function is mapped over the argument's cells. For example, consider the function `succ`, which takes an integer array of rank 0 and produces an array of rank 0 by adding `1`. `(succ 1)` would evaluate to `(succ 2)`. `(succ [1 2])`, which is equivalent to `[(succ 1) (succ 2)]`, would evaluate to `[2 3]`.

Mapping functions over the cells of its arguments is also allowed on functions with multiple arguments. Consider `(+ [1 2] [3 4])`. The two arguments are mapped over simultaneously, so this is equivalent to `[(+ 1 3) (+ 2 4)]`, which evaluates to `[4 6]`. A case like `(+ [[1 2 3] [4 5 6]] [7 8])` is also legal. In this, the cells of the second argument are replicated so that the shapes match. Thus, `(+ [[1 2 3] [4 5 6]] [7 8])` is equivalent to `(+ [[1 2 3] [4 5 6]] [[7 7 7] [8 8 8]])`, which is the same as `[[(+ 1 7) (+ 2 7) (+ 3 7)] [(+ 4 8) (+ 5 8) (+ 6 8)]]`, which evaluates to `[[8 9 10] [12 13 14]]`. For a more thorough explanation of this, see the sections *Functions operate on "cells" of input* and *Functions distribute over a frame of cells* of *An Introduction to Rank-polymorphic Programming in Remora* [2].

2.2 Type System

Remora is a typed language, and its type system uses dependent typing to allow users to specify the shape of arrays. The type of an array thus includes both the type of the elements of the array (`int`, `bool`, *etc.*) and the shape of an array. For example, the array `[[1 2 3] [4 5 6]]` has type `[int 2 3]`, since it is a 2x3 array of integers. `[#t #f]` has type `[bool 2]`. `#t` has type `bool` (the `[]` can be dropped for scalar arrays). Function types are written as `(→ (arg-types...) return-type)`. Notice that since a function's includes information about the expected shapes of arguments, the rank of the function's cells can be determined from its type.

Types can be abstracted over with "type-variables". Type variables are variables that can appear in types and stand in for other types. They are expressed using a forall expression, as in the type `(∀ (@t) (→ (@t) @t))`, which is the type of a polymorphic identity function. The `@` symbol in the variable name is important — it indicates to Remora that the expected type is the type of an array. Without the `@`, it would expect an atomic type, like `int` or `bool`, as in the type `(∀ (t) (→ ([t 3]) t))`, which might be the type of a function that extracts the first element of a three element array (note that the return type `t` is shorthand for `[t]`, a scalar array with entries of type `t`). To create a type-polymorphic value, you can write an expression with the syntax `(Tλ (`*`tvar ...`*`) e)`. The specified type variables will then be available in the expression *e*. A `Tλ` can be used with the syntax `(t-app` *`t-lambda t-args ...`*`)`.

Similarly, array shapes can be abstracted over with "index-variables". They are expressed using a pi expression, as in the type `(Π (n) (→ ([int n] [int n]) int))`, which might be the type of a function that computes the dot product of two arrays. Notice that since both arrays are statically guaranteed to be of the same length (`n`)! Similarly to type-variables, index-variable names can optionally begin with an `@`. In the case of index-variables, the absence of a `@` indicates that the variable represents a single dimension, while the presence of a `@` indicates that the variable represents a *shape* (a sequence of dimensions). To create an index-polymorphic value, you can write an expression with the syntax `(Iλ (`*`ivar ...`*`) e)`. The specified index variables will then be available in the expression *e*. A `Iλ` can be used with the syntax `(i-app` *`i-lambda i-args ...`*`)`.

Some operations are also allowed to be performed on index variables. Two dimension indices are allowed to be added together to produce a new dimension index, and two shape indices are allowed to be appended together. For example, the type of a function that appends two rank-1 arrays together may be `(Π (a b) (→ ([int a] [int b]) [int (+ a b)]))`. However, these are the limits of Remora's dependent type system. Notably, multiplying two dimensions together is forbidden.

With this under our belts, we can now look at some example function definitions. Below is `double`, which doubles an integer:

```
(define double (λ ([x int])
                 (+ x x)))
```

Below is `id`, which simply returns the given value:

```
(define id (Tλ (@t)
             (λ ([x @t])
               x)))
```

Below is `dot-product`, which computes the dot product of two vectors (assuming we have the function `sum` at our disposal):

```
(define dot-product (Iλ (n)
                      (λ ([x [int n]] [y [int n]])
                        (sum (+ x y)))))
```

There is one last major element of Remora's type system: boxes. Boxes exist in Remora for two reasons.

First, sometimes you may have an array of arrays where the length of the inner arrays is not uniform. As an example, consider the array `[[0] [0 1] [0 1 2]]`. It would be incorrect to give it the type `[int 3 1]`, `[int 3 2]`, or `[int 3 3]`. In fact, `[[0] [0 1] [0 1 2]]` is not a legal expression in Remora! To express such a value, boxes must be used.

Second, sometimes it is impossible to statically determine the size of an array. Maybe the program asks for user input that determines the length of an array. Or maybe Remora's dependent type system isn't powerful enough to express the shape of an array. Consider the function `filter-zero`, which takes an argument of type `[int 10]` (an array of 10 integers) and removes all zeroes occurring in it. What would be the shape return type? We cannot say, because it could have a length anywhere from 0 to 10.

Boxes are to Remora arrays what pointers are to C arrays, although they are quite a bit different. Boxes are not necessarily implemented as pointers (this decision has not yet been made). More importantly, boxes have a existential type that captures index variables. Let's consider the below example:

```
(boxes (len) [char len] [5]
  ((6) "Monday")
  ((7) "Tuesday")
  ((9) "Wednesday")
  ((8) "Thursday")
  ((6) "Friday"))
```

In Remora, strings are a syntactic sugar for an array of characters (`"Monday"` has type `[char 6]`). In this example, we want to create an array of arrays of characters, where each array of characters is the name of a day of the week. Since the name lengths are not uniform, we need to use boxes. This expression has type `[(Σ (len) [char len]) 5]`. That is, an array of 5 elements, where each element has type `(Σ (len) [char len])`. `Σ` is an existential type. `(Σ (len) [char len])` can be understood as, "There exists some index `len` such that the element has type `[char len]` (an array of characters with length `len`)". In the `boxes` expression, the meaning of the different elements are as follows:

- `(len)` — We are capturing the index variable `len`. Like with `Iλ`, we could use a `@` to indicate that the variable is a shape rather than a dimension.
- `[char len]` — The boxed elements will have type `[char len]`, for some `[char len]`.
- `[5]` — This is the shape of the array that contains the boxes.

- Last are individual expressions that represent the boxes. For each, we first specify what value `len` will have for that boxed value.

The example of `[[0] [0 1] [0 1 2]]` could be represented as follows:

```
(boxes (len) [int len] [3]
  ((1) [0])
  ((2) [0 1])
  ((3) [0 1 2]))
```

We can then look inside boxes using an unbox expression (using the function polymorphic `length` function that is found in the standard library):

```
(unbox weekdays (day len)
  (= 6 ((t-app (i-app length len []) char) day)))
```

This would evaluate to `[#t #f #f #f #t]`.

2.3 Parallelism

Mapping of functions over the cells of arguments can be done completely in parallel. In the prior example of `(+ [[1 2 3] [4 5 6]] [7 8])`, separate threads or processors can independently compute `(+ 1 7)`, `(+ 2 7)`, `(+ 3 7)`, *etc.* Even more importantly, this parallelization is made explicit statically by the type system, enabling the compiler to parallelize the operation.

2.4 Higher-Order

Remora is a higher-order language. It allows functions to be treated as first-class objects, and the standard library provides higher-order functions like `reduce`, `fold`, and `filter`. (Note the absence of `map` — function calls in Remora provide `map` for free.)

Using these higher-order functions is strongly encouraged in Remora over using a recursive approach because they make the structure of the program explicit to the compiler. Consider a function that sums together the elements of an array. One way to write it is using a recursive approach. Below is such an implementation, written in pseudo-Remora:

```
(define sum-recursive (λ ([arr [int n]])
                        (if (> (length arr) 0)
                            (+ (head arr) (rest arr))
                            0)))
```

Below is such an implementation written using the `reduce` function:

```
(define sum-reduce (λ ([arr [int n]])
                     (reduce + arr)))
```

`reduce` is an operation that can be performed in parallel (see 3.2 for more information), so the compiler is able to easily make `sum-reduce` efficient on a parallel architecture. However, doing so with `sum-recursive` would require complicated analysis of the code.

2.5 Example - Polynomial Evaluation

This section will show an implementation of a function that efficiently evaluates a polynomial at a specified value on a serial processor. To do so, a few standard library functions need to be introduced:

- `reduce/zero` - This performs a reduction, but an initial element is given. This is opposed to `reduce`, which doesn't take an initial value, but requires the array to be non-empty.
- `open-scan/zero` - A scan is similar to a reduction, except that it returns all intermediate values. For example, `(open-scan/zero * 1 [2 3 4])` would return `[1 2 6 24]`. The *open* refers to the result including the initial value as the first element of the array. (`(scan/zero * 1 [2 3 4])` returns `[2 6 24]`.)
- `with-shape` - This function creates an array with the shape of the first argument, replicating the second argument for each entry.

Below is the definition of `poly-eval`. For a more detailed explanation of the implementation, see [2]:

```
(define poly-eval (Iλ (l) (λ ([coeffs l] [x 0])
    ; x-powers is x^0, x^1, x^2, ...
    (define x-powers (open-scan/zero * 1
                                     (with-shape coeffs x)))
    ; Multiply the coefficients by their corresponding x-power,
    ; and then sum them up
    (reduce/zero + 0
                 (* coeffs x-powers)))))
```

3 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.

3.1 Familiarization

CUDA by Example [4] was a useful starting point for learning about GPU programming, as well as nVidia's documentation of CUDA [5]. 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.

Figure 1: Branching on a GPU

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

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.
- 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 statically how much memory needs to be allocated, such as in an `iota` call.

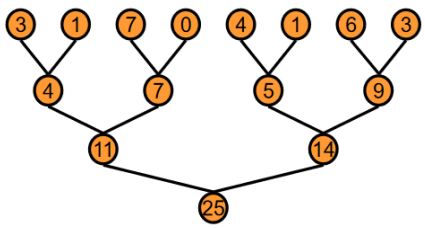
3.2 Experimentation

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 bench-marked 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 straightforward 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.

4 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 some comments on each, is available in Appendix B. The below sections discuss in some detail the most significant pieces of work on the topic.

4.1 Guy Blelloch

Guy Blelloch's book *Vector Models for Data-Parallel Computing* [6], 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 [7].

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.
```

4.2 Accelerate

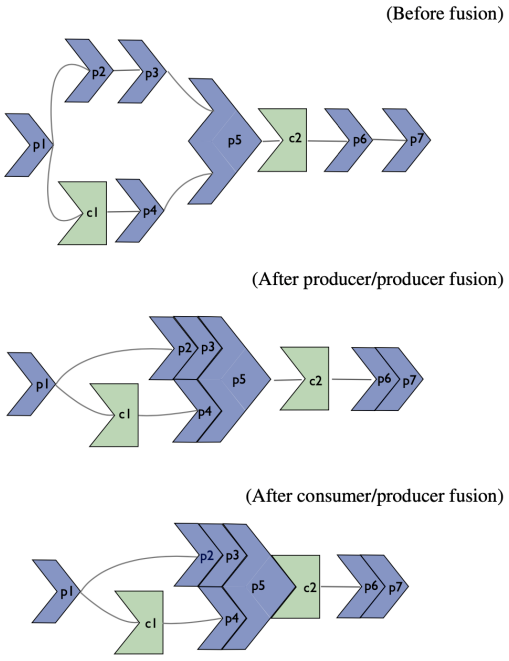
According to its website [8]:

> *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 synchronously 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* [9] 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



4.3 Futhark

According to its website [10]:

> 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 [7] 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.

5 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 12, 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.

5.1 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.

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* [11] 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 are present. In usages, the system is similar. The presence of at least one type argument causes the

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
```

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]])
```

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.

5.1.1 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])))
```

5.2 Type Check

The type checker is also straightforward since the typing rules are explicitly laid out in *The Semantics of Rank Polymorphism* [11]. 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)`.

5.3 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.

5.4 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 [12]. 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.

5.5 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.

6 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

7 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] K. E. Iverson, *A Programming Language*. USA: John Wiley & Sons, Inc., 1962, ISBN: 0471430145.

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

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

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

[7] T. Henriksen, "Design and implementation of the futhark programming language," Ph.D. dissertation, University of Copenhagen, Universitetsparken 5, 2100 KÃ¸benhavn, Nov. 2017.

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

[9] 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.

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

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

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

[13] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a portable nested data-parallel language," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 4–14, 1994, ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1994.1038. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731584710380.

[14] T. Henriksen and C. E. Oancea, "A t2 graph-reduction approach to fusion," in *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, ser. FHPC '13, Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 47–58, ISBN: 9781450323819. DOI: 10.1145/2502323.2502328. [Online]. Available: https://doi.org/10.1145/2502323.2502328.

[15] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional gpu programs," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13, Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 49–60, ISBN: 9781450323260. DOI: 10.1145/2500365.2500595. [Online]. Available: https://doi.org/10.1145/2500365.2500595.

[16] J. Reppy and J. Wingerter, λ_{cu} — *an intermediate representation for compiling nested data parallelism*, Presented at the *Compilers for Parallel Computing Workshop (CPC '16)*, Valladolid, Spain, Jul. 2016.

[17] T. Henriksen, K. F. Larsen, and C. E. Oancea, "Design and gpgpu performance of futhark's redomap construct," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2016, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 17–24, ISBN: 9781450343848. DOI: 10.1145/2935323.2935326. [Online]. Available: https://doi.org/10.1145/2935323.2935326.

[18] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea, "Apl on gpus: A tail from the past, scribbled in futhark," in *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, ser. FHPC 2016, Nara, Japan: Association for Computing Machinery, 2016, pp. 38–43, ISBN: 9781450344333. DOI: 10.1145/2975991.2975997. [Online]. Available: https://doi.org/10.1145/2975991.2975997.

[19] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 556–571, ISBN: 9781450349888. DOI: 10.1145/3062341.3062354. [Online]. Available: https://doi.org/10.1145/3062341.3062354.

[20] T. Henriksen, *Streaming combinators and extracting flat parallelism*, 2017. [Online]. Available: https://futhark-lang.org/blog/2017-06-25-futhark-at-pldi.html.

[21] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 53–67, ISBN: 9781450362252. DOI: 10.1145/3293883.3295707. [Online]. Available: https://doi.org/10.1145/3293883.3295707.

[22] R. W. Larsen and T. Henriksen, "Strategies for regular segmented reductions on gpu," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, ser. FHPC 2017, Oxford, UK: Association for Computing Machinery, 2017, pp. 42–52, ISBN: 9781450351812. DOI: 10.1145/3122948.3122952. [Online]. Available: https://doi.org/10.1145/3122948.3122952.

[23] T. Henriksen, *How futhark manages gpu memory*, 2018. [Online]. Available: https://futhark-lang.org/blog/2018-01-28-how-futhark-manages-gpu-memory.html.

[24] T. Henriksen, *Why futhark (sometimes) goes wrong*, 2018. [Online]. Available: https://futhark-lang.org/blog/2018-12-08-why-futhark-sometimes-goes-wrong.html.

[25] J. S. Bertelsen, "Implementing a cuda backend for futhark," M.S. thesis, University of Copenhagen, 2019.

[26] T. Henriksen, *Giving programmers what they want, or what they ask for*, 2019. [Online]. Available: https://futhark-lang.org/blog/2019-01-13-giving-programmers-what-they-want.html.

[27] T. Henriksen, *Incremental flattening for nested data parallelism on the gpu*, 2019. [Online]. Available: https://futhark-lang.org/blog/2019-02-18-futhark-at-ppopp.html.

[28] T. Henriksen, *What is the minimal basis for futhark?* 2019. [Online]. Available: https://futhark-lang.org/blog/2019-04-10-what-is-the-minimal-basis-for-futhark.html.

[29] T. Henriksen, *Higher-order parallel programming*, 2020. [Online]. Available: https://futhark-lang.org/blog/2020-05-03-higher-order-parallel-programming.html.

[30] T. Henriksen, *How futhark represents values at runtime*, 2021. [Online]. Available: https://futhark-lang.org/blog/2021-08-02-value-representation.html.

[31] P. Munksgaard, *Array short-circuiting*, 2022. [Online]. Available: https://futhark-lang.org/blog/2022-11-03-short-circuiting.html.

[32] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea, "Memory optimizations in an array language," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22, Dallas, Texas: IEEE Press, 2022, ISBN: 9784665454445.

[33] T. Henriksen, *In-place mapping and the pleasure of beautiful code nobody will ever see*, 2022. [Online]. Available: https://futhark-lang.org/blog/2022-12-06-in-place-map.html.

[34] 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 ((((name len) (id 20)) 1))))))))
          (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 ((((name len) (id 20)) 1))))))))
          (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

The following sections provide a full list of the readings for this project, as well as some informal comments/notes on them. They are organized by publish date, from oldest to newest. A more user-friendly version is available on Notion at https://liam-stevenson.notion.site/583922871f0c4d6a939441e9cca91c05?v=7ef79fca86ee41f58fb16ae2cc704ffd&pvs=4.

B.1 Vector Models for Data-Parallel Computing [6]

This book covers the same topic as Guy Blelloch's PhD thesis, and Futhark treats it like the Bible. However, it is kind of outdated.

In chapter 10, it shows a general way to convert any nested parallelism into flat parallelism. This conversion does not affect the asymptotic work or span of the program. However, this is essentially done by layers simulations of vector machines (V-RAMs), which produce bad constant factors and lots of intermediate data structures. As a result, it isn't efficient enough to be practical.

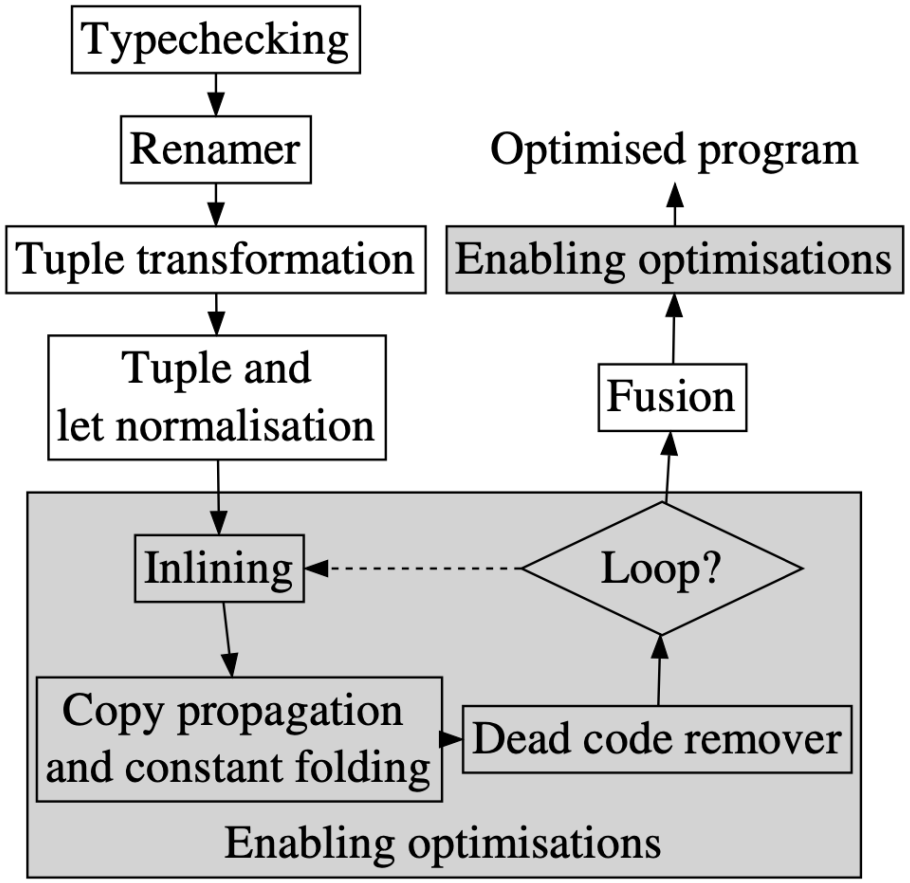
In chapter 11, it describes a compiler for Paralation Lisp using the flattening techniques described in chapter 10.

B.2 Implementation of a Portable Nested Data-Parallel Language [13]

This paper walks through the implementation of NESL, a programming language by Guy Blelloch that utilizes the techniques described in his book [6]. The code is compiled to an IR called "VCode", which is then interpreted at runtime. Thus, this is an implementation of the "V-RAM" machine described in the book. This paper is basically an implementation of the theory described in the book, so it's really only worth reading to see the benchmarks if you've read the book. The benchmarks seem to show that NESL generally runs some constant factor slower than hand-written code.

B.3 A T2 Graph-Reduction Approach to Fusion [14]

This paper walks through the implementation of L0, which seems to be a Futhark predecessor. The architecture of the compiler is diagrammed below.



The main optimization this compiler is doing is fusion, hence the title. It walks through an algorithm for fusing when possible in L0, but it was dense and I didn't comprehend it well. My main takeaway is illustrated in the below diagram. In it, variables that are used multiple times are still fused when something that depends on it depends only on it.

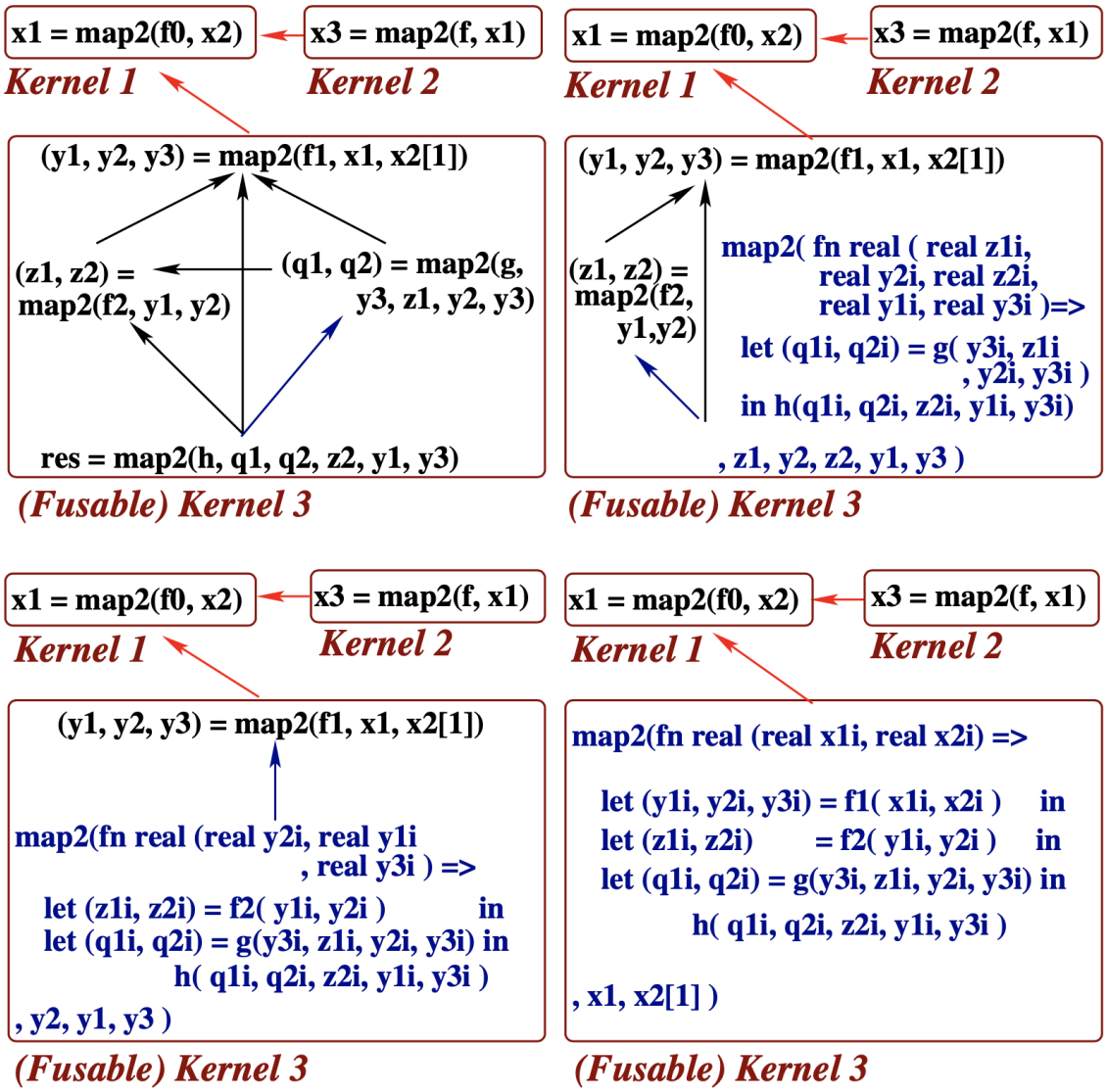


Figure 8. Fusion By T2 Transformation on the Dependency Graph

This would be akin to the below scala transformation:

```
val list: List[X] = ???
val as = list.map(f)
val bs = as.map(g)
val cs = as.zipWith(bs).map(h)
// transformed to:
val list: List[X] = ???
val (as, bs) = list.map { l =>
        val a = f(l)
        (a, g(a))
}.unzip
val cs = as.zipWith(bs).map(h)
// transformed to:
val list: List[X] = ???
val cs = list.map { l =>
        val a = f(l)
        val b = g(a)
  h((a, b))
}
```

It also shows possible fusions:

```
// replicate can be fused
// without restrictions
let x = replicate(N,a)in
let y = map2(f, x, b) in
let z = map2(g, x, c) in
let x[i] = ...
    ≡
let x = replicate(N, a) in
let y = map2( fn β1 (α1 b_i)
                => f(a,b_i), b)
let z = map2( fn β2 (α2 c_i)
                => g(a,c_i), c)
in let x[i] = ...
```

```
//map2 o map2 ⇒ map2
let (x1, x2) = map2(f, a1)
in  map2(g, x1, y)
    ≡
map2(fn β (α1 a1_i, α2 y_i)
  =>let (x1_i, x2_i) = f(a1_i)
    in  g(x1_i, y_i)
, a1, y )
```

```
//reduce2 o map2⇒redomap2
let (x1, x2) = map2(f, a1)
in  reduce2(⊕,e1,e2, x1,y)
    ≡
redomap2(⊕
, fn (β1,β2) ( β1 e1, β2 e2
             , α1 a1_i,α2 y_i)
   => let (x1_i, x2_i) = f(a1_i)
      in  ⊕(e1,e2,x1_i,y_i)
, (e1, e2), a1, y )
```

```
//redomap2 o map2⇒redomap2
let (x1, x2) = map2(f, a1)
in  redomap2(⊕, g, e, x1, y)
    ≡
redomap2(⊕
, fn β (β e, α1 a1_i, α2 y_i)
   => let (x1_i, x2_i) = f(a1_i)
      in  g(e, x1_i, y_i)
, e, a1, y )
```

```
//filter2 o filter2⇒filter2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c1,a1,a2)
in  let y = filter2(c2, x1) ..
    ≡
let (y, dead) = filter2(
  fn bool (α1 a1_i,α2 a2_i)=>
      if   c1(a1_i, a2_i)
      then c2(a1_i)
      else false
, a1, a2 ) ..
```

```
//reduce2 o filter2⇒redomap2
//IFF consumer's input list
//  ≡ producer's output list
let x = filter2(c, a)
in  reduce2(⊕, e, x)
    ≡
reduce2(fn β (β e, β a_i) =>
  if c(a_i) then ⊕(e,a_i) else e
, e, a )
```

```
//reduce2 o filter2⇒redomap2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c, a1, a2)
in  reduce2(⊕, e, x1)
    ≡
redomap2(⊕
, fn β (β e, α1 a1_i, α2 a2_i)
   => if c(a1_i, a2_i)
      then ⊕(e, a1_i) else e
, e, a1, a2 )
```

```
//redomap2 o filter2⇒redomap2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c, a1, a2)
in  redomap2(⊕, g, e, x1)
    ≡
redomap2(⊕
, fn β (β e, α1 a1_i, α2 a2_i)
   => if c(a1_i, a2_i)
      then g(e, a1_i) else e
, e, a1, a2 )
```

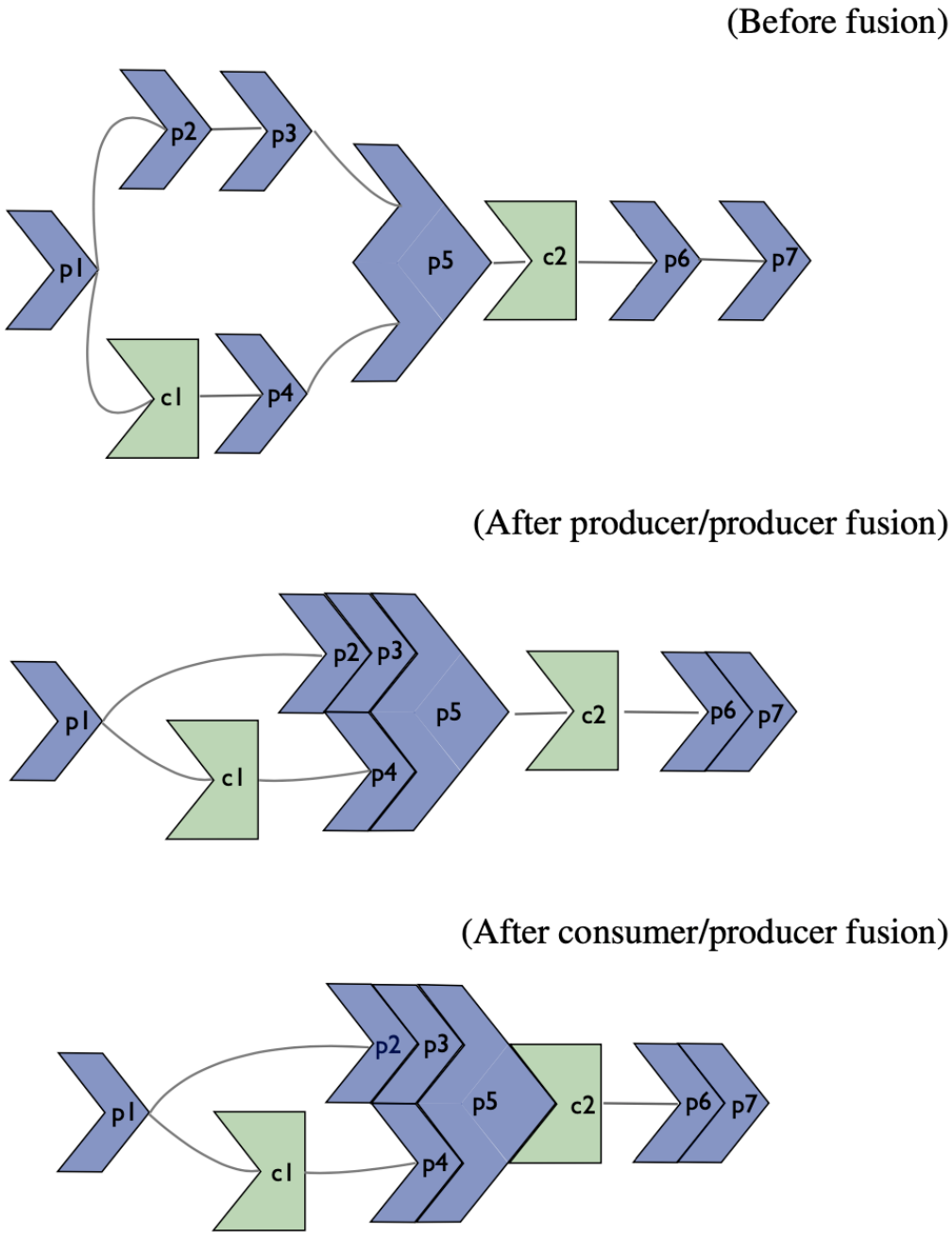

Figure 14. Compositional Algebra For Fusion

However, this requires arrays to be regular, so boxes would not be compatible with the stuff in this paper.

B.4 Optimising Purely Functional GPU Programs [15]

Half of this paper is dedicated to "sharing recovery", which is not relevant to Remora. It is used for languages embedded in Haskell, which Accelerate, the language discussed in the paper, is.

The other half is about fusion, which is very relevant. However, the language discussed, Accelerate, doesn't have filter, so considerations must be made before applying things to Remora. A point is made that let bindings are a barrier to inlining. Sometimes it is more efficient to compute a value multiple times inline rather than creating an intermediate array that can be reused, so a heuristic to decide when would be good. The paper divides all SOACs 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



B.5 λcu — An Intermediate Representation for Compiling Nested Data Parallelism [16]

This paper introduces λcu, an IR for data parallel languages. An important note is that despite the title, the IR represented in the paper does not allow for arbitrary nesting. Rather, in section 3.4 they explain that they believe this is easily achievable by adding segmented SOACs to the IR. The main part worth looking at is section 4.2 "Fusion Rules". This discusses how both horizontal and vertical fusion is possible and enumerates some of the fusion rules for the IR.

$$\textsc{Map-Map}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad B = \texttt{MAP}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ n}{B = \texttt{MAP}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ n}$$

$$\textsc{Map-Permute}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad B = \texttt{PERMUTE}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ \Lambda\ n}{B = \texttt{PERMUTE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ \Lambda\ n}$$

$$\textsc{Map-Reduce}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad b = \texttt{REDUCE}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ id_r\ \Lambda\ n}{b = \texttt{REDUCE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ id_r\ \Lambda\ n}$$

$$\textsc{Filter-Filter}\ \frac{\begin{array}{c} A = \texttt{FILTER}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ \{\ v \texttt{ => } p_1 \texttt{ using } ys\ \}\ n \\ \oplus \\ B = \texttt{FILTER}\ \{\ i \texttt{ => } e_2 \texttt{ using } zs\ \}\ \{\ v \texttt{ => } p_2 \texttt{ using } ws\ \}\ (\#A) \end{array}}{\begin{array}{l} B = \texttt{FILTER}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (zs \setminus \{A\})\ \} \\ \quad \{\ i \texttt{ => let } a = e_1 \texttt{ in } p_1\ \&\&\ [A\ \texttt{[}\,i\,\texttt{]}/a]p_2 \texttt{ using } ys \cup (ws \setminus \{A\})\ \}\ n \end{array}}$$

$$\textsc{Filter-Reduce}\ \frac{\begin{array}{c} A = \texttt{FILTER}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ \{\ w \texttt{ => } p \texttt{ using } ys\ \}\ n \\ \oplus \quad b = \texttt{REDUCE}\ \{\ i \texttt{ => } e_2 \texttt{ using } zs\ \}\ id_r\ \Lambda\ (\#A) \end{array}}{b = \texttt{REDUCE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in if } [a/w]p \texttt{ then } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ else } id_r \texttt{ using } xs \cup (ys \cup zs) \setminus \{A\})\ \}\ id_r\ \Lambda\ n}$$

Fig. 5. Selected fusion rewriting rules

B.6 Design and GPGPU Performance of Futhark's Redomap Construct [17]

This paper walks through Futhark's redomap, which is an SOAC that fuses a reduce of a map. The below snippet summarizes it:

Denoting with $\bar{z}^{(n)}$ the sequence $z_1, \ldots, z_n$, the type of the `redomap` (in the Futhark core language) was extended to:

$$(((\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \to \bar{\alpha}^{(p)}),\ ((\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}) \to (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)})),\ \bar{\alpha}^{(p)}, [\beta_1, n], \ldots, [\beta_q, n]) \quad \to \quad (\bar{\alpha}^{(p)},\ [\gamma_1, n], \ldots, [\gamma_r, n])$$

1^{st} argument is a binary associative operator, of type $(\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \to \bar{\alpha}^{(p)}$, where $\bar{\alpha}^{(p)}$ intuitively denotes a tuple of p elements,

2^{nd} argument is the folded function `g` of type: $((\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}) \to (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)}))$, That is, it receives the accumulator of type $\bar{\alpha}^{(p)}$ and an arbitrary number q of (array) elements $\bar{\beta}^{(q)}$ and produces a new accumulator $\bar{\alpha}^{(p)}$ and an arbitrary number r of (array) elements $\bar{\gamma}^{(r)}$

3^{rd} argument is the neutral element of type $\bar{\alpha}^{(p)}$,

- The remaining arguments are q arrays of equal-size outermost dimension n (i.e., $[\beta_1, n], \ldots, [\beta_q, n]$).
- The result has two components: (i) the reduced part of type $\bar{\alpha}^{(p)}$, and (ii) the `mapped` part $[\gamma_1, n], \ldots, [\gamma_r, n]$ which corresponds to r arrays of outermost size equal to n whose elements were produced by each invocation of g.

The semantics of `redomap` is:

$$\texttt{redomap}(\oplus, g, \bar{e}^{(p)}, \bar{b}^{(q)}) \equiv \texttt{let}\ (\bar{a}^{(p)},\ \bar{c}^{(r)}) = \texttt{map}(g,\ \bar{b}^{(q)})\ \texttt{in}\ (\texttt{reduce}(\oplus, \bar{e}^{(p)}, \bar{a}^{(p)}),\ \bar{c}^{(r)})$$

except that in practice it is executed very similar to the OpenMP-style parallel loop with reduction pragmas. That is, each processor computes a partial accumulator from a chunk of the mapped arrays, and the partial accumulators are then reduced across processors.

Futhark had a weaker redomap construct before that didn't operate on tuples. However, doing so allowed fusion to occur in more places.

Case 1: "fusion is allowed between two SOACs belonging to the same block of let statements even if the array produced by the first SOAC is used after the second." Ex:

```
let x = map(f, a) in
let r = reduce(+, 0.0, x)
in (r, x)
```

the `reduce` and the `map` are safely fused into:

```
let (x,r)=redomap( +
                 , fn (f32, f32) (f32 e, f32 a) =>
                     let x = f(a) in (e+x,x)
                 , 0.0, a)
in (r, x)
```

Case 2: Horizontal fusion:

Second, we allow horizontal fusion (i.e., when the two SOACs are not in a producer-consumer relation), whenever the two SOACs belong to the same block of `let` statements, their outermost sizes are equal, and as before, there is no (unfused) use of the result array in between the two SOACs. For example:

```
let x = reduce(+, 0.0, a)
let y = reduce(*, 1.0, a)
```

is fused horizontally as:

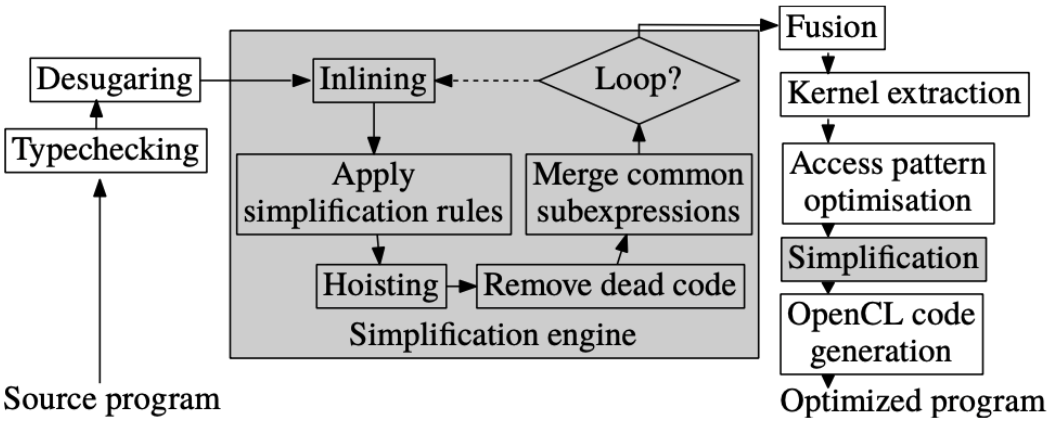
```
let (x,y)=reduce( fn ( f32, f32 ) ( f32 x1, f32 y1,
                                    f32 x2, f32 y2 ) =>
                     let r1 = x1 + x2
                     let r2 = y1 * y2
                     in (r1, r2)
                 , 0.0, a, a)
```

B.7 APL on GPUs: A TAIL from the Past, Scribbled in Futhark [18]

This walks through creating a compiler that goes APL → Tail → Futhark. This may become more relevant if we ever choose to compile to Futhark.

B.8 Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates [19]

This paper explains how Futhark is designed and implemented, as of 2017. It gives the below diagram of the compiler architecture:



The first major point discussed is Futhark's uniqueness type system that allows for in-place updates, which is not relevant to Remora.

The below table summarizes the stream operators available. Although I understand the point of `stream_red`, I don't get how `stream_map` helps performance. The point of stream operators is explained in the notes of [20].

Notation: n, m, p, q, r integers, a, b, c arrays, f function, α, β, γ types, $\oplus$ associative binop, $\bar{\#}$ vectorized array concatenation, $\overline{[n]\alpha}^{(q)}$ expands to $[n]\alpha_1, \ldots, [n]\alpha_q$, and we informally use $\overline{a[i]}^{(q)} \equiv a_1\,[i], \ldots, a_q\,[i]$ as tuple value.

$$\textbf{map}: \Pi n.(\overline{\alpha}^{(p)} \to \overline{\beta}^{(q)}) \to \overline{[n]\alpha}^{(p)} \to \overline{[n]\beta}^{(q)}$$

$$\textbf{reduce}: \Pi n.(\overline{\alpha} \to \overline{\alpha} \to \overline{\alpha}) \to \overline{\alpha} \to \overline{[n]\alpha} \to \overline{\alpha}$$

$$\textbf{scan}: \Pi n.(\overline{\alpha} \to \overline{\alpha} \to \overline{\alpha}) \to \overline{\alpha} \to \overline{[n]\alpha} \to \overline{[n]\alpha}$$

$$\textbf{stream\_map}: \Pi n.(\Pi m.\overline{[m]\beta}^{(q)} \to \overline{[m]\gamma}^{(r)}) \to \overline{[n]\beta}^{(q)} \to \overline{[n]\gamma}^{(r)}$$

$$\textbf{stream\_map}\ f\ \bar{b}^{(q)} \equiv \overline{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \overline{c_s}^{(r)}$$
where $\overline{c_i}^{(r)} = f\ (\overline{b_i}^{(q)})$
for any s-partitioning of $\bar{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)}$

$$\textbf{stream\_red}: \Pi n.(\overline{\alpha}^{(p)} \to \overline{\alpha}^{(p)} \to \overline{\alpha}^{(p)}) \to (\Pi m.\overline{\alpha}^{(p)} \to \overline{[m]\beta}^{(q)} \to (\overline{\alpha}^{(p)}, \overline{[m]\gamma}^{(r)})) \to \overline{\alpha}^{(p)} \to \overline{[n]\beta}^{(q)} \to (\overline{\alpha}^{(p)}, \overline{[n]\gamma}^{(r)})$$

$$\textbf{stream\_red}\ \oplus\ f\ (\overline{e}^{(p)})\ \bar{b}^{(q)} \equiv (\overline{e}^{(p)} \oplus \overline{a_1}^{(p)} \oplus \ldots \oplus \overline{a_s}^{(p)}, \overline{c}^{(r)})$$
where $(\overline{a_i}, \overline{c_i}) = f\ \overline{e}\ \overline{b_i}$ and $\overline{c} = \overline{c_1}\ \bar{\#} \ldots \bar{\#} \overline{c_s}$,
for any s-partitioning of $\bar{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)}$

$$\textbf{stream\_seq}: \Pi n.(\Pi m.\overline{\alpha}^{(p)} \to \overline{[m]\beta}^{(q)} \to (\overline{\alpha}^{(p)}, \overline{[m]\gamma}^{(r)}) \to \overline{\alpha}^{(p)} \to \overline{[n]\beta}^{(q)} \to (\overline{\alpha}^{(p)}, \overline{[n]\gamma}^{(r)})$$

$$\textbf{stream\_seq}\ f\ (\overline{a_0}^{(p)})\ \bar{b}^{(q)} \equiv (\overline{a_s}^{(p)}, \overline{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \overline{c_s}^{(r)})$$
where $(\overline{a_i}^{(p)}, \overline{c_i}^{(r)}) = f\ \overline{a_{i-1}}^{(p)} \overline{b_i}^{(q)}$
for any s-partitioning of $\bar{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)}$

Section 5.1 explains some of Futhark's flattening rules, which could be very helpful. Section 5.2 goes into some optimizations that ensure coalesced memory accessing using loop tiling.

B.9 Streaming Combinators and Extracting Flat Parallelism [20]

This is a blog post that puts the content of the paper [19] in a more accessible form:

This article discusses the functions `stream_red` and `stream_map`. `stream_red` is similar to reduce,

but you supply two functions: one is a function that iteratively does the reduction on a chunk of the array, and the other combines the results of the chunk reductions. Thus, it is akin to doing (reduce (map (fold ...) (chunk ...)) ...). Splitting into chunks allows the compiler/runtime to saturate the GPU while limiting parallelism overhead. The chunk size can be made 1 to turn it into a standard reduce, or N to turn it into a sequential fold.

B.10 Design and Implementation of the Futhark Programming Language [7]

This is Noels Hendriksen's PhD thesis and gives a thorough walkthrough of the Futhark compiler.

Section 4 discusses GPU hardware. It is argued that constant memory and texture memory are impractical for a compiler to use, largely due to their limited size.

It is mentioned that Futhark inlines all function calls, which it can do since it doesn't allow recursion.

A notable motivating factor behind size types in Futhark was to enable horizontal fusion.

Section 7.1 lists the set of parallel primitives that Futhark uses under the hood. These are different from the set that are available to the user because they are less user friendly, but they enable more fusion.

For the most part, fusion gets the same treatment as it does in [14]. It is noted that sometimes you have the choice between different sets of fusions. For example, you might be able to do either horizontal or vertical fusion, but not both. Doing this optimally is NP complete, and Futhark just does a greedy algorithm. It is also noted that sometimes it can be beneficial to duplicate computation, but the Futhark compiler just never does due to not wanting to implement a heuristic for it.

Chapter 8 discusses moderate flattening, which somewhat went over my held. While I get the gist of it and there are blog posts explaining it, I did not fully understand the algorithm as described here. Incremental flattening is also alluded to in this chapter, but it is said that not much research had yet been done on it. It is further discussed in [21].

The intro of chapter 9 gives some good food for thought on an IR for a GPU program.

Section 9.1 explains an optimization that is done to try to increase memory coalesces. The IR has an operation called manifest that can permute the dimensions of an array in memory (manifest((0,1), x) gives x in row major, manifest((1,0), x) gives x in column major, etc.). Kernels are inspected, and the best manifestation of each array is found for that kernel. Before the kernel is started, the arrays are manifested. A pass is done to try to catch redundant manifestations. It is said that since transpositions are fast (about the speed of a memory copy), this usually leads to an increase in performance, in some cases dramatically.

Section 9.2 discusses how loop tiling works in Futhark. Again, I don't think I fully grasped how it works, but I think it will be important to creating a performant compiler.

B.11 Strategies for Regular Segmented Reductions on GPU [22]

The title is pretty self-explanatory. The paper provides 3 strategies/algorithms for performing a segmented reduction where the segments are of equal size. The compiler generates code for all 3 strategies and chooses at runtime, based on the below heuristic:

1. If more than 2^{16} segments are present, use the sequential segments strategy.
2. Otherwise, if the segment size is greater than half the work-group size, use the large segments strategy.
3. Otherwise, use the small segments strategy.

B.12 How Futhark Manages GPU Memory [23]

GPU memory allocation is slow. Futhark uses a simple reference counting system, and initially it just allocated and freed memory on the GPU as needed. However, this became a slow point do to the slow nature of GPU allocation. Thus, they built a lightweight allocator that reuses blocks of memory that the program already allocated.

B.13 Why Futhark (Sometimes) Goes Wrong [24]

This article discussions the limitations on the Futhark compiler. Futhark sometimes fails to compile type-checked programs due to GPU limitations. It is pointed out that Guy Blelloch's flattening algorithm in theory fixes these issues, but its constant factors are so bad that it is not good in practice

B.13.1 "Cannot allocate memory block in kernel"

This happens when the compiler cannot predict how much memory a kernel will need in advance. For a Remora example, consider:

```
(def (make-box [n 0]) (box (iota n)))
(make-box (iota 10))
```

The compiler can't figure out how much space each box will need, and memory can't be allocated in the GPU kernel. I think to deal with this situation in Remora, we may need to sequentially run the make-box function, and then a kernel can be made for each cell if viable. Perhaps there is a way to segment the kernel into multiple, where a segment runs until it needs to allocate memory, tells the CPU how much memory is needed, then the CPU allocates the memory and starts another kernel that kicks of the next kernel?

B.13.2 "Cannot compile assertion inside parallel kernel"

Apparently assert statements are super bad for performance inside a GPU kernel, so the Futhark compiler doesn't allow assertions in places that would end up being in a kernel.

B.14 Implementing a CUDA Backend for Futhark [25]

This wasn't very helpful. It essentially amounted to "we slightly modified the OpenCL backend". The main thing I got from this was a very general overview of the compiler architecture, mainly that the Futhark compiler produces optimized code in an IR language, which each backend then translates into whatever language.

B.15 Giving programmers what they want, or what they ask for [26]

This article is just musing on programmers using small arrays as data structures and how the Futhark compiler should treat such things. The article makes the argument that it's better for the compiler to assume arrays will be big, and it is up to the programmer to use things like records and tuples instead.

B.16 Incremental Flattening for Nested Data Parallelism [21]

The related article sums up this paper pretty well. The paper is more technical, and it also goes into how autotuning works. Essentially, the user provides test inputs, and the program is run on them repeatedly. Various parameters that control which branches are chosen are changed randomly until a minimum is found.

B.17 Incremental flattening for nested data parallelism on the GPU [27]

This is a blog post that puts the content of the paper [21] in a more accessible form.

This article starts by defining flat parallelism and nested data parallelism. It also mentions Guy Blelloch's work and discusses why it isn't sufficient. This boils down to bad constant factors, polynomial space increase, and doing too much parallelism, which causes overhead.

B.17.1 Moderate Flattening

"Moderate flattening" is Noels's name for what the Futhark compiler does (or at least used to do). (Note: this is probably covered in previous Futhark papers, like Noels's thesis, so this might be repeated). It flattens nested parallelism, but only to an extent. He uses the example of matrix multiplication. The below Futhark code:

```
map (\xs ->
       map (\ys -> let zs = map2 (*) xs ys
                   in reduce (+) 0 zs)
           yss)
    xss
```

is transformed into:

```
map (\xs ->
       map (\ys ->
              foldl (\acc (x,y) -> acc + x * y)
                    0 (zip xs ys))
           yss)
    xss
```

When compiled, the code will exploit the parallelism of the maps but perform the summation sequentially, and it is also noted that loop tiling will be performed on the fold.

B.17.2 Incremental Flattening

With incremental flattening, the compiler produces two versions of the matrix multiplication code. The first is the moderately flattened code, as shown above. The second is a segmented reduction. At runtime, when the code is called, a check is done two see how big m and n are. If they are small enough, the segmented reduction is chosen. Choosing the threshold for m and n's sizes may be complex - the article says it needs to be "autotuned" to produce good results, but doesn't go into detail. It seems that "autotuning" might involve benchmarking the code and letting it set parameters based on that.

B.17.3 Work-Group Incremental Flattening

There's another section that discusses another alternative compilation that can be chosen at runtime, but I don't really get it. I think it's for when one block of threads can perform the entire computation on its own, which allows us to avoid intermediate arrays in some cases?

B.17.4 Remora Applications

I think we definitely need to do moderate flattening. Remora has the advantage of being able to do the incremental flattening check at compile time instead of runtime in a lot of cases (hopefully). However, tuning the parameters for when to do the different versions might be complex. The work-group optimization is probably not worth it for this initial Remora compiler, as it seems to add a lot of complexity.

B.18 What is the minimal basis for Futhark? [28]

This goes over the set of parallelism primitives that the compiler treats specially rather than being implemented in terms of other primitives. This list is:

- `flatten`
- `unflatten`
- `concat`
- `rotate`
- `transpose`
- `scatter`
- `zip`
- `unzip`
- `reduce_by_index`
- `map`
- `reduce`
- `scan`
- `partition`
- `stream_map`
- `stream_map`

B.19 The Semantics of Rank Polymorphism [11]

In this paper, Justin Slepak lays out a formal definition of the semantics of Remora, including the type system.

B.20 An Introduction to Rank-polymorphic Programming in Remora [2]

This is an introduction to Remora.

B.21 Higher-order parallel programming [29]

I put this on this list because I thought it was going to be about higher-order functions, but instead it just talks about how it is able to be more performant than NumPy.

B.22 How Futhark represents values at runtime [30]

This basically explains that Futhark represents arrays of tuples as tuples of arrays under the hood, and same for records. Remora should probably to the same thing if it supports tuples and records because it makes it easier to get coalesced memory accesses.

B.23 Array short-circuiting [31]

This is a blog post that puts the content of the paper [32] in a more accessible form.

This wasn't very relevant to remora. This is about an optimization that occurs when arrays are updated in place using uniqueness types.

B.24 Memory Optimizations in an Array Language [32]

This wasn't very relevant to remora. This is about an optimization that occurs when arrays are updated in place using uniqueness types.

B.25 In-place mapping and the pleasure of beautiful code nobody will ever see [33]

This is a blog post that is basically an addendum to the following article and paper: [31] and [32].

This wasn't very relevant to remora. This is an extension on the blog post [31].