Contents

Figures

Tables

Abstract

It is well established that declarative array languages are a high-level and expressive means of writing programs for parallel architectures. However, they are not without their disadvantages. Flat data parallel array languages are limited in their modularity as they do not support nested arrays and parallel functions cannot be called from within parallel contexts. Nested data parallel languages solve this problem but they also assume irregularity of all nested structures. This places a cost on nesting, a cost that is needlessly paid for a certain class of programs, those where nesting is strictly regular.

A second problem is that arrays, by definition, allow for random access. This means that arrays must be loaded into memory in their entirety. If memory is limited, as is the case with GPUs, array languages offer no high-level means of expressing programs containing structures too large for available memory but that do not require random access.

The research in this dissertation describes an extension to the Accelerate language: irregular array sequences. It allows for both a limited, but still useful, form of nesting and for the writing and execution of streaming programs under limited working memory.

Furthermore, in realising irregular array sequences, we describe a generalised program flattening (vectorisation) transform that does not introduce the cost on nesting that is unnecessarily incurred for strictly regular (sub)programs. This transform is applicable to a much broader domain than just array sequences.

Complementing this extension, this dissertation also describes two other extensions to Accelerate, a foreign function interface (FFI) and GPU-aware garbage collection. The former is the first instance of an embedded domain specific language having an FFI.

Chapter 1

Introduction

***TODO:** The softer part of the introduction*

This dissertation builds on existing work on Accelerate, and also array languages generally. Both prior to and during my contribution, numerous other researchers have also made many contributions. The research which I have contributed to is listed below. I explicitly state which work is my own:

- We have developed a new parallel structure in addition to the parallel array. Irregular arrays sequences offer a single level of nesting with greater modularity and controlled memory usage (Chapter 3).
- I developed a novel extension to Blelloch's flattening transform [5, 6] that identifies regular (sub)computations and transforms them according to different rules than irregular computations. By doing this I was able to take advantage of the greater efficiency afforded by regular computations, but fall back to less efficient methods when computations are irregular (Chapter 4).
- I implemented a system for treating GPU memory as a cache for GPU computations. A programmer using Accelerate with this feature does not have to be as concerned about their programs working memory fitting into the, limited, memory of the GPU (Section 6.2.
- I developed an FFI for Accelerate. This was the first, to my knowledge, FFI for an embedded language. The technique for doing this is applicable to all embedded languages with multiple execution backends. In Accelerate, it allows for highly optimised low-level libraries to be called from within a high-level Accelerate program and for Accelerate programs to be called from CUDA C programs (Section 6.1).

Before we describe any of this work, however, we need to first introduce the background and context of our research in Chapter 2.

Chapter 2

Background

Before we explore the contributions of this work, we will first give a brief overview of the concepts it is built upon and the context in which it exists, as well as its dependent technologies. More specifically, we will describe:

- The Nested Data Parallel model of programming and the concept of program flattening, or vectorisation, that underlies it (2.1).
- The CUDA framework for general purpose GPU programming (2.2).
- The Accelerate language and its implementation in Haskell (2.3).

2.1 The Nested Data Parallel Programming Model

The programming model of Nested Data Parallelism (NDP) is the most general form of data parallelism. It allows for arbitrary nesting of parallel operations within other parallel operations, as well as parallel structures which contain other parallel structures. For example, arrays of arrays. It is an expressive, high level and modular method of parallel programming [4]. A variety of different useful algorithms have been implemented in languages supporting this model [2, 3, 20].

Nested data parallelism (NDP) was first popularised by the NESL language [5], but it currently exists in a number of different languages, albeit in a restricted form in some. These include Proteus [15], Futhark [22], Nova [13], Manticore [19], and Data Parallel Haskell (DPH) [36]. In most[1] of these language this model of programming is achieved via a program transformation first described by Blelloch and Sabot [6]. Originally, NESL programs were intended to execution on vector computers. They are strictly SIMD in nature so a transform was needed that would remove nested parallel structures and nested parallel operations while preserving program behaviour. While computer hardware has changed considerably since the

[1]The exception to this is Manticore which has a execution model based on fork-join parallelism.

introduction of Nesl, the transform itself still has much utility.

Blelloch's Transform works by having two distinct components. The first being a *flattened array representation*, the second program *vectorisation*. We will address each of these in turn.

2.1.1 Array representation

Computing with nested arrays requires a means for representing the nested structure in the flat memory provided by the von Neumann model.

To give a simple example, suppose we have this nested array.

```
{ {A,B,C}, {D,E}, {F,G,H,I}, {}, {J} }
```

It is an array of arrays. The immediately obvious way of representing this is to have an array of pointers. Indeed, most contemporary languages represent nested arrays in this way. Note that this is distinct from multidimensional arrays. While one can treat, say, a 2D array as an array of arrays, the fact that each *subarray* has to be the same length does not make it a true nested array structure.

So if an array of pointers allows us to represent arbitrary nested array structures, what is the problem? It lies in the fact that modern computer hardware is optimised for locality of reference. Having different (often small) parts of the structure spread over memory with pointers between them means that any operations performed over them has to constantly chase pointers and access new areas of memory. This results in many cache misses and is unworkable in the context of high performance. In addition, allocating pointer-based structures involves numerous small allocations. This means construction and destruction is expensive, and, in the context of a garbage collector, longer GC pauses.

An alternative solution to this problem, proposed by Blelloch and Sabot [6], is to *flatten* the array structure. This means an array is split into a pair consisting of an array of *segment descriptors* and an array of values. The example above is represented as:

```
({3,2,4,0,1}, {A,B,C,D,E,F,G,H,I,J})
```

The segment descriptors in the *segments* array hold the size of corresponding subarray. So, we know that the first subarray contains the first 3 elements of the values vector, the next subarray 2, the next 4, etc.

Further nesting levels can then be represented by adding additional segment descriptors. For example:

```
{ { {A,B}, {C} },
  { {D,E}, {}, {F} } }
```

becomes,

```
({2,3}, ({2,1,2,0,1}, {A,B,C,D,E,F}))
```

The segment descriptors we have in both of these examples are what we will refer to as *size segment descriptors*, as they hold the size of subarrays. However, they are somewhat limited. Many operations with arrays in this form have worse complexity than the pointer-based equivalent. In particular, indexing, one of the most basic array operations, is no longer constant time. For example, finding the element at index 3 of the subarray at index 2 of

```
({3,2,4,0,1}, {A,B,C,D,E,F,G,H,I,J})
```

involves taking the size of all subarrays before the one at index 2 and adding them together. In this case that's $3 + 2 = 5$. Then we have to add on 3 to that, $5 + 3 = 8$. This gives us the index of `I`, which is what we want. In general, size segment descriptors require at $O(n)$ (where n is the number of subarrays) work to perform indexing.

There is another form of segment descriptor that solves this problem. Instead of holding the size of each subarray, it holds the offset. So, for our first example,

```
({0,3,5,9,9}, {A,B,C,D,E,F,G,H,I,J})
```

We will call these *offset segment descriptors*. In general, it takes $O(1)$ work to index arrays in this form as we already know where the corresponding subarray is located in the values vector.

Another useful operation, calculating the size of a given subarray, is similarly constant time. Simply take the offset of the subarray and subtract from the offset of the next subarray. However, this does not work for the final subarray and requires accessing two different elements of the segments array, which can be a problem for fusion (more on that in Chapter 5).

Instead, the better choice is *size-offset segment descriptors* where both the size and the offset are stored.

```
({(3,0),(2,3),(4,5),(0,9),(1,9)}, {A,B,C,D,E,F,G,H,I,J})
```

This has the advantages of both representations, but does require twice as much memory to store the segment descriptors.

It is worth noting that there are other, more complex, nested array representations in existence. In particular Lippmeier et al. [27] describe one that introduces the concept of *virtual segments*. They allow for arrays to be replicated without explicitly storing them more than once. This problem arises when trying to support higher-order nested parallelism, when arrays are able to store functions as values. In the context of this work, we don't support higher order nested parallelism, so we won't explore it in detail.

2.1.2 Flattening

Flattening, or vectorisation, is a syntax driven program transformation. When given a program as input it outputs a flat data parallel program which is semantically equivalent, but operates on a flattened array representation.

As an example, consider this simple program.

```
foo xs = map (map (λx → 2*x + 1)) xs
```

If we wanted to rewrite this by hand, such that it didn't rely on a nested `map` and explicitly worked with one of the nested array representation described above, we would most likely end up with this.

```
foo' xs = let (segs, vals) = xs
          in (segs, map (λx → 2*x + 1) vals)
```

In this case, we're simply mapping $2x + 1$ over the all the values in the nested array. It is trivial to see that `foo` and `foo'` are equivalent, modulo the data representation. Blelloch's flattening transformation is a method for producing these equivalent programs. It works by *lifting* functions into vector space. That is, given some function

```
f :: a → b
```

it lifts it into the function

```
f↑ :: Vector a → Vector b
```

operating over vectors. Of course if `a` or `b` are already vectors [2] then this function will be working over the nested representation of the now nested vectors.

Going back to our example, if we will start by lifting the lambda term

```
𝓛⟦λx → 2*x + 1⟧
```

For a function, we need to remember the length of the vector it should return. For all lifted functions, the length of the result should be the same as length of the argument. So we keep track of the argument as part of the *lifting context.*

```
λx → 𝓛x⟦2*x + 1⟧
```

We can then use lifted version of the binary operation `+`. We will call this `+↑`.

```
λx → 𝓛x⟦2*x⟧ +↑ 𝓛x⟦1⟧
```

For constants, we have to `replicate` that constant out over a new vector. So for `1` and `2`, we replicate it out by the length of the lifting context.

```
λx → replicate (length x) 2 *↑ 𝓛x⟦x⟧ +↑ replicate (length x) 1
```

All that leaves us with is lifting the variable `x`. In this case, no extra work needs to be done. We just leave it as `x` as it refers to the vector bound as argument to in the lambda term[3].

[2] In these examples, we're treating vectors and arrays as being the same thing. For now that is acceptable, but we will refine our definitions of both of these things in the following section.

[3] In general this is not always true. See Chapter 4 for an in-depth explanation of how variables are treated during the lifting process.

```
λx → replicate (length x) 2 *↑ x +↑ replicate (length x) 1
```

This is now the lifted version of our lambda term. Going back to the complete function.

```
foo xs = map (map (λx → 2*x + 1)) xs
```

We want to use lifting to remove the nested `map`. We can do that by first lifting the inner `map` into application of the lifted function.

```
Lift[|map (λx → 2*x + 1)|]
```

Because we're now lifting a term that is already a vector, we are lifting into the nested array representation.

```
λ(segs, vals) → (segs, ℒ⟦λx → 2*x + 1⟧ vals)
```

Essentially, we are able to replace a `map` with the application of the lifted version of its first argument to the values of its second. More generally, the *lifting rule* for `map` is

```
ℒ⟦map f⟧ = λ(segs, vals) → (segs, ℒ⟦f⟧ vals)
```

Continuing on with our example, we can substitute the result of `ℒ⟦λx →2*x + 1⟧` into the lifted version of the inner `map`.

```
λ(segs, vals) → (segs, (λx → replicate (length x) 2 *↑ x +↑ replicate (length x) 1) vals)
```

For the outer `map`, we could perform much the same thing, if we wanted a lifted version of `foo`, but if we instead just want `foo` to be *flattened* then we can replace it with the function above. More generally, this equivalence will always hold

```
map f = ℒ⟦f⟧
```

Using it we can rewrite programs with nested `map`s to remove any nesting.

A similar approach can be taken for other array combinators. All that is required is to have a version of that combinator that can operate over the segmented representation. However, this is problematic for `fold`s. If we want a parallel `fold` that assumes its first argument is associative, what does its lifting rule look like?

```
ℒ⟦fold f z arr⟧ = fold_seg f z ℒ⟦arr⟧
```

For this combinator, we do not lift the `f` or `z` arguments. What this means is that they cannot contain any nested parallelism. This is a common restriction on NDP languages.

The combinator $\texttt{fold}_{seg}$ is a necessary one to support `fold` in NDP languages. It performs a *segmented* fold. It can be thought of as performing a fold for every subvector of a vector of vectors.

```
fold_seg :: (a → a → a)
         → a
         → (Vector (Int,Int), Vector a)
         → Vector a
```

In Chapter 4 we will describe a more general form of flattening and formally specify these operations.

2.1.3 Avoidance

One extension to flattening which is important to our work is *vectorisation avoidance*, avoiding vectorisation where necessary. For example, in this simple function from above,

```
foo xs = map (map (λx → 2*x + 1)) xs
```

flattening resulted in

```
λ(segs, vals) → (segs, (λx → replicate (length x) 2 *↑ x +↑ replicate (length x) 1) vals)
```

Doing so results in many intermediate arrays. If vectorisation can be avoided for the term

```
λx → 2*x + 1
```

then it is not necessary to construct all the intermediate arrays. Keller et al. [25] describe a way of avoiding vectorisation by having a pre-vectorisation tagging phase tagging sub-computations that do not need to be vectorised. In Chapter 4 we will describe a different solution to vectorisation avoidance, but one that could be improved by taking ideas from Keller et al. [25].

2.2 CUDA

The CUDA (Compute Unified Device Architecture)[33] programming environment provides libraries and extensions to LLVM, C and C++, as well as Fortran. They enable programs to be written for and executed on NVIDIA GPUs.

The Accelerate compiler I will describe in 2.3 generates LLVM IR that is compiled by the CUDA framework. For the purposes of explanation, however, I will give examples in CUDA C as that is how the majority of programmers use the CUDA framework.

Concretely, a C programmer wanting to take advantage of a CUDA capable GPU would annotate functions in their program with special directives indicating they are intended for GPU execution. When run through the *nvcc* compiler any function annotated with these directives will be compiled into code in the Parallel Thread Execution (PTX) instruction set. These annotated functions, known as kernels, can be called with a special notation that specifies how they should be scheduled, in particular how many multiprocessors to use. The code in 2.1 shows a simple example where two vectors of size N are added together by N threads.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Listing 2.1: Adding two N length vectors in CUDA C[33]

This example corresponds to this simple Haskell function

```
vecAdd :: [Float] → [Float] → [Float]
vecAdd = zipWith (+)
```

In general, it's fairly clear how trivially parallel programs, like those that just rely on `maps` and `zipWiths`, can be implemented in CUDA C. It is less clear however, how to map other high-level combinators (for example `fold` and `scan`) to a corresponding GPU kernel. To do such things, the programmer has to concern themselves with the hierarchy of the GPU. Kernels are executed by threads in SIMD groups called warps arranged in thread blocks. Different threads in the same block can coordinate by fast shared memory, whereas threads in different blocks can only coordinate via the slower global memory. To further complicate the programmer's task, there are also issues of SIMD divergence and memory access patterns. All of these must be taken into account to achieve optimal performance.

The difficulties of directly programming through the low-level CUDA framework is what inspired many languages, compilers, and frameworks that try to lift the level abstraction. One of those is the Accelerate language and compiler. It allows for data parallel programming through collective array operations. We explore it in depth in Section 2.3.

2.3 Accelerate

As part of our work, we extend the array language *Accelerate*. What follows is an overview of Accelerate from the perspective of a user of the language: how to write programs, how they are executed, what its limitations are, etc. In the subsequent section (2.3.2) we will explore some of the internal design of the compiler.

2.3.1 Introduction

Accelerate is a parallel array language consisting of a carefully selected set of operations (combinators) on multidimensional arrays, which can be compiled efficiently to bulk-parallel SIMD hardware. It is *deeply embedded* in Haskell, meaning that we write Accelerate programs with (slightly stylised) Haskell syntax.

Accelerate code embedded in Haskell is not compiled to SIMD code by the Haskell compiler; instead, the Accelerate library includes a *runtime compiler* that generates parallel SIMD code at application runtime. Currently, there are two complete code-generation backends, one for multi-core CPUs and one for Nvidia GPUs. Both are able to share a significant code base owing to the fact they both work via LLVM.

The collective operations in Accelerate are based on the scan-vector model [11, 38], and consist of multi-dimensional variants of familiar Haskell list operations such as `map` and `fold`, as well as array-specific operations such as index permutations.

For example, this is a simple vector dot-product function written in Accelerate:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

The function `dotp` consumes two Accelerate computations. The type `Acc a` can be interpreted to mean an Accelerate computation producing an `a`. So, in this case, each of the argument computations produce a one-dimensional array (`Vector`) of `Float` when evaluated. This function then returns a new computation which, when evaluated, yields a single (`Scalar`) result as output.

The Accelerate code for `dotp` is almost identical to what we would write in standard Haskell over lists, and is certainly more concise than an explicitly GPU-accelerated or SIMD-vectorized[4] low-level dot-product, while still compiling to efficient code [10, 30, 31].

The functions `zipWith` and `fold` are defined by the Accelerate library and have *highly parallel* semantics, supporting up to as many parallel threads as data elements. The type of `fold` is:

```
fold :: (Shape sh, Elt e)
     ⇒ (Exp e → Exp e → Exp e)
     → Exp e
     → Acc (Array (sh:.Int) e)
     → Acc (Array sh e)
```

This definition exposes some other important components to the language. The type class `Shape` indicates that a type is admissible as an array *shape*. In the simple terms, this means that it must be one of 1D, 2D, 3D, etc. The valid members of the `Shape` class are:

```
instance Shape Z
instance Shape sh ⇒ sh :. Int
```

This allows for arbitrary rank shapes to be built at the type level as snoc-lists [10, 24]. Accelerate defines some synonyms for common shapes:

```
type DIM0 = Z
type DIM1 = DIM0:.Int
```

[4]That is, a program utilising the SSE/AVX instruction set extensions for x86 processors.

```
type DIM2 = DIM1:.Int
type DIM3 = DIM2:.Int
type DIM4 = DIM3:.Int
...
```

The shape `Z` is the scalar shape. An array of shape `Z` is called a scalar array and only has 1 element. This was the result type of `dotp`, via this synonym:

```
type Scalar = Array Z
```

Similarly, a vector is just a 1D array.

```
type Vector = Array DIM1
```

Going back to `fold`, we also have the `Elt` type class. This indicates that a type is admissible as an element of an array. It encompasses many primitive types including:

```
instance Elt Int
instance Elt Float
instance Elt Double
instance Elt Char
instance Elt Bool
instance Elt Word8
instance Elt Word16
instance Elt Word32
...
```

In addition, tuples of arity up to 16 are admissible.

```
instance (Elt a, Elt b)               ⇒ Elt (a,b)
instance (Elt a, Elt b, Elt c)        ⇒ Elt (a,b,c)
instance (Elt a, Elt b, Elt c, Elt d) ⇒ Elt (a,b,c,d)
...
```

In fact, `Elt` is user extensible via representation types. We won't cover how that works in detail here, but the salient point is that `Elt` is a truly open type class.

The type signature for `fold` also shows how Accelerate is stratified into two different levels. We've already seen collective *array computations*, represented by terms of the type `Acc t`, but there is also *scalar expressions*, represented by terms of type `Exp t`. More concretely, values of type `Acc t` and `Exp t` do not execute computations directly, rather they represent abstract syntax trees (ASTs) constituting a computation that, once executed, will yield a value of type `t`. Collective operations comprise many scalar operations which are executed in data-parallel, but scalar operations *cannot* initiate collective operations. This stratification helps to statically exclude nested parallelism. We will explore in more detail what this restriction means for modularity in Chapter 3.

Operations

Accelerate offers a number of different array operations. We'll give a brief overview of the key ones here.

`map` and `zipWith` behave in much the same way as those same operations over lists. For multidimensional arrays `zipWith` yields an array of the minimum shape of all dimensions in the array. For example, if array `a` has extent

```
Z :. 5 :. 4
```

and array `b` has extent

```
Z :. 3 :. 6
```

then the result of

```
zipWith f a b
```

Will have extent

```
Z :. 3 :. 4
```

The `fold` we used in `dotp` works by folding along the inner dimension. One can view this as performing a series of many folds. Every vector along the inner dimension is reduced.

`generate` is the most general array operation.

```
generate :: (Shape sh, Elt e) ⇒ Exp sh → (Exp sh → Exp e) → Array sh e
```

It constructs an array of the given extent using the supplied function to fill its elements. For exammple:

```
evens :: Exp Int → Acc (Vector Int)
evens sz = generate (index1 sz) (λix → unindex ix * 2)
```

This example also shows the scalar operations `index1` and `unindex1`.

```
index1 :: Exp Int → Exp DIM1
index1 :: Exp DIM1 → Exp Int
```

They convert between `Int`s and `DIM1`s.

Similarly, we have the operations `lift` and `unlift`. They are generic functions for converting tuples in the meta-language to tuples in the object language, and vice versa. Expressing these functions in the Haskell type system is tricky, but one can think of them in this form.

```
lift   :: (Exp e1, Exp e2, … , Exp en) → Exp (e1, e2, … , en)
unlift :: Exp (e1, e2, … , en)         → (Exp e1, Exp e2, … , Exp en)
```

Various different forms of prefix sum (scan) are supported. One of the main ones is, `scanl`, for performing a left scan.

```
scanl :: (Shape sh, Elt e)
      ⇒ (Exp e → Exp e → Exp e)
      → Exp e
      → Acc (Array (sh:.Int) e)
      → Acc (Array (sh:.Int) e)
```

It too is similar in semantics to `scanl` in Haskell. Like `fold`, it assumes the binary operation is associative and works over the inner dimension.

Other array and scalar operations we will introduce as the need arises.

2.3.2 Internals

The accelerate compilation pipeline has 4 major stages and uses two different syntax representations. The stages are:

- Sharing recovery – Converts the higher order abstract syntax (HOAS) into first order syntax with De Bruijn indices and also recovers sharing.
- Fusion and optimisation – Tries to minimise unnecessary array traversals and optimises for optimal code generation. This produces an AST with explicit delayed arrays.
- Code generation – Generates LLVM IR from the AST and passes it off to the LLVM compiler. This stage is different for each backend. The CPU backend generates x86 binaries (with vector instructions) and the PTX backend generates PTX code for CUDA capable GPUs.
- Execution – Schedules and executes the computations on the target hardware.

The first of these stages we won't explore in detail, as it is not a concern of this work. It is sufficient to understand it as the process that produces the first-order syntax in Listing 2.2 from the programmer-friendly higher-order syntax. See McDonell et al. [30] for details on how this stage of the pipeline works.

The fusion and optimisation stage we will discuss in Chapter 5 where we will build upon the previous work described in McDonell et al. [30].

Code generation is not discussed in detail as it is not a focus of this work. See McDonell et al. [31] for how LLVM is generated from th AST in Listing 2.2.

Execution will be covered in Chapter 5 where we build upon the previous work[10, 30, 31].

Abstract Syntax Tree

The abstract syntax tree for the core Accelerate syntax is given in 2.2 for the `Acc` strata and 2.3 for the `Exp` strata. These ASTs are all parameterised by the recursive step `acc`.

Aside: Tying the recursive knot

By having the recursive step as a parameter, rather than simply making a GADT recursive, we are able to annotate it differently in different contexts. For our AST, not only can we recover the recursion with `OpenAcc`

```
newtype OpenAcc aenv a = OpenAcc (PreOpenAcc OpenAcc aenv a)
```

we are also able to, for example, annotate it with labels for user-debugging,

```
data LabelledAcc aenv a = LabelledAcc String (PreOpenAcc OpenAcc aenv a)
```

The second parameter is a snoc-list, represented using left-nested tuples, of the environment. The `Alet` AST form allows for new variables to be introduced into the environment and `Avar` extracts their values. `Idx` is a typed De Bruijn index.

```
data Idx env a where
  ZeroIdx :: Idx (env,a) a
  SuccIdx :: Idx env a
          → Idx (env,b) a
```

We can see the relation between the `Exp` and `Acc` stratas by looking at the definition of the `Generate` constructor, and expanding the type synonyms within it

```
Generate :: (Elt e, Shape sh)
         ⇒ PreOpenExp acc aenv () sh
         → PreOpenFun acc aenv () (sh → e)
         → PreOpenAcc acc aenv (Array sh e)
```

we observe how both arguments are parameterised by the recursive `acc` step and array environment but have closed scaler environments. We say that these term are open with respect to the array environment but closed with respect to the scalar environment.

Tuples are represented with a snoc-list, like the environment.

In the case of `Index`, we see how the recursive step can be embedded in `Exp` terms. While the type system does not enforce this, for most stages of the compilation we have as an invariant that any `Acc` term embedded in `Exp` must be an `Avar`. This way, scalar computations can access arrays previously computed but cannot compute further array operations.

```
type Acc = OpenAcc ()
newtype OpenAcc aenv a = OpenAcc (PreOpenAcc OpenAcc aenv a)

data PreOpenAcc acc aenv a where
  Map      :: (Elt a, Elt b, Shape sh)
           ⇒ PreFun     acc aenv (a → b)
           → acc            aenv (Array sh a)
           → PreOpenAcc acc aenv (Array sh b)
  ZipWith  :: (Elt a, Elt b, Elt c, Shape sh)
           ⇒ PreFun     acc aenv (a → b → c)
           → acc            aenv (Array sh a)
           → acc            aenv (Array sh b)
           → PreOpenAcc acc aenv (Array sh c)

  Generate :: (Elt e, Shape sh)
           ⇒ PreExp     acc aenv sh
           → PreFun     acc aenv (sh → e)
           → PreOpenAcc acc aenv (Array sh e)

  Fold     :: (Elt a, Shape sh)
           ⇒ PreFun     acc aenv (a → a → a)
           → acc            aenv (Array (sh:.Int) a)
           → acc            aenv (Array sh a)
           → PreOpenAcc acc aenv (Array sh a)

  Avar     :: Arrays a
           ⇒ Idx aenv a
           → acc aenv a

  Alet     :: (Arrays a, Arrays b)
           ⇒ acc            aenv     a
           → acc            (aenv,a) b
           → PreOpenAcc acc aenv     b

  Atuple   :: Arrays t
           ⇒ Atuple     acc aenv (TupleRepr t)
           → PreOpenAcc acc aenv t
  ...
```

Listing 2.2: The first-order abstract syntax of the `Acc` level of Accelerate.

```
type Fun = OpenFun ()
type PreFun acc = PreOpenFun acc ()
type OpenFun = PreOpenFun OpenAcc

data PreOpenFun acc env aenv f where
  Body :: Elt b
       ⇒ PreOpenExp acc env     aenv b

  Lam  :: (Elt a, Elt b)
       ⇒ PreOpenFun acc (env,a) aenv b
       → PreOpenFun acc env     aenv (a → b)

type Exp = OpenExp ()
type PreExp acc = PreOpenExp acc ()
type OpenExp = PreOpenExp OpenAcc

data PreOpenExp acc env aenv e where
  Index   :: (Shape sh, Elt e)
          ⇒ acc                 aenv (Array sh e)
          → PreOpenExp acc env aenv sh
          → PreOpenExp acc env aenv e

  Let     :: (Elt a, Elt b)
          ⇒ PreOpenExp acc env     aenv a
          → PreOpenExp acc (env,a) aenv b
          → PreOpenExp acc env     aenv b

  Var     :: Elt e
          ⇒ Idx env e
          → PreOpenExp env aenv e

  Tuple   :: (Elt t, IsTuple t)
          ⇒ Tuple      acc env aenv (TupleRepr t)
          → PreOpenExp acc env aenv t
  ...

data Tuple acc env aenv t where
  NilTup  :: Tuple acc env aenv ()

  SnocTup :: Elt a
          ⇒ Tuple      acc env aenv t
          → PreOpenExp acc env aenv a
          → Tuple      acc env aenv (t,a)
```

Listing 2.3: The first-order abstract syntax of the `Exp` level of Accelerate.

Chapter 3

Sequences

In this chapter we will introduce *irregular array sequences* as an extension to Accelerate. This extension adds to the expressiveness of Accelerate in a significant way, but requires both a new program transformation, described in Chapter 4 and other optimisations, described in Chapter 5.

3.1 Limitations of Accelerate

While flat data parallel array languages offer an expressive, high-level means of parallel programming, they are are not without limitations. In particular:

1. The reduction of modularity that results from strictly flat data-parallelism (3.1.1).

2. The memory usage that arises from the inherent random access nature of arrays (3.1.2).

3.1.1 Modularity

Recall the simple dot product example from Section 2.3:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

We would like to re-use this function to implement a matrix-vector product, by applying it to each row of the input matrix:

```
mvm_ndp :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_ndp mat vec =
  let Z :. m :. _ = shape mat
  in  generate (Z:.m) (λrow → the $ dotp vec (slice mat (row :. All)))
```

Here, `generate` creates a one-dimensional vector containing `m` elements by applying the supplied function at each index in data-parallel. At each index, we extract the appropriate row of the matrix using `slice`, and pass this to our existing `dotp` function together with the input vector.

The `slice` operation is a generalised array indexing function which is used to cut out *entire dimensions* of an array. In `slice mat (row :. All)`, we extract `All` columns at one specific `row` of the given two-dimensional matrix, resulting in a one-dimensional vector.

`the` is a specialised from of indexing for extracting the value from a scalar array.

```
the :: Acc (Scalar a) → Exp a
the = (! Z)
```

Unfortunately, since both `generate` and `dotp` are data-parallel operations, this definition requires nested data-parallelism and is thus not permitted.[1] More specifically, the problem lies with the data-parallel operation `slice` which *depends on* the scalar argument `row`. The clue that this definition includes nested data-parallelism is in the use of `the`. This effectively conceals that `dotp` is a collective array computation in order to match the type expected by `generate`, which is that of scalar expressions. Consequently, we cannot reuse `dotp` to implement matrix-vector multiplication; instead, we have to write it from scratch using `replicate`

```
mvm_rep :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_rep mat vec =
    let Z :. m :. _ = shape mat
        vec'        = replicate (Z :. m :. All) vec
    in
    fold (+) 0 ( zipWith (*) mat vec' )
```

Here, we're using the inverse of `slice`, `replicate`. It replicates a given array in across an arbitrary set of dimensions. With `replicate (Z :. m :. All) vec`, we're treating `vec` as a row vector and replicating it across the Y dimension m times.

Similar to this example, computations on irregular structures, such as sparse matrices, which are most naturally expressed in nested form, require an unwieldy encoding when we are restricted to only flat data-parallelism (see Section 3.2.1).

3.1.2 Memory usage

In addition to the loss of modularity, array-based programming languages like Accelerate suffer from another limitation. In particular, they are hampered by resource constraints, such as limited working memory. This is particularly problematic for compute accelerators such as GPUs, which have their own, usually much smaller, high-performance memory areas separate from the host's main memory[2]. An array, by definition, supports constant time random access. While certain optimisations can be made to avoid this in some cases (e.g. fusion, see Section 5.4), arrays must generally be loaded into device memory in their entirety.

[1]By not permitted, we mean it will not compile. It will type check, but the Accelerate compiler will reject it. This happens at Accelerate compile time which is actually Haskell runtime.

[2]The current top-of-the-line NVIDIA Tesla V100 has 16GB of on-board memory; much less than the amount of host memory one can expect in a workstation-class system this product is aimed at. Most other

Where algorithmically feasible, such devices require us to split the input into *chunks*, which we stream onto, process, and stream off of the device one by one. This requires a form of nesting if we want to maintain code portability across architectures with varying resource limits, while still writing high-level, declarative code.

3.2 Irregular array sequences

The type of sequences we present are *irregular sequences* of arrays. An irregular sequence of arrays is an ordered collection of arrays, where the extent (the size of the array in each dimension [3]) of the arrays within one sequence may vary. This is in contrast to *regular sequences* which are sequences that contain arrays that are all the same extent. Clearly, irregular sequences are the more general form, so opting for them in order to widen the domain of programs that we can write is the obvious choice. There are, however, advantages in having sequences that are known to be regular in terms of scheduling and code generation. In chapter 4 we explore this in more detail, but here we will put aside the issue of performance and simply explore what can be expressed with our sequences. Later we will show how opting for the more general form of them does not degrade performance for the subset of programs that do not require irregularity.

We denote sequences as `[Array sh e]` in Accelerate. That can be read as a sequence of arrays of shape `sh` containing elements of type `e`. We also support sequences of array tuples, so `[(Array sh e, Array sh' e')]` is also possible. This syntax, borrowed from Haskell lists, we can use because we are in the context of an embedded language and cannot clash with lists in the meta language. In addition to being lightweight syntax, there is some semantic similarity to lists as we will demonstrate.

In the practical implementation that we use for the benchmarks, we restrict the level of nesting to a depth of one; that is, we support sequences of arrays, but not sequences of sequences of arrays. Previous work [27] showed that the efficient implementation of more deeply nested irregular structures requires sophisticated runtime support whose SIMD implementation (e.g., for GPUs) raises an entire set of questions in its own right. However, we would like to stress that the program transformation at the core of this work, presented in Section 4.4, is *free of this limitation* and may be used on programs with deeper nesting levels. Nevertheless, we leave further exploration of this generalisation to future work.

In Accelerate, we distinguish embedded scalar computations and embedded array computations by the type constructors `Exp` and `Acc`, respectively. Similarly, we use a new contructor, `Seq`, to mark sequence computations. And, just like an array computation of type `Acc (Array sh e)` encompasses many data-parallel scalar computations of type `Exp e`

GPUs include significantly less memory.

[3]While Accelerate uses the term *shape* to refer to both the type level dimensionality and the value level size of each dimension, to avoid ambiguity we will reserve shape for the former and use extent for the latter.

to produce an array, a sequence computation of type `Seq [Array sh e]` comprises many stream-parallel array computations of type `Acc (Array sh e)`. Specifically, we consider values of type `[Array sh e]` to be sequences (or streams) of arrays, and values of type `Seq [Array sh e]` to be *computations* that, when run, produce sequences of arrays. This is an important distinction. Evaluating a value of type `Seq t` does not trigger any sequence computation, it only yields the *representation* of a sequence computation. To actually trigger a sequence computation, we must consume the sequence into an array computation to produce an array by way of the function:

```
consume :: Arrays arr ⇒ Seq arr → Acc arr
```

To consume a sequence computation, we need to combine the stream of arrays into a single array first — note how the argument of `consume` takes a `Seq arr` and not a `Seq [arr]` and the `Arrays` constraint on `arr`.[4] Depending on the desired functionality, this can be achieved in a variety of ways. The most common combination functions are `elements`, which combines all elements of all the arrays in the sequence into one flat vector; and `tabulate`, which concatenates all arrays along the outermost dimension (trimming according to the smallest extent along each dimension, much like multi-dimensional uniform `zip`):

```
elements :: (Shape sh, Elt e) ⇒ Seq [Array sh e] → Seq (Vector e)
tabulate :: (Shape sh, Elt e) ⇒ Seq [Array sh e] → Seq (Array (sh:.Int) e)
```

Conversely, we produce a stream of array computations by a function that is not unlike a one-dimensional sequence variant of `generate`:

```
produce :: Arrays a ⇒ Exp Int → (Exp Int → Acc a) → Seq [a]
```

Its first argument determines the length of the sequence and the second is a stream producer function that is invoked once for each sequence element.

In addition to these operations for creating and collapsing sequences, we only need to be able to map over sequences with:

```
mapSeq :: (Arrays a, Arrays b) ⇒ (Acc a → Acc b) → Seq [a] → Seq [b]
```

to be able to elaborate the function `dotp` (the first example of an Accelerate program) into sequence-based matrix-vector multiplication:

```
mvm_seq :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_seq mat vec =
  let Z :. m :. _ = shape mat
      rows        = produce m (λrow → slice mat (Z :. row :. All)) :: Seq [Vector Float]
  in consume (elements (mapSeq (dotp vec) rows))
```

We stream the matrix into a sequence of its rows using `produce`, apply the previously defined dot-product function `dotp` to each of these rows, `combine` the scalar results into a vector with

[4]Recall from Chapter 2 that the type class `Arrays` constrains a type to be either an array or a tuple of arrays

`elements`, and finally `consume` that vector into a single array computation that yields the result.

We are also able to combine two sequences with `zipWithSeq`:

```
zipWithSeq ::(Arrays a, Arrays b, Arrays c) ⇒ (Acc a → Acc b → Acc c) → Seq [a] → Seq [b]
```

Naturally, this yields sequence that is equal in length to the shortest of the two inputs.

Although we are re-using Haskell's list type constructor `[]` for sequences in the embedded language, the Accelerate code generator does not actually represent them in the same way. Nevertheless, the notation is justified by our ability to incrementally stream a lazy list of arrays into a sequence of arrays for pipeline processing with:

```
streamIn :: (Shape sh, Elt e) ⇒ [Array sh e] → Seq [Array sh e]
```

3.2.1 Irregularity

The arrays in the sequence used in the implementation of $\texttt{mvm}_{\texttt{seq}}$ are all of the same size; after all, they are the individual rows of a dense matrix. Hence, the sequence is *regular*. In contrast, if we use sequences to compactly represent *sparse* matrices, the various sequence elements will be of varying size, representing an irregular sequence or stream computation.

We illustrate this with the example of the multiplication of a sparse matrix with a dense vector. We represent sparse matrices in the popular *compressed row (CSR)* format, where each row stores only the non-zero elements together with the corresponding column index of each element. For example, the following matrix is represented as follows (where indexing starts at zero):

$$\begin{pmatrix} 7 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 2 & 3 \end{pmatrix} \quad \Rightarrow \quad [\ [(0,7.0)],\ [],\ [(1,2.0),(2,3.0)]\ ]$$

Representing our sparse matrix as a sequence of the matrix rows in CSR format, we can define sparse-matrix vector product as:

```
type SparseVector a = Vector (DIM1, a)
type SparseMatrix a = [SparseVector a]

smvmseq :: Seq (SparseMatrix Float) → Acc (Vector Float) → Acc (Vector Float)
smvmseq smat vec
  = collect . elements
  $ mapSeq (λsrow → let (ix,vs) = unzip srow in dotp vs (gather ix vec)) smat
```

Here, `gather` is a form of backwards permutation.

```
gather :: Acc (Array sh sh') → Acc (Array sh' e) → Acc (Array sh e)
```

As discussed above, when the irregularity is pronounced, we need to be careful with scheduling; otherwise, performance will suffer. We will come back to this issue in Chapter 5.

Again though, we emphasise that whether a sequence is regular or irregular is a detail tracked by the compiler. The programmer is able to treat all sequences as irregular and be assured that any available regularity will be exploited on their behalf.

3.2.2 Streaming

The `streamIn` function (whose signature was given above) turns a Haskell list of arrays into a sequence or stream of those same arrays. If that stream is not consumed all at once, but rather array by array or perhaps chunkwise (processing several consecutive arrays at once), then the input list will be demanded lazily as the stream gets processed. Similarly, we have the function:

```
streamOut :: Arrays a ⇒ Seq [a] → [a]
```

which consumes the results of a sequence computation to produce an incrementally constructed Haskell list with all the results of all the array computations contained in the sequence. Overall, this allows for stream computations exploiting pipeline parallelism. In particular, the production of the stream, the processing of the stream, and the consumption of the stream can all happen concurrently and possibly on separate processing elements.

Moreover, our support for irregularity facilitates balancing of resources. For example, if the sequence computation is executed on a GPU, the underlying scheduler can dynamically pick chunks of consecutive arrays from the sequence such that it balances two competing constraints. Firstly, modern hardware, in particular GPUs, offer considerable parallelism. The number of arrays in the chunk needs to expose sufficient parallelism to fully utilise the parallelism available. Secondly, the limits imposed by the relatively small amounts of working memory (on GPUs) are not exceeded. This flexibility, together with the option to map pipeline parallelism across multiple CPU cores, provides the high-level declarative notation that we sought in the introduction.

To illustrate, we have implemented the core of an audio compression algorithm. The computation proceeds by, after some pre-processing, moving a sliding window across the audio data, performing the same computation at each window position. Finally, some post-processing is performed on the results of those windowed computations. As the computations at the various window positions are independent, they may be parallelised. However, each of the windowed computations on its own is also compute intensive and offers ample data-parallelism. Here, we will not go into detail about the pre and post-processing steps nor much about the algorithm itself. We are primarily concerned with the expressiveness of our sequences extension here, so we will leave a full description of the algorithm to Chapter 7.

This style of decomposition is common and well suited to a stream processing model. We perform the preprocessing in vanilla Haskell, stream the sequence of windowed computations through Accelerate code, and consume the results for post-processing. In our application, the stream is irregular, as the information density of the audio waveform varies at different times in the audio stream, which in turn leads to different array sizes for each windowed computation. If we choose to offload the Accelerate stream computations to a GPU, we realise a stream-processing pipeline between CPU and GPU computations.

This is the essential structure of the computation:

```
type AudioData = ⟨tuple of arrays⟩

zc_core = post_processing . zc_stream . pre_processing

zc_stream :: AudioData → [(Array DIM2 Double, Array DIM2 Double)]
zc_stream audioData = streamOut $ mapSeq (processWindow audioData) windowIndexes
  where
    windowIndexes :: Seq [Scalar Int]
    windowIndexes = produce (sizeOf audioData) id

    processWindow :: AudioData → Scalar Int → Acc (Array DIM2 Double, Array DIM2 Double)
    processWindow = ...
```

The core of the algorithm, `zc_core`, consists of the steps of pre-processing the audio data, the Accelerate stream computation, and finally post-processing. The Accelerate stream computation generates a stream of window indexes (`windowIndexes`) using `produce`, maps the windowed data processing function `processWindow` over that sequence using `mapSeq`, and incrementally produces a list of outputs, one per window, with `streamOut`. In the current setup, `pre_processing` must be complete before stream processing can start; we choose to do this because, since the data in successive steps of the sliding window strongly overlap, the approach here is more efficient than explicitly creating a stream of windowed data. Instead, the input stream `windowIndexes` is a sequence of integer values that indicate which window (subset of the input data) the associated sequence computation ought to extract from the input `audioData`. This setup is similar to the use of `generate` and `slice` in $\texttt{mvm}_{\texttt{ndp}}$ in Section 3.1.1.

In contrast to `pre_processing` and stream generation, we do use pipeline parallelism to overlap the stream processing performed by `mapSeq (processWindow audioData)` with the `post_processing` by using `streamOut`. Hence, we have got three sources of parallelism: (1) `processWindow` contains considerable data-parallelism; (2) the stream scheduler can run multiple array computations corresponding to distinct stream elements in parallel; and (3) `streamOut` provides pipeline parallelism between stream processing and `post_processing`. The second source of parallelism allows the Accelerate runtime considerable freedom in adapting to the resources of the target system, and we will see in Chapter 7 that this is helpful in providing performance portability between multicore CPUs and GPUs.

Chapter 4

Flattening

To efficiently implement the sequence computations described in Chapter 3, there are many issues that need to be addressed. This chapter focuses on the most significant one: transforming nested, possibly irregular, data-parallelism into a form that can be efficiently executed on SIMD hardware. The solution not only allows for sequence computations, but, as we will show, actually solves the more general problem of efficient execution of nested arrays in a SIMD context. What distinguishes the approach presented here from others, is that we consider both regular and irregular computation, brought about by our desire to have efficient regular sequences as they have different needs with regard to their representation in memory. Concretely, in this Chapter we will:

- Introduce the $\textsc{Acc}_{NDP}$ language. A simple calculus with arrays and parallel operations that operate over them (Section 2.1).
- Using $\textsc{Acc}_{NDP}$ as a basis, describe a novel generalisation of Blelloch's flattening transform that explicitly distinguishes between regular and irregular computations, to generate more efficient code for the former while still supporting the latter (Section 4.4).

On the basis of the contents of this chapter, Chapter 5 describes the concrete implementation in Accelerate.

4.1 Introduction

In Chapter 2 we described the nested data-parallel model of programming. Research in languages supporting this model have, primarily, focused on the problem of executing them on SIMD hardware. Here, we will build upon this research, not only to apply it to our streaming array language domain, but we will also describe a more general extension that recognises inherent differences between regular and irregular array programs.

Blelloch and Sabot [6]'s original version of the flattening transformation was for a first-order language with built-in second-order combinators, NESL, which makes it a good fit for Accelerate which has similar properties. Nevertheless, the original transformation has two severe shortcomings, which makes the generated code non-competitive: (1) it lifts code into vector space that ought to remain as is for best performance; and (2) it doesn't treat the important special case of operations on regular multi-dimensional arrays specially. We address point (1) by adapting the work on *vectorisation avoidance* [25], and tackle point (2) with a generalisation of flattening that identifies regular (sub)computations and optimises for them. Moreover, this transformation is, in contrast to previous work, type-preserving. This is necessary for integration with Accelerate, which is built around a typed AST and is type preserving in all its compilation stages[31]. It also has the added benefit of statically guaranteeing that the transform we describe here produces type-correct programs.

As a basis on which to describe the transform, we introduce a simple language which we call Acc_{NDP}. For the sake of simplicity, the language has fewer parallel operations than Accelerate. However, it is also more general in that it supports arbitrarily nested parallel arrays, as the transformation truly is an extension of similar flattening approaches. The following section will describe Acc_{NDP} in detail. The reader will recognise its similarity to Accelerate, but unlike Accelerate we will formally specify its syntax and type system.

4.2 Acc_{NDP}

Before introducing the precise syntax, let us have a look at the high-level properties to give a general overview of the language:

- **Parallel arrays**: All parallelism is specified with multi-dimensional arrays and data parallel combinators that operate over them. The arrays are immutable, thus avoiding the possibility of race conditions and impurity.
- **Non-array values**: Not every value in Acc_{NDP} is an array. This is in contrast to the APL family of languages[23] but corresponds with Accelerate.
- **Strictness**: As in Accelerate, let bindings and array elements are strict. Despite being embedded in the non-strict meta language of Haskell, Accelerate is itself wholly strict. We are not concerned with lazy arrays.

The grammar of the language is listed in Figure 4.1 and the typing rules in 4.3. What follows is a description of each of the constructs of the language in detail. We also define some of the conventions we will use throughout this chapter.

4.2.1 Literals

The supported literals are integer literals and the `Z` shape descriptor. The latter is explained in more detail below. It would be trivial to support additional forms of numeric literal

$$
\begin{array}{llcl}
\text{literals} & l & ::= & \mathbb{Z} \mid \mathrm{Z} \\
\text{variables} & v & ::= & v_0 \mid v_1 \mid \ldots \\
\text{tuples} & t & ::= & (e_0, \ldots, e_n) \\
\text{unary-ops} & p_1 & ::= & \text{indexHead} \mid \text{indexTail} \mid \text{indexLeft} \mid \text{indexRight} \\
\text{binops} & p_2 & ::= & (+) \mid (*) \mid (-) \mid \text{intersect} \mid (:.) \mid (+\!\!+) \\
\text{expressions} & e & ::= & l \mid v \mid t \mid e_1 \mathbin{!} e_2 \mid p_1\ e \mid p_2\ e_1\ e_2 \\
 & & \mid & \text{let } v_0 = e_1 \text{ in } e_2 \\
 & & \mid & \text{prj } \mathbb{N}^+\ e \\
 & & \mid & \text{extent } e \\
 & & \mid & \text{generate } e_1\ (\lambda v_0.\, e_2) \\
 & & \mid & \text{fold } (\lambda v_1\ v_0.\, e_1)\ e_2\ e_3 \\
 & & \mid & s \\
\text{segments} & s & ::= & \text{generate}_{seg}\ e_1\ (\lambda v_0.\, e_2) \\
 & & \mid & \text{fold}_{seg}\ (\lambda v_1\ v_0.\, e_1)\ e_2\ e_3\ e_4 \\
 & & \mid & \text{segmented } e \\
 & & \mid & \text{cross } e_1\ e_2 \\
 & & \mid & e_1 \mathbin{\#} e_2 \\
 & & \mid & \text{intersects } e_1\ e_2 \\
 & & \mid & \text{lefts } e \\
 & & \mid & \text{rights } e
\end{array}
$$

Figure 4.1: The grammar of ACC$_{NDP}$

$$
\frac{}{\text{IsShape Z}} \qquad \frac{\text{IsShape } sh}{\text{IsShape } (sh :. \text{Int})}
$$

$$
\frac{}{\text{IsScalar Int}} \qquad \frac{\text{IsShape } sh}{\text{IsScalar } sh} \qquad \frac{\text{IsScalar } \alpha_1 \quad \text{IsScalar } \alpha_2 \quad \ldots \quad \text{IsScalar } \alpha_n}{\text{IsScalar } (\alpha_1, \alpha_2, \ldots, \alpha_n)}
$$

Figure 4.2: Type level predicates

$$\frac{}{\Gamma \vdash \mathrm{Z} :: \mathrm{Z}} \qquad \frac{l \in \mathbb{Z}}{\Gamma \vdash\ l :: \mathrm{Int}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ ix :: sh \\ \Gamma \vdash\ i :: \mathrm{Int}\end{array}}{\Gamma \vdash\ ix{:}.i :: sh{:}.\mathrm{Int}}$$

$$\frac{\Gamma \vdash\ e_1 :: \alpha_1 \qquad \Gamma \vdash\ e_2 :: \alpha_2 \qquad \dots \qquad \Gamma \vdash\ e_n :: \alpha_n}{\Gamma \vdash\ (e_1, e_2, \dots, e_n) :: (\alpha_1, \alpha_2, \dots, \alpha_n)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e :: sh{:}.\mathrm{Int} \\ \mathrm{IsShape}\ sh\end{array}}{\Gamma \vdash \mathrm{indexHead}\, e :: \mathrm{Int}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e :: sh{:}.\mathrm{Int} \\ \mathrm{IsShape}\ sh\end{array}}{\Gamma \vdash \mathrm{indexTail}\, e :: sh} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_0 :: sh \\ \Gamma \vdash\ e_1 :: sh \\ \mathrm{IsShape}\ sh\end{array}}{\Gamma \vdash \mathrm{intersect}\, e_0\ e_1 :: sh}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e :: sh \mathbin{+\!\!+} sh' \\ \mathrm{IsShape}\ sh \\ \mathrm{IsShape}\ sh'\end{array}}{\Gamma \vdash \mathrm{indexLeft}\, e :: sh} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e :: sh \mathbin{+\!\!+} sh' \\ \mathrm{IsShape}\ sh \\ \mathrm{IsShape}\ sh'\end{array}}{\Gamma \vdash \mathrm{indexRight}\, e :: sh'} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_0 :: sh \\ \Gamma \vdash\ e_1 :: sh' \\ \mathrm{IsShape}\ sh \\ \mathrm{IsShape}\ sh'\end{array}}{\Gamma \vdash\ e_0 \mathbin{+\!\!+} e_1 :: sh \mathbin{+\!\!+} sh'}$$

$$\frac{\Gamma \vdash\ e :: sh \mathbin{+\!\!+} Z}{\Gamma \vdash\ e :: sh} \qquad \frac{\Gamma \vdash\ e :: sh \mathbin{+\!\!+} (sh'{:}.\mathrm{Int})}{\Gamma \vdash\ e :: (sh \mathbin{+\!\!+} sh'){:}.\mathrm{Int}} \qquad \frac{\Gamma \vdash\ e :: (sh \mathbin{+\!\!+} sh') \mathbin{+\!\!+} sh''}{\Gamma \vdash\ e :: sh \mathbin{+\!\!+} (sh' \mathbin{+\!\!+} sh'')}$$

$$\frac{\begin{array}{c}p \in \{(+), (*), (-)\} \\ \Gamma \vdash\ e_0 :: \mathrm{Int} \\ \Gamma \vdash\ e_1 :: \mathrm{Int}\end{array}}{\Gamma \vdash\ p\ e_0\ e_1 :: \mathrm{Int}} \qquad \frac{}{\alpha,\ \Gamma \vdash\ v_0 :: \alpha} \qquad \frac{\Gamma \vdash v_n :: \alpha}{\beta,\ \Gamma \vdash\ v_{n+1} :: \alpha}$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathrm{Array}\ sh\ \alpha \\ \Gamma \vdash ix :: sh\end{array}}{\Gamma \vdash e\ !\ ix :: \alpha} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 :: \alpha \\ \alpha,\ \Gamma \vdash e_2 :: \beta\end{array}}{\Gamma \vdash \mathrm{let}\ v_0 = e_1\ \mathrm{in}\ e_2 :: \beta}$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: (\alpha_1, \alpha_2, \dots, \alpha_n) \\ 1 \leq i \leq n\end{array}}{\Gamma \vdash prj\ i\ e :: \alpha_i} \qquad \frac{\Gamma \vdash e :: \mathrm{Array}\ sh\ \alpha}{\Gamma \vdash \mathrm{extent}\ e :: sh} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 :: sh \\ sh,\ \Gamma \vdash e_2 :: \alpha \\ \mathrm{IsShape}\ sh\end{array}}{\Gamma \vdash \mathrm{generate}\ e_1\ (\lambda v_0.\, e_2) :: \mathrm{Array}\ sh\ \alpha}$$

$$\frac{\alpha,\ \alpha,\ \Gamma \vdash e_1 :: \alpha \qquad \Gamma \vdash e_2 :: \alpha \qquad \Gamma \vdash e_3 :: \mathrm{Array}\ (sh{:}.\mathrm{Int})\ \alpha}{\Gamma \vdash \mathrm{fold}\ (\lambda v_1\ v_0.\, e_1)\ e_2\ e_3 :: \mathrm{Array}\ sh\ \alpha}$$

Figure 4.3: The typing rules for $\mathrm{A}\textsc{cc}_{NDP}$

$$\frac{\Gamma \vdash e_1 :: \text{Segments } sh \qquad sh,\ \Gamma \vdash e_2 :: \alpha \qquad \text{IsShape } sh}{\Gamma \vdash \text{generate}_{seg}\ e_1\ (\lambda v_0 .\, e_2) :: \text{Vector } \alpha}$$

$$\frac{\alpha,\ \alpha,\ \Gamma \vdash e_1 :: \alpha \qquad \Gamma \vdash e_2 :: \alpha \qquad \Gamma \vdash e_3 :: \text{Segments } (sh\text{:.Int}) \qquad \Gamma \vdash e_4 :: \text{Vector } \alpha}{\Gamma \vdash \text{fold}_{seg}\ (\lambda v_1\ v_0 .\, e_1)\ e_2\ e_3\ e_4 :: (\text{Segments } sh,\ \text{Vector } \alpha)}$$

$$\frac{\text{IsShape } sh' \qquad \Gamma \vdash\ e :: \text{Array } sh\ sh'}{\Gamma \vdash \text{segmented } e :: \text{Segments } (sh +\!\!\!+ sh')} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 :: \text{Segments } sh \\ \Gamma \vdash\ e_2 :: \text{Segments } sh'\end{array}}{\Gamma \vdash \text{cross } e_1\ e_2 :: \text{Segments } (sh +\!\!\!+ sh')}$$

$$\frac{\Gamma \vdash\ e_1 :: \text{Segments } sh \qquad \Gamma \vdash\ e_2 :: sh}{\Gamma \vdash\ e_1 \# e_2 :: \text{Int}} \qquad \frac{\Gamma \vdash\ e :: \text{Segments } (sh +\!\!\!+ sh')}{\Gamma \vdash \text{expand } e :: (\text{Segments } sh,\ \text{Vector } sh')}$$

$$\frac{\Gamma \vdash\ e_1 :: \text{Segments } sh \qquad \Gamma \vdash\ e_2 :: \text{Segments } sh}{\Gamma \vdash \text{intersects } e_1\ e_2 :: \text{Segments } sh}$$

$$\frac{\Gamma \vdash\ e :: \text{Segments } (sh +\!\!\!+ sh')}{\Gamma \vdash \text{lefts } e :: \text{Segments } sh} \qquad \frac{\Gamma \vdash\ e :: \text{Segments } (sh +\!\!\!+ sh')}{\Gamma \vdash \text{rights } e :: \text{Segments } sh'}$$

Figure 4.4: The typing rules for segmented operations in ACC$_{NDP}$

(e.g. floating point literals), but only integer literals are necessary to define the flattening transformation and to give illustrative examples. While not practical for implementation, we also treat all integers as unbounded. Once again, this simplifies explanation without sacrificing correspondence to the implementation.

4.2.2 Variables and let bindings

We use DeBruijn indices [17] to represent variables. This is a common technique when representing ASTs to avoid issues of alpha-equivalence and fresh name generation. Briefly, they work by using the natural numbers in place of variables. The number indicates how many levels of binding up the super-term the variable is bound. For example

```
let v_0 = A in
  let v_0 = B in
    (v_0 + v_1) / 2
```

Here, we're calculating average of `A` and `B`. Note that while `v`$_0$ is bound at the outermost let binding, inside the body of the inner binding it is referred to via `v`$_1$.

Note that there is no general lambda term in the definition of ACC$_{NDP}$ and hence no ability to abstract over functions. While there exists the second order operations `generate` and `fold` that appear to take functions as arguments, this is just syntax. The only places lambdas occur is in the arguments to these operations. This first order nature may appear a burdensome restriction for the language, but as is indicated by NESL[5], even first order array languages can express useful examples. Also by embedding a first order language in a higher order meta language we can recover a lot of the expressiveness. Accelerate itself is a good example of this.

4.2.3 Tuples

Tuples of arbitrary arity are supported in $\text{A}\textsc{cc}_{NDP}$. They are constructed with the familiar notation of comma separated values wrapped in parens. To be precise about our terminology, we will use the Accelerate convention of referring to tuples as scalar if they contain other scalar values. For example,

```
(Int,Int)
```

is still scalar despite it being a product of primitive `Int`s. Similarly,

```
Array (Z:.Int) (Int,Int)
```

is an array of scalar elements. We consider a type scalar if it is a tuple of scalars or is a primitive (in this case just `Int` or `Z`). Formally, this is the type level predicate `IsScalar` as defined in Figure 4.2.

Tuple projection works by supplying a numeric literal representing which element of the tuple to project out. For example:

```
prj 2 (3,4,5)
```

yields 4, the second element. Thus, we have

```
fst :: (a,b) → a
fst = prj 1
```

and

```
snd :: (a,b) → b
snd = prj 2
```

4.2.4 Scalar Operations

The scalar operations of the language are divided into the binary and unary operations. We treat all operations as prefix when traversing the AST, but when writing programs we will often use infix notation for clarity. The semantics of the arithmetic operations are much as you would expect. We don't specify behaviour in edge cases, like division by zero, leaving that to be a property of the implementation. By assuming all integers are unbounded we can also ignore issues of overflow. All non arithmetic operations are over array extents. We describe those below.

4.2.5 Shapes and extents

We treat array extents (the value level representations of shapes) as heterogeneous snoc-lists. The primary way in which they are constructed is via `Z` and `(:.)`. Like Accelerate, we use `Z` to represent the zero dimension. An array of extent `Z` has a single element. We refer to such arrays as unit arrays. The snoc-operator `(:.)` extends the dimensionality of an extent by 1.

We capture shapes with the type level predicate `IsShape`. In Accelerate we achieve the same via a type class, and we can view the shape predicate in the same way, just lacking a dictionary and extensibility. As such, we will often write functions in this form.

```
foo :: IsShape sh ⇒ … sh …
```

Essentially, we steal the syntax for typeclasses from Haskell for ease of explanation.

The innermost dimension of an array is at the head of its shape descriptor, and therefore on the right end, of the snoc-list. For example the extent

```
Z :. 3 :. 4 :. 2
```

has an inner dimension of size 2 and an outer dimension of size 3. While for the most part the ordering of the array is orthogonal to what is described in the chapter, we will be consistent with Accelerate and opt for a row-major order. Therefore, an expression like this

```
generate (Z :. 3 :. 2) (λv_0. v_0)
```

results in an array laid out like so[1]

```
{ { Z:.0:.0, Z:.0:.1 },
  { Z:.1:.0, Z:.1:.1 },
  { Z:.2:.0, Z:.2:.1 } }
```

We support the following shape operations to construct and extract dimensions from shapes.

- `indexHead  :: IsShape sh ⇒(sh:.Int) →Int` gives the innermost dimension of the shape.
- `indexTail  :: IsShape sh ⇒(sh:.Int) →sh` strips the innermost dimension from a shape descriptor.
- `(:.)       :: IsShape sh ⇒sh →Int → sh:.Int` extends the shape by adding a new inner dimension.
- `(++)       :: (IsShape sh, IsShape sh') ⇒sh →sh' →sh++sh'` concatenates two shape descriptors.

[1]While ACC$_{NDP}$ does not explicitly support array literals we use them in this chapter to demonstrate what the actual values within arrays are. As seen here, we use curly braces to denote them.

- `indexLeft  :: (IsShape sh, IsShape sh') ⇒sh⧺sh' →sh` gives the left shape of a concatenation of shapes. This depends on explicit type information to decide how to split its argument. It is not necessary to write programs but is produced by flattening.
- `indexRight :: (IsShape sh, IsShape sh') ⇒sh⧺sh' →sh'` gives the right shape of a concatenation of shapes. This depends on explicit type information to decide how to split its argument. It is not necessary to write programs but is produced by flattening.

These operations on shapes are necessary to fully implement our flattening transform. They allow us to combine them arbitrarily and extract their components. The first three of these, are simple polymorphic functions over shapes. The remaining three, however, introduce some extra complexity in the form of the (⧺) type level shape concatenation operator. Unlike `(:.)`, we don't treat `(⧺)` as a type constructor, but rather as a type function that allows us to combine shape descriptors. For example,

```
Z:.Int:.Int ⧺ Z:.Int ~ Z:.Int:.Int:.Int:.Int
```

Rather than trying to add support for functions at the type level (and all the complexity that involves) to our language, we simply add typing rules specifically for this one function. Naturally, in the case of `indexLeft` and `indexRight`, we encounter the problem that they are ambiguous without explicit type information, making type inference undecidable in their presence. Fortunately, in our case, these two operations do not occur in source programs, only the output of our flattening transformation described below. As such, the ambiguity is resolved by our AST being typed.

It is also important for the purposes of flattening that we recognise (⧺) is associative. We go so far as to encode this fact into the type system by having rules that allow us to treat terms of type

```
(sh⧺sh')⧺sh''
```

also having type

```
sh⧺(sh'⧺sh'')
```

In a more advanced type systems (like Haskell's) it is possible to capture such properties with value-level witnesses and so it is not necessary for such things to be encoded in the type system. For our purposes however, it is simpler just to include the extra rules.

`(⧺)` is also commutative, but this is not a fact that is relied upon at all, so we don't encode it.

In addition to the above operations we also support

```
intersect :: Shape sh ⇒ sh → sh → sh
```

This is a useful operation for a combining shapes and implementing operations like `zipWith`. We get the largest shape that *fits* inside both given shapes with `intersect`. For example:

```
intersect (Z:.1:.2) (Z:.2:.1) = Z:.1:.1
```

4.2.6 Array Indexing

Arrays can be indexed by the bang (`!`) operator. As per the typing rule, shape of the array and the type of the index must match. For example, if `a` is a matrix,

```
a ! Z:.2:.1
```

will yield the value in the 2nd row and 1st column.

We do not specify what happens in the event an array is accessed outside its extent. In this regard, indexing is a partial operation. Moreover, we make no attempt at preserving this undefined behaviour as part of our flattening transformation – i.e. a program that accesses an array out of bounds may, after flattening, no longer do so. We assume all programs given to the transform are safe in this regard.

4.2.7 Array extents

The function `extent :: Array sh e →sh` returns the extent of an array, the value level representation of the shape. This is equivalent to `shape` in Accelerate, renamed in order to avoid ambiguity between type level and value level information.

4.2.8 Array generation

The built-in operation `generate` can be used to create arrays from a generation function.

```
generate :: IsShape sh ⇒ sh → (sh → e) → Array sh e
```

An expression of the form `generate ix (λv`$_0$`. e)` constructs an array of extent `ix` with every element computed from `(λv`$_0$`. e)`. For example,

```
evens :: Vector Int
evens = generate (Z:.10) (λix → indexHead ix * 2)
```

Note: Here we're assuming we can write terms using named variables instead of De-Bruijn indices and that we have a notion of higher-order functions. For the purposes of explanation, we will assume we can write terms in a higher-order Haskell-like meta-language that we can use to construct ACC$_{NDP}$ terms. By doing this, we can write additional higher order array combinators that will be useful not only to write more interesting programs, but also, as we will show, implementing many auxiliary functions for flattening.

Combining `generate` with indexing gives us the more traditional array combinator:

```
map :: (a → b) → Array sh a → Array sh b
map f arr = generate (extent arr) (λix → f (arr ! ix))
```

Similarly, we can also combine arrays by defining

```
zipWith :: (a → b → c) → Array sh a → Array sh b
zipWith f arr1 arr2 = generate (intersect (extent arr1) (extent arr2))
                               (λix → f (arr1 ! ix) (arr2 ! ix))
```

4.2.9 Array reduction

To perform reductions, we have `fold`, which is parameterised by a scalar function and a scalar starting value. Like the `fold` in Accelerate, it is shape polymorphic [24] and assumes associativity of the scalar function.

```
fold :: (a → a → a) → a → Array (sh:.Int) a → Array sh a
```

Also, like in Accelerate, this is a multi-dimensional fold. It works over arrays of arbitrary rank by reducing only along the inner dimension.

4.3 Segmented Arrays

In addition to the core constructs of Acc_{NDP}, we add constructs for operating with *segment descriptors*. The rules for these are given in 4.4. We treat `Segments` as an abstract data type, giving no definition but requiring the given operations to work with it. To understand what it represents, we need to be more specific about what exactly the shape of an array is. Up until this point, we have said that the shape of an array is its dimensionality. However, another interpretation is that it is a mapping from a higher dimensional index into a position in a flat vector.

When we have, an array `a` such that

```
a :: Array sh e
```

then `a` can be indexed with values of type `sh`. We know that in memory, the elements of `a` are stored in a contiguous block, but we want to abstract away from that. From an index of type `sh` into the array, we can determine the corresponding (linear) index into the contiguous block by using the extent of the array. For example, if `a` has the shape

```
Z:.5:.4:.3
```

and an index of

```
Z:.2:.1:.0
```

we know that the value of the array at this index is at position

```
0 + 3(1 + 4(2 + 5(0))) = 27
```

in the contiguous block. We're calculating the linear index via the formula.

```
linearIndex Z Z = 0
linearIndex (sh:.n) (ix:.i) = i + n*(linearIndex sh ix)
```

where the first argument is the shape of the array and the second is the index into it. Hence, we can view a shape descriptor as specifying a set of possible indices and a mapping from them to linear indices.

Given some shape, `sh`, we can enumerate all its indices with

```
indices :: IsShape sh ⇒ sh → Array sh sh
indices sh = generate sh (λv_0. v_0)
```

For example, suppose

```
sh = Z:.3:.4
```

then its indices are

```
indices (Z:.3:.4) = { { Z:.0:.0, Z:.0:.1, Z:.0:.2, Z:.0:.3 }
                    , { Z:.1:.0, Z:.1:.1, Z:.1:.2, Z:.1:.3 }
                    , { Z:.2:.0, Z:.2:.1, Z:.2:.2, Z:.2:.3 } }
```

If we are viewing shapes in this way, then we can form an intuition about segment descriptors by viewing them in a similar way. They also represent a set of indices and a mapping to linear indices, but are less restricted. A shape descriptor specifies only a single size for each dimension, so can only represent regular sets of indices. However, with segment descriptors, we represent sets of indices that are more general. For example, suppose we want to represent an irregular 2D array with 4 rows and that the first row has 4 elements, the second has 3, the third has 0 and the fourth has 1. We would need a value of type

```
unevenRows :: Segments (Z:.Int:.Int)
```

Before approaching how to construct an expression of the right type, lets first address how it is used.

We can enumerate its indices with

```
indices_seg :: Segments sh → Vector sh
indices_seg segs = generate_seg segs (λv_0. v_0)
```

so therefore its indices are

```
indices_seg unevenRows = { Z:.0:.0, Z:.0:.1, Z:.0:.2, Z:.0:.3
                         , Z:.1:.0, Z:.1:.1, Z:.1:.2
                         , Z:.3:.0 }
```

Here, we're using `generate`$_{seg}$. It is the segmented equivalent of `generate`.

```
generate_seg :: IsShape sh ⇒ Segments sh → (sh → e) → Vector e
```

It constructs a flat vector given some segments descriptors and a generation function. The `Vector` type is, like in Accelerate, just a synonym for a 1-dimensional array.

```
type Vector = Array (Z:.Int)
```

That still leaves use with the problem of how to construct segment descriptors. Unlike with shape descriptors, we can't just represent them with a list of integer dimensions. Instead, we use `segmented`. It takes an array of some shape `sh` containing values of `sh'` and gives back `Segments (sh⧺sh')`. For our example

```
unevenRows = segmented { Z:.4, Z:.3, Z:.0, Z:.1}
```

We say that this set of indices is irregular.

We can also construct segment descriptors with `cross`.

```
cross :: Segments sh → Segments sh' → Segments (sh⧺sh')
```

It is equivalent to (⧺), but operating over segments. That is to say, if

$\texttt{ix} \in \texttt{indices}_{seg}\ \texttt{segs}$

and

$\texttt{ix'} \in \texttt{indices}_{seg}\ \texttt{segs'}$

then

$\texttt{ix} \mathbin{+\!\!\!+} \texttt{ix'} \in \texttt{indices}_{seg}\ \texttt{(cross segs segs')}$

There are a number of other operations for working with segment descriptors. We will introduce them where required in the next section.

4.4 Regularity Identifying Flattening

With ACC_{NDP} defined, we can now construct a flattening transform. This transform will use ACC_{NDP} as its source language (minus the segmented operations) and also as the target language (including the segmented operations).

We will describe flattening first by example before specifying it formally. Consider the simple function of summing up elements of a vector.

```
sum :: Vector Int → Int
sum xs = the (fold (+) 0 xs)
```

Here, we use the auxiliary function `the`, borrowed from Accelerate, that indexes a zero dimension array, extracting the single element it contains.

```
the :: Array Z e → e
the arr = arr ! Z
```

Using nesting, we can use this to write a function that sums up each vector contained in a larger vector.

```
sums :: Vector (Vector Int) → Vector Int
sums xss = map sum xss
```

If the ultimate goal is to remove any nesting from a program, what we require is a version of this function that does not depend on nesting. How can we do that? We'll look at two possible situations. Suppose `xss` is regular, that is, all sub-vectors are of the same length, then we can treat our vector of vectors as a 2D array. This gives an alternative version of `sums`.

```
sums_R :: Array DIM2 Int → Vector Int
sums_R xss = fold (+) 0 xss
```

What we have is essentially the same as `sum`, but with the fold now occurring over the inner dimension of the 2D array and no longer needing to call `the` to remove the nesting.

If, however, we do not know that `xss` is regular, we would need to write a more general version of `sums`.

```
sums_Ir :: (Segments DIM2, Vector Int) → Vector Int
sums_Ir (segs, vs) = snd (fold_seg (+) 0 segs vs)
```

Here, we're representing this collection of vectors in a *segmented* form. Like the segmented arrays described in Section 2.1, it is split into a *flat data vector* (containing all the elements of all vectors) together with a set of segment descriptors describing how to index the flat vector.

Unlike our regular representation, we can't just use the same version of the fold operation to work over a segmented representation. Instead we use the segmented variant of `fold`. This has the type:

```
fold_seg :: (a → a → a)
         → a
         → Segments (sh :. Int)
         → Vector a
         → (Segments sh, Vector a)
```

We will cover in more detail the implementation of this operation in chapter 5. Suffice to say however, any implementation of it is going to be significantly more expensive than that of `fold`. Moreover, whatever the concrete representation of `Segments` is, maintaining and passing them around is an additional overhead. Clearly, we want to incur these costs only when necessary. The use of the segmented fold should be avoided wherever possible. Putting it another way, we want to make sure that when it is possible to do so, the *regular* fold is used instead.

This is not just true for this particular example or just for `fold`. In general, computations on regular, multi-dimensional arrays are much more efficient than performing the same computations on nested, potentially irregular structures. In other words, the mere ability to handle irregular structures presents a significant runtime overhead, even if it is never used. Hence, it is crucial for our variant of flattening to preserve regularity. Computations which are regular, even when written using nesting, should remain regular when flattened.

So, to recap, what we require is a method by which to take $\textsc{Acc}_{NDP}$ terms and convert them to terms that perform the same function over the most appropriate representations of its arrays. We do this by *lifting* all open terms into terms that operate in a lifted context. We will described this in the next section. However, before that, it is important to note that the regular and irregular representations are not the only ones we care about. The *avoided* representation is when we don't change the representation of arrays or scalar values at all.

As a simple example, consider this scalar function:

```
average :: Int → Int → Int
average x y = (x + y) / 2
```

Flattening of this code takes each subcomputation from a scalar function to one operating over higher dimensional arrays, the actual shape depending on the context.

```
average_R :: Array sh Int → Array sh Int → Array sh Int
average_R xs ys = zipWith (/) (zipWith (+) xs ys) (replicate (extent xs) 2)
```

This code features excessive array traversals and many superfluous intermediate structures. In this simple example, fusion optimisations can improve the code, but more generally we will arrive at better code when flattening directly *avoids* flattening purely scalar or non-nested subexpression, and generates the following code instead:

```
average_avoid :: Array sh Int → Array sh Int → Array sh Int
average_avoid xs ys = zipWith (λx y → (x + y) / 2) xs ys
```

The concrete flattening transformation presented in the rest of this section avoids the flattening of subexpressions not used in a nested context or containing nesting and preserves regularity, where flattening is over a regular domain.

4.4.1 Lifted type relation

While type information is not necessary to perform a flattening transformation, if we wish to describe it in such a way that it is type *preserving* we have to be mindful of what is happening at the type level. Here, we discuss the type transformations $\mathcal{N}$ and $\mathcal{F}$. In the next section we will show how the term transformation $\mathcal{L}$ matches up to $\mathcal{N}$ and $\mathcal{F}$.

Previous work on flattening [9] describes the type level component as a simple function. However, in this work, some types have more than one possible flattened representation. Having both regular and irregular flattening contexts give us a choice between different representations, the exact type of a term after flattening will not be known till after that term has been traversed. For this reason, we will represent our type level transformation relationally.

To aid in explanation, we will use something similar to Haskell's generalised algebraic data types (GADTs) [35] both to describe the type level relations and also to track information in the term level transformation. This advantage of this it allows us to bring type equivalencies into the environment by simply matching on constructors. It also has the added benefit of more closely matching the concrete implementation in Accelerate.

Normalisation

The first type transformation, $\mathcal{N}$ `t` $\texttt{t}_{norm}$, computes an array normal form $\texttt{t}_{norm}$ of an Accelerate type `t`. The input is any (possibly nested) type and the output is a tuple (can be a nested tuple) of arrays with no array nesting.

```
data 𝒩 t t' where
   Scalar :: IsScalar e
          ⇒ 𝒩 e (Array Z e)
   Nest   :: 𝒩 e (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)
          → 𝒩 (Array sh e) (Array (sh ⧺ sh_1) e_1, Array (sh ⧺ sh_2) e_2, … , Array (sh ⧺ sh_n) e_n)
   Tuple_𝒩 :: (𝒩 t_1 t_1', 𝒩 t_2 t_2', … , 𝒩 t_n t_n')
          → 𝒩 (t_1, t_2, … , t_n) (t_1', t_2', … , t_n')
```

Before we begin describing this fundamentally simple relation, it is first necessary to discuss an unusual feature of the meta language we use for describing the transform. For the most part, one can view the meta-language as being Haskell. This matches the implementation in Accelerate and leads to understandable definition, for a reader already familiar with Haskell. However, to further aid explanation, we allow for the meta language to generalise over tuples. We use the (...) notation to capture the fact that something can take an arbitrary n-tuple. Suppose we have a function like

```
foo :: (Array sh_1 e_1, Array sh_2 e_2, …  Array sh_n e_n) → r
```

it is able to take any tuple value as argument; e.g. both `(Array sh1 e1, Array sh2 e2)` and `(Array sh1 e1, Array sh2 e2, Array sh3 e3)` would be considered valid argument types. In addition, we treat one-tuples as being equivalent to a single value. So `Array sh e` would be a valid argument as well. While it is possible to write tuple-arity polymorphic functions in Haskell, with representation types or generic programming libraries, doing so require significant extra type level programming that has a tendency to permeate throughout the code, even in the parts where it is not relevant. We opt not to do this in this work as it what we describe already has a complex type level component and it would only obscure the relative simplicity of what is happening. By assuming our meta language lets us generalise over tuples and single values we sidestep the issue, leaving it to be explored in Chapter 5.

Going back to the relation, we will first look at the `Scalar` constructor. It specifies that scalars of type `e` are wrapped in a zero dimension array of type `Array Z e`, keeping in mind that our definition of scalar values includes tuples of scalars.

The `Nest` constructor captures how a regular n-dimensional array of regular m-dimensional arrays becomes an $n+m$-dimensional array. Or, to put it in terms of shapes, an array of shape `sh` containing arrays of shape `sh'` becomes an array of shape `sh++sh'`. For example, vectors of vectors of `Int`s, after expanding synonyms, has the type:

```
Array (Z :. Int) (Array (Z :. Int) Int)
```

We normalise that type to two-dimensional arrays of type `Array DIM2 Int`, as witnessed by:

```
Nest (Nest Scalar) ::
  𝒩 (Array (Z :. Int) (Array (Z :. Int) Int)) (Array (Z :. Int :. Int)  Int)
```

The complexity comes when dealing with tuples. For example:

```
Array (Z :. Int) (Array (Z :. Int) Int, Array (Z :. Int) Int)
```

This is a vector of pairs of vectors witnessed by:

```
Nest (Tuple𝒩 (Nest Scalar), (Nest Scalar)) ::
  𝒩 (Array (Z :. Int) (Array (Z :. Int) Int, Array (Z :. Int) Int))
     (Array (Z :. Int :. Int) Int, Array (Z :. Int :. Int) Int)
```

Because there is an array of non-scalar pairs, this has to be expanded out into a pair of arrays.

This normalisation process gives us a way to remove the nesting from data structures without specifying precisely what representation is necessary. Alone it is not sufficient in the case of irregular nesting. It is also actually a function even though we choose to handle it representationally. The need for that becomes apparent with the subsequent type level transformation.

Flattening

The second type relation, $\mathcal{F}$ `t` $\texttt{t}^{flat}$, takes an Accelerate type to its flattened form $\texttt{t}^{flat}$ by first normalising `t` by way of $\mathcal{N}$ `t` $\texttt{t}^{norm}$, and then extending the dimensionality of the normalised $\texttt{t}^{norm}$ in either a regular or irregular form.

The first argument to $\mathcal{F}$ is what we refer to as the lifting shape. It represents to what dimension the type has been lifted to. To put it another way. A term of type

$$\mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{t}\ \texttt{t}^{flat}$$

witnesses that $\texttt{t}^{flat}$ can be also be represented as an array of shape $\texttt{sh}_{\mathcal{L}}$ containing values of type `t`. The difference between the two representations being that $\texttt{t}^{flat}$ does not contain any explicit non-scalar nesting. It is important to note that the two types are not actually isomorphic. While it is possible to convert from values of $\texttt{t}^{flat}$ to `Array` $\texttt{sh}_{\mathcal{L}}$ `t`, the reverse direction is not always possible. That is because $\mathcal{F}$ also captures information about regularity and avoidance.

This desire to capture the regularity makes this transform ambiguous. This is why it is necessary to treat the whole transformation as relation. If the enclosing data-parallel context is regular, we simply increase the dimensionality of all arrays in $\texttt{t}^{norm}$ by $\texttt{sh}_{\mathcal{L}}$. However, if the context is irregular, we need to introduce segment descriptors, as discussed in the previous section. These two cases are covered by the alternatives `Regular` and `Irregular` below:

```
data 𝓕 sh_𝓛 t t^flat where
  Avoid_S    :: IsScalar t
             ⇒ 𝓕 sh_𝓛 t t                                      -- avoid flattening
  Avoid_A    :: IsScalar e
             ⇒ 𝓕 sh_𝓛 (Array sh e) (Array sh e)
  Regular    :: 𝓝 t (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)    -- regular context
             → 𝓕 sh_𝓛 t ( Array (sh_𝓛++sh_1) e_1
                         , Array (sh_𝓛++sh_2) e_2
                         , …
                         , Array (sh_𝓛++sh_n) e_n)
  Irregular :: 𝓝 t (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)    -- irregular context
             → 𝓕 sh_𝓛 t ( (Segments (sh_𝓛++sh_1), Vector e_1)
                         , (Segments (sh_𝓛++sh_2), Vector e_2)
                         , …
                         , (Segments (sh_𝓛++sh_n), Vector e_n))
  Tuple_𝓕    :: (𝓕 sh_𝓛 t_1 t_1^flat, 𝓕 sh_𝓛 t_2 t_2^flat, … , 𝓕 sh_𝓛 t_n t_n^flat)
             → 𝓕 sh_𝓛 (t_1, t_2, .., t_n) (t_1^flat, t_2^flat, … , t_n^flat)
```

The `Tuple`$_{\mathcal{F}}$ constructor may at first glance seem unnecessary, but it allows for is different components of tuples to be flattened independently. This is crucial as one component might be used in a regular context, while another in an irregular context; similarly one might use avoidance, while the others don't, et cetera. One example of this is:

```
foo :: Vector Int → (Vector (Vector Int), Vector (Vector Int))
foo xs = ( generate (Z:.10) (λ_ → xs)
         , map (λx → generate (Z:.x) (_ → x)) xs)
```

Here we are constructing a pair of nested vectors from a single non-nested vector of `Int`s. The first component of the pair is of length 10 with each vector inside it `xs`. Because each sub-vector is the same length it is regular and thus can use a regular representation after flattening. The second component of the pair is of the same length as `xs`, but with each sub-vector of length equal to an element of `xs`. Just looking at this function in isolation, we have no way of knowing the contents of `xs` ahead of time, so we have to treat the second component as irregular.

To further underpin this, consider how tuples are often used in functional programs. A programmer would expect

```
bar = let a = A
          b = B
      in C
```

to be the same as

```
bar' = let (a,b) = (A, B)
       in C
```

Assuming that B does not depend on A (or vice versa), it would be undesirable for the simple introduction of a tuple grouping related bindings to introduce substantial performance changes. Similarly, *currying* or *uncurrying* functions should not alter their performance characteristics.

Flattening avoidance is covered by the $\texttt{Avoid}_{\mathcal{S}}$ and $\texttt{Avoid}_{A}$ alternatives, where we keep the type the same. We don't need to normalise as the `e` in $\texttt{Avoid}_{\mathcal{S}} :: \mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{e e}$ and $\texttt{Avoid}_{A} :: \mathcal{F}\ \texttt{sh}_{\mathcal{L}}$ `(Array sh e) (Array sh e)` is guaranteed to be scalar by the `IsScalar` constraint.

The $\mathcal{F}$ type is not a singleton. That is to say, there are multiple witness for some type relations. For example,

$\mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{(Int, Int) (Int, Int)}$

has both

$\texttt{Avoid}_{\mathcal{S}}$

and

$\texttt{Tuple}_{F}\ (\texttt{Avoid}_{\mathcal{S}},\ \texttt{Avoid}_{\mathcal{S}})$

as inhabitants. This can be a problem when trying to resolve whether a particular type is *completely* avoided: when flattening was avoidable for all components. We resolve this by introducing a *smart* constructor:

$$\begin{aligned}\texttt{tuple}_{F} :: &(\mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{t}_1\ \texttt{t}_1^{flat},\ \mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{t}_2\ \texttt{t}_2^{flat},\ \ldots,\ \mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{t}_n\ \texttt{t}_n^{flat})\\ &\rightarrow \mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ (\texttt{t}_1,\ \texttt{t}_2,\ \ldots,\ \texttt{t}_n)\ (\texttt{t}_1^{flat},\ \texttt{t}_2^{flat},\ \ldots,\ \texttt{t}_n^{flat})\end{aligned}$$

This function will check its arguments to see if they are all $\texttt{Avoid}_{\mathcal{S}}$ and return $\texttt{Avoid}_{\mathcal{S}}$ if they are. This excludes the second inhabitant above and enables a simple check for avoidance of scalars.

4.4.2 The lifting transformation

The lifting transform $\mathcal{L}$ takes a term in our language and yields a term in the same language of a $\mathcal{F}$-related type that contains no nested parallelism nor nested arrays. Its complete type is:

$$\mathcal{L}[\![.]\!] :: \texttt{Expr}\ \Gamma\ \texttt{t} \rightarrow \texttt{Env}\ \texttt{sh}_{\mathcal{L}}\ \Gamma\ \Gamma^{flat} \rightarrow (\exists\ \texttt{t}^{flat}\ .\ (\mathcal{F}\ \texttt{sh}_{\mathcal{L}}\ \texttt{t}\ \texttt{t}^{flat},\ \texttt{Expr}\ \Gamma^{flat}\ \texttt{t}^{flat}))$$

This transformation has many components, so we'll look at each one in turn. The first argument of type $\texttt{Expr}\ \Gamma\ \texttt{t}$ is the typed abstract syntax (AST) of the core language term that is to be flattened, where Γ is a type-level list of the free variables in the term and `t` is its type.

The second argument, of type $\texttt{Env}\ \texttt{sh}_{\mathcal{L}}\ \Gamma\ \Gamma^{flat}$, is an environment that relates the types of the free variables Γ to their flattened form Γ^{flat}. Like the $\mathcal{F}$ relation, it is also parameterised by the lifting shape.

Finally, the result combines the witness for the flattened result type $\mathtt{t}^{flat}$ with the lifted term $\mathtt{Expr}\ \Gamma^{flat}\ \mathtt{t}^{flat}$, whose type parameters have been flattened to match, establishing type-preservation of the transformation.

The structure of the environment $\mathtt{Env}\ \mathtt{sh}_{\mathcal{L}}\ \Gamma\ \ \Gamma^{flat}$ is somewhat more involved than usual, due to special requirements during flattening. Specifically, we need to handle access variables to brought into scope with a `generate` outside the most immediate one. To give an example for this:

```
mults :: Expr [] (Vector (Vector Int))
mults = generate (Z:.10) (λv_0 → generate v_0 (λv_0 → indexHead v_1 * indexHead v_0))
```

Here we're constructing the lower triangle of a multiplication table. If we want to flatten this, we have to lift the inner `generate` from 1D to 2D, but not lift the outer generate at all. However, in the body of the inner generate we are referring to a variable bound by the outer generate. This presents a problem. The variable was bound at a point which is not being lifted to the same dimension. We partially resolve this by tracking lifting shape in the environment, but this doesn't take into account the actual *extent* of the outer array. Because of this, we track both regular (the $(:_R)$ alternative) and irregular contexts (the $(:_{Ir})$ alternative). When the lifting transform encounters a generate it must extend the lifting shape by the shape of the generated array. The generation function can then be lifted under this larger context.

```
data Env sh_𝓛 Γ Γ^flat  where
  []     :: Env Z [] []
  (:)    :: 𝓕  t  t^flat                        -- standard free variable
         → Env sh_𝓛 Γ Γ^flat
         → Env sh_𝓛 (t : Γ) (t^flat  : Γ^flat )
  (:_R)  :: Shape sh                           -- regular flattening context
         ⇒ Expr Γ^flat  sh
         → Env sh_𝓛 Γ Γ^flat
         → Env (sh_𝓛++sh) Γ Γ^flat
  (:_Ir) :: Shape sh                           -- irregular flattening context
         → Expr Γ^flat  (Segments sh)
         → Env sh_𝓛 Γ Γ^flat
         → Env (sh_𝓛++sh) Γ Γ^flat
```

A key insight of the environment structure is that a closed term can only be lifted into `Z`. Intuitively, this makes sense. We only need to lift things into a higher dimension when it is used in a nested context. A closed term, by definition, is not used in any context, nested or otherwise.

The use of this environment structure can be seen in the definition of the auxiliary function `var` defined in Figure 4.5, which looks up free variables in the given environment. In addition to that conventional purpose, it also replicates the variable according to each enclosing context; i.e., while traversing the environment to look up the variable, it inserts a $\mathtt{replicate}_R$ and $\mathtt{replicate}_{Ir}$ for every $(:_R)$ and $(:_{Ir})$ that it comes across, respectively.

The transformation rules of the flattening transformation are given in Figure 4.7. What

```
-- Lifting variables
var :: Env sh_ℒ Γ Γ^flat  → Var Γ t → (∃ t^flat . (𝓕 sh_ℒ t t^flat , Expr Γ^flat  t^flat ))
var (r    :   _  ) v_0    = (r, v_0)
var (_    :   env) v_n+1  | (r, e) ← var env v_n
                          = (r, weaken e)
var (sh   :_R  env) v     | (r, e) ← var env v         -- Gone past a level of regular nesting
                          = replicate_R r sh e
var (segs :_Ir env) v     | (r, e) ← var env v         -- Gone past a level of irregular nesting
                          = replicate_Ir r segs e

-- Replication under a regular context
replicate_R  :: 𝓕 sh_ℒ t t^flat
             → Expr Γ sh
             → Expr Γ t^flat
             → (∃ t^flat_new . (𝓕 (sh_ℒ⧺sh) t t^flat_new , Expr Γ t^flat_new ))
replicate_R Avoid_S            _  t = (Avoid_S, t)
replicate_R Avoid_A            _  t = (Avoid_A, t)
replicate_R (Regular r)        sh t = (Regular r, replicate_R→R r sh t)
replicate_R (Irregular r)      sh t = (Irregular r, replicate_Ir→R r sh t)
replicate_R (Tuple (r_1,..))   sh t | (r'_1, t_1) ← replicate_R r_0 sh (prj 1 t)
                                    , …
                                    = (Tuple (r'_1, … ), (t_1, … ))

-- Replicate a regular array under a regular context
replicate_R→R :: Expr Γ sh
              → Expr Γ (Array (sh_ℒ⧺sh_1) e_1, … )
              → Expr Γ (Array (sh_ℒ⧺sh⧺sh_1) e_1, … )
replicate_R→R sh arr = ( generate (inside sh' (extent (prj 1 arr))) (λix → prj 1 arr ! outside ix)
                       , … )

-- Replicate an irregular array under a regular context
replicate_Ir→R :: Expr Γ sh
               → Expr Γ ((Segments (sh_ℒ⧺sh_1), Vector e_1), … )
               → Expr Γ ((Segments (sh_ℒ⧺sh⧺sh_1), Vector e_1), … )
replicate_Ir→R sh arr = replicate_Ir→Ir (segmented (generate Z sh)) arr

-- Replication under an irregular context
replicate_Ir  :: 𝓕 sh_ℒ t t^flat
              → Expr Γ (Segments sh)
              → Expr Γ t^flat
              → (∃ t^flat_new . (𝓕 (sh_ℒ⧺sh) t t^flat_new , Expr Γ t^flat_new ))
replicate_Ir Avoid_S            _  t = (Avoid_S, t)
replicate_Ir Avoid_A            _  t = (Avoid_A, t)
replicate_Ir (Regular r)        sh t = (Regular r, replicate_R→Ir r sh t)
replicate_Ir (Irregular r)      sh t = (Irregular r, replicate_Ir→Ir r sh t)
replicate_Ir (Tuple (r_1,..))   sh t | (r'_1, t_1) ← replicate_Ir r_0 sh (prj 1 t)
                                     , …
                                     = (Tuple (r'_1, … ), (t_1, … ))

-- Replicate a regular array under an irregular context
replicate_R→Ir :: Expr Γ (Segments sh)
               → Expr Γ (Array (sh_ℒ⧺sh_1) e_1, … )
               → Expr Γ ((Segments (sh_ℒ⧺sh⧺sh_1), Vector e_1), … )
replicate_R→Ir segs arr = replicate_Ir→Ir segs (irregular arr)

-- Replicate an irregular arrays under an irregular context
replicate_Ir→Ir :: Expr Γ (Segments sh)
                → Expr Γ ((Segments (sh_ℒ⧺sh_1), Vector e_1), … )
                → Expr Γ ((Segments (sh_ℒ⧺sh⧺sh_1), Vector e_1), … )
replicate_Ir→Ir segs arr = ( let segs_old = prj 1 (prj 1 arr)
                                 segs_new = insides segs segs_old
                                 vals_old = prj 2 (prj 1 arr)
                             in (segs_new, generate_seg segs_new (λix → vals_old ! segs_old # outside ix))
                           , … )
```

Figure 4.5: Lifting variables into higher dimensions.

```
map :: (∀ Γ^flat . Expr Γ^flat  a → Expr Γ^flat  b)
    → Expr Γ (Array sh a)
    → Expr Γ (Array sh b)
map f as = generate (extent as) (λv_0. f (weaken as ! v_0))

map_seg  :: (∀ Γ^flat . Expr Γ^flat  a → Expr Γ^flat  b)
       → Expr Γ (Segments sh, Vector b)
       → Expr Γ (Segments sh, Vector a)
map_seg  f as = let segs = prj 1 as
             in ( segs, generate_seg  segs (λv_0. f (weaken as ! Z :. weaken segs # v_0)))

zipWith :: (∀ Γ^flat . Expr Γ^flat  a → Expr Γ^flat  b → Expr Γ^flat  c)
        → Expr Γ (Array sh a)
        → Expr Γ (Array sh b)
        → Expr Γ (Array sh c)
zipWith f as bs = generate (intersect (extent as) (extent bs)) (λv_0. f (weaken as ! v_0) (weaken bs ! v_0))

zipWith_seg  :: (∀ Γ^flat . Expr Γ^flat  a → Expr Γ^flat  b → Expr Γ^flat  c)
        → Expr Γ (Segments sh, Vector a)
        → Expr Γ (Segments sh, Vector b)
        → Expr Γ (Segments sh, Vector c)
zipWith_seg  f as bs =
  let segs = intersects (prj 1 as) (prj 1 bs)
  in ( segs, generate_seg  segs (λv_0. f (weaken as ! Z :. weaken segs # v_0) (weaken bs ! Z :. weaken segs # v_0)))

unit :: IsScalar a ⇒ Expr Γ a → Expr Γ (Array Z a)
unit a = generate Z (λ_. (weaken a))

inside :: Expr Γ sh' → Expr Γ (sh++sh'') → Expr Γ (sh++sh'++sh'')
inside sh' sh = indexLeft sh ++ sh' ++ indexRight sh

outside :: Expr Γ (sh++sh'++sh'') → Expr Γ (sh++sh'')
outside sh = indexLeft sh ++ indexRight sh

insides :: Expr Γ (Segments sh') → Expr Γ (sh++sh'') → Expr Γ (sh++sh'++sh'')
insides segs_inner segs_outer = lefts segs_outer `cross` segs_inner `cross` rights segs_outer

-- Add a fresh variable to the environment
weaken :: Expr Γ t → Expr (a : Γ) t
```

Figure 4.6: Auxiliary operations implemented as meta-functions

```
$\mathcal{L}[\![$c$]\!]$ _   = (Avoid$_S$, c)
$\mathcal{L}[\![$v$]\!]$ env = var env v
$\mathcal{L}[\![$p$_1$ e$]\!]$ env
  | (Avoid$_S$, e$^{flat}$)                ← $\mathcal{L}[\![$e$]\!]$ env
  = (Avoid$_S$, p$_1$ e$^{flat}$)
  | (Regular Scalar, e$^{flat}$)    ← $\mathcal{L}[\![$e$]\!]$ env
  = (Regular Scalar, map p$_1$ e$^{flat}$)
  | (Irregular Scalar, e$^{flat}$) ← $\mathcal{L}[\![$e$]\!]$ env
  = ( Irregular Scalar
    , (prj 1 e$^{flat}$, map p$_1$ (prj 2 e$^{flat}$)))
$\mathcal{L}[\![$p$_2$ e$_1$ e$_2]\!]$
  | (r$_1$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$
  , (r$_2$, e$_2^{flat}$) ← $\mathcal{L}[\![$e$_2]\!]$
  = liftBinOp p$_2$ r$_1$ r$_2$ e$_1^{flat}$ e$_2^{flat}$
$\mathcal{L}[\![$extent e$]\!]$ env
  | (Avoid$_A$, e$^{flat}$)                 ← $\mathcal{L}[\![$e$]\!]$ env
  = (Avoid$_A$, extent e$^{flat}$)
  | (Regular (Nest _), e$^{flat}$)     ← $\mathcal{L}[\![$e$]\!]$ env
  = (Avoid, extent$_R$ e$^{flat}$)
  | (Irregular (Nest _), e$^{flat}$) ← $\mathcal{L}[\![$e$]\!]$ env
  = (Irregular Scalar, extent$_{Ir}$ e$^{flat}$)
$\mathcal{L}[\![$e$_1$ ! e$_2]\!]$ env
  | (Avoid$_A$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  , (Avoid$_S$, e$_2^{flat}$)              ← $\mathcal{L}[\![$e$_2]\!]$ env
  = (Avoid$_S$, e$_1^{flat}$ ! e$_2^{flat}$)
  | (Regular (Nest r), e$_1^{flat}$)     ← $\mathcal{L}[\![$e$_1]\!]$ env
  , (Regular Scalar, e$_2^{flat}$)       ← $\mathcal{L}_F[\![$e$_2]\!]$ env
  = (Regular r, e$^{flat}$ !$_R$ e$_2^{flat}$)
  | (Irregular (Nest r), e$_1^{flat}$) ← $\mathcal{L}_F[\![$e$_1]\!]$ env
  , (Irregular Scalar, e$_2^{flat}$)   ← $\mathcal{L}_F[\![$e$_2]\!]$ env
  = (Irregular r, e$^{flat}$ !$_{Ir}$ sh')
$\mathcal{L}[\![$(let v$_0$ = e$_1$ in e$_2]\!]$ env
  | (r$_1$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  , (r$_2$, e$_2^{flat}$) ← $\mathcal{L}[\![$e$_2]\!]$ (r$_1$ : env)
  = (r$_2$, let v$_0$ = e$_1^{flat}$ in e$_2^{flat}$)
$\mathcal{L}[\![$(e$_1$,…,e$_n$)$]\!]$ env
  | (r$_0$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  …
  , (r$_n$, e$_n^{flat}$) ← $\mathcal{L}[\![$e$_n]\!]$ env
  = (tuple$_F$ (r$_1$,…,r$_n$), (e$_1^{flat}$,…,e$_n^{flat}$))
```

```
$\mathcal{L}[\![$prj l e$]\!]$ env
  | (Avoid$_S$, e$^{flat}$)                 ← $\mathcal{L}[\![$e$]\!]$ env
  = (Avoid$_S$, prj l e$^{flat}$)
  | (Tuple (..,r$_\mathcal{L}$,..), e$^{flat}$) ← $\mathcal{L}[\![$e$]\!]$ env
  = (r$_\mathcal{L}$, prj l e$^{flat}$)
$\mathcal{L}[\![$generate e$_1$ (λ v$_0$. e$_2$)$]\!]$ env
  -- Flattening is avoidable
  | (Avoid$_S$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  , (Avoid$_S$, e$_2^{flat}$) ← $\mathcal{L}[\![$e$_2]\!]$ (Avoid : env)
  = (Avoid$_A$, generate e$_1^{flat}$ (λ v$_0$. e$_2^{flat}$))
  -- Regular array in a regular context
  | Left ctx          ← context env
  , (Avoid$_S$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  , (r, e$_2^{flat}$)          ← $\mathcal{L}_F[\![$e$_2]\!]$ (Regular Scalar : e$_1^{flat}$ :$_R$ env)
  = (nest r, let v$_0$ = enum$_R$ ctx e$_1^{flat}$ in e$_2^{flat}$)
  -- Array is irregular or in irregular context
  | (r$_1$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ env
  , segs              ← segmentsOf r$_1$ e$_1^{flat}$
  , (r$_2$, e$_2^{flat}$) ←
      $\mathcal{L}_F[\![$e$_2]\!]$ (Irregular Scalar : segs :$_{Ir}$ env)
  = (nest r$_2$, let v$_0$ = enum$_{Ir}$ (context env) segs in e$_2^{flat}$)
$\mathcal{L}[\![$fold (λv$_1$ v$_0$. e$_1$) e$_2$ e$_3]\!]$ env
  | (Avoid$_S$, e$_2^{flat}$) ← $\mathcal{L}[\![$e$_2]\!]$ env
  , (Avoid$_S$, e$_1^{flat}$) ← $\mathcal{L}[\![$e$_1]\!]$ (Avoid : env)
  , (Avoid$_A$, e$_3^{flat}$) ← $\mathcal{L}[\![$e$_3]\!]$ env
  = (Avoid$_A$, fold (λv$_1$ v$_0$. e$_1^{flat}$) e$_2^{flat}$ e$_3^{flat}$)
  | (Avoid$_S$, e$_2^{flat}$)                ← $\mathcal{L}[\![$e$_2]\!]$ env
  , (Avoid$_S$, e$_1^{flat}$)                ← $\mathcal{L}[\![$e$_1]\!]$ (Avoid : env)
  , (Regular (Nest r), e$_3^{flat}$) ← $\mathcal{L}[\![$e$_3]\!]$ env
  = (Regular r, fold (λv$_1$ v$_0$. e$_1^{flat}$) e$_2^{flat}$ e$_3^{flat}$)
  | (Avoid$_S$, e$_2^{flat}$)                  ← $\mathcal{L}[\![$e$_2]\!]$ env
  , (Avoid$_S$, e$_1^{flat}$)                  ← $\mathcal{L}[\![$e$_1]\!]$ (Avoid : env)
  , (Irregular (Nest r), e$_3^{flat}$) ← $\mathcal{L}[\![$e$_3]\!]$ env
  = (Irregular r, fold$_{seg}$ (λv$_1$ v$_0$. e$_1^{flat}$) e$_2^{flat}$ e$_3^{flat}$)
```

Figure 4.7: The lifting transformation

follows is an explanation of how each language construct is flattened.

Literals

This is the simplest construct in terms of flattening. When it encounters a literal value, flattening returns it as it is, paired with the `Avoid`$_{\mathcal{S}}$ constructor to signal that it does not need to be lifted.

Variables

When flattening a variable, the transformation refers to the auxiliary function `var` discussed previously. If one views the variable as just a simple Debruijn index then the variable itself is let unchanged but may have a series of replicates applied to it in order to make it *fit* the current context. Because our indices are typed, we need to produce a witness of the relation between the type of the variable in the original term and the type in the flattened term.

Let bindings

The let-binding rule shows how bindings are introduced and how they are tracked via `Env`. The binding is first flattened, giving a $\mathcal{F}$-witness. This witness is placed into the environment for flattening the body of the let-binding. This ensures that any occurrences of that variable are assigned the right type.

To avoid any confusion, the let-binding we are introducing here exists in the target language, not the meta language. We assume that the let-syntax of our meta language is overloaded to construct let bindings in either the meta language or the object language as necessary.

Tuples and tuple projection

In order to lift a tuple constructor, we lift each component and construct a new tuple containing them. Similarly to let bindings, we assume that our meta language overloads tuple construction to support construction in our object language.

In the case of tuple projection, it is similarly just a matter of projecting the same component from the lifted tuple.

Primitive operations

For unary operations, lifting is straightforward. We simply `map` the operation over the lifted operand.

It gets more complex with binary operations as we have the possibility that each operand ends up having a different representation after lifting. Each could be either avoided, regular or irregular. In total this gives us 9 different possible combinations, but we are able to cheat a bit by handling symmetric cases by swapping the arguments of the operation.

```
liftBinOp :: (IsScalar e_1, IsScalar e_2)
          ⇒ (Expr Γ e_1 → Expr Γ e_2 → Expr Γ e)
          → F sh_L e_1 e_1^flat
          → F sh_L e_2 e_2^flat
          → Expr Γ e_2^flat
          → Expr Γ e_2^flat
          → (∃ e^flat . (F sh_L e e^flat , Expr Γ e^flat ))
liftBinOp p r_1 r_2 e_1 e_2
  = case (r_1, r_2) of
      (Avoid_S,           Avoid_S)           → p e_1 e_2
      (Avoid_S,           Regular Scalar)    → map (λv_0. p e_1 v_0) e_2
      (Avoid_S,           Irregular Scalar)  → (prj 1 e_2, map (p e_1) (prj 2 e_2))
      (Regular Scalar,    Regular Scalar)    → zipWith p e_1 e_2
      (Regular Scalar,    Irregular Scalar)  → zipWith_seg p (irregular e_1) e_2
      (Irregular Scalar, Irregular Scalar)   → zipWith_seg p e_1 e_2
      _                                      → liftBinOp (flip p) r_2 r_1 e_2 e_1
```

Array shapes

With `extent`, if the term it is applied to does not need to be flattened then it just returns the original expression.

The more interesting case occurs when the argument expression flattens in a regular context. In this case we simply take the shape of the resulting array, and strip off the outer dimensions corresponding to the lifting shape and the inner dimensions corresponding to the original array. This is done with $\texttt{extent}_R$.

```
extent_R :: Expr Γ (Array (sh_L++(sh++sh_1)) e_1, ... )
        → Expr Γ sh
extent_R (arr, ... ) = indexLeft (indexRight (extent arr))
```

This is the one case where flattening can still be avoided even if an expression depends on an expression that had to be flattened.

We do something similar in the irregular case with $\texttt{extent}_{Ir}$, but here we cannot avoid flattening.

```
extent_Ir :: Expr Γ^flat ((Segments (sh_L++(sh++sh_1)), Vector e_1), ... )
         → Expr Γ^flat (Segments sh_L, Vector sh)
extent_Ir arr = expand (lefts (reassoc (prj 1 (prj 1 arr))))

reassoc :: Expr Γ (Segments (sh++(sh'++sh'')))
        → Expr Γ (Segments ((sh++sh')++sh''))
reassoc = id  -- encoded in type system
```

Here, we take the segment descriptors and using `lefts` extract new descriptors that ignore the inner dimensionality. We use `reassoc` to *reassociate* the ordering of each level of nesting. This is not necessary, as the type system handles this for us, but it helps us to see what is going on. We start with

```
Segments (sh_𝓛++(sh++sh_1))
```

We then reassociate to

```
Segments ((sh_𝓛++sh)++sh_1)
```

And remove the innermost level of nesting

```
Segments (sh_𝓛++sh)
```

The `expand` operation

```
expand :: Expr Γ (Segments (sh++sh')) → Expr Γ (Segments sh, Vector sh')
```

then lets us split this up into a set of segment descriptors that describe a mapping into a vector of extents.

Indexing

The rules for indexing `(!)` are defined in terms of $\mathcal{L}_F$, which enforces flattening by ignoring $\texttt{Avoid}_S$ and $\texttt{Avoid}_A$ in the flattening of the term e that constitutes the first argument of the indexing operation. The exact definition is in Figure 4.8

The lifted indexing itself is relatively straightforward. In the regular case, we take a series of slices of the input arrays.

```
(!_R) :: Expr Γ (Array (sh_𝓛++sh++sh_1) e, ... )
     → Expr Γ (Array sh_𝓛 sh)
     → Expr Γ (Array (sh_𝓛++sh_1) e, ... )
arr !_R ix = ( generate (outside (extent arr)) (λv_0 . prj 0 arr ! inside (ix ! indexLeft v_0) v_0)
            , ... )
```

We use `outside` as defined in 4.6 in order to construct an array of shape $\texttt{sh}_{\mathcal{L}}\texttt{++sh}_1$. Then by doing some index manipulation, we can pull out the corresponding elements from the input array.

For irregular indexing, it works in much the same way, except for needing to work with the segmented representation.

```
(!_Ir) :: Shape sh
      ⇒ Expr Γ ((Segments (sh_𝓛++sh++sh_1), Vector e_1), ... )
      → Expr Γ (Segments sh_𝓛, Vector sh)
      → Expr Γ ((Segments (sh_𝓛++sh_1), Vector e_1), ... )
arr !_Ir ix = ( let segs = (outsides (prj 0 (prj 1 arr)))
               in segs, generate_seg segs (λv_0 . prj 2 (prj 1 arr) ! inside (ix ! prj 1 ix # indexLeft v_0) v_0)
             , ... )
```

Array construction

The `generate` rule is one of the more interesting ones. In our language, it is the only way in which nesting can be introduced. All other operations either remove nesting or preserve

```
𝓛_F⟦.⟧ :: Expr Γ t → Env sh_𝓛 Γ Γ^flat → (∃ t^flat . (𝓕 sh_𝓛 t t^flat , Expr Γ t^flat ))
𝓛_F⟦e⟧ env | (r, e^flat ) ← 𝓛⟦e⟧ env
                = replicate r (context env) e^flat

context :: Env sh_𝓛 Γ Γ^flat → Either (Expr Γ^flat sh_𝓛) (Expr Γ^flat (Segments sh_𝓛))
context [] = Left Z
context (_ : env) =
  case context env of
    Left sh    → Left (weaken sh)
    Right segs → Right (weaken segs)
context (sh' :_R env) =
  case context env of
    Left sh → Left (sh ++ sh')
    Right segs → Right (segs `cross` segmented (unit sh))
context (segs' :_Ir env) =
  case context env of
    Left sh → Right (segments (unit sh) `cross` segs')
    Right segs → Right (segs `cross` segs')

replicate :: 𝓕 t t^flat
          → Either (Expr Γ sh_𝓛) (Expr Γ (Segments sh_𝓛))
          → Expr Γ t^flat
          → (∃ t^flat '. (𝓕 sh_𝓛 t t^flat ', Expr Γ t^flat '))
replicate Avoid_S                ctx t = replicate_S ctx t
replicate Avoid_A                ctx t = replicate_A ctx t
replicate (Tuple_F (r_1, .., r_n)) ctx t | (r_1^flat, t_1) ← replicate r_1 ctx (prj 1 t)
                                       , ...
                                       . (r_n^flat, t_n) ← replicate r_n ctx (prj n t)
                                       = (Tuple_F (r_1^flat, ... , r_n^flat) (t_1, ... , t_n))

replicate_S :: IsScalar t
            ⇒ Either (Expr Γ sh_𝓛) (Expr Γ (Segments sh_𝓛))
            → Expr Γ t
            → (∃ t^flat . (𝓕 sh_𝓛 t t^flat , Expr Γ t^flat ))
replicate_S (Left ctx)  t = (Regular Scalar, generate ctx (λ_ . weaken t))
replicate_S (Right ctx) t = (Irregular Scalar, generate_seg ctx (λ_ . weaken t))

replicate_A :: IsScalar e
            ⇒ Either (Expr Γ sh_𝓛) (Expr Γ (Segments sh_𝓛))
            → Expr Γ (Array sh e)
            → (∃ t^flat . (𝓕 sh_𝓛 (Array sh e) t^flat , Expr Γ t^flat ))
replicate_A (Left ctx)  t = ( Regular (Nest Scalar)
                            , backpermute (ctx ++ extent t) indexRight)
replicate_A (Right ctx) t = ( Irregular (Nest Scalar)
                            , let segs = ctx ++ segments (unit (extent t))
                              in (segs, generate_seg segs (ix → t ! indexRight ix)))
```

Figure 4.8: Forced lifting

it. Because of this, we have to handle a greater number of possible cases. These cases are handled by a backtrack search. First, we have to check if flattening can be avoided entirely, which would be the cheapest solution. This is only true if flattening can be avoided for both arguments and the second argument is scalar.

If it is not possible to avoid flattening for both arguments, but it is for the first argument (the extent of the output) then we can be smarter. We know that this occurrence of `generate` is not introducing any new irregular nesting. However, it may still be getting used in an irregular context, or its second argument (the generation function) may itself be introducing irregularity. For this reason, we need to use `context`, given in Figure 4.8, to determine whether we are in a regular or irregular context. If we are in regular context then we do a lifted enumeration with $\texttt{enum}_{R}$.

```
enum_R :: Shape sh ⇒ Expr Γ sh_L → Expr Γ sh → Expr Γ (Array (sh_L++sh) sh)
enum_R sh_L sh = generate (sh_L++sh) (λv_0 . indexRight v_0)
```

The intuition behind this function is it produces a set of enumerations we wish to generate values for. As an example,

```
enum_R 2 (Z:.2) (Z:.3) = { {Z:.0, Z:.1, Z:.2}
                         , {Z:.0, Z:.1, Z:.2} }
```

Here, we're producing a 2D array containing 2 enumerations of `Z:.3`. In 4.7, we use an enumeration like this to satisfy our generation function. This gives us the actual result we need, but in order to convince the type checker of the meta language of this fact, we have to do some manipulation of the $\mathcal{F}$ relation witness.

```
nest :: F (sh_L++sh) e e^flat
     → F sh_L (Array sh e) e^flat
nest (Regular r)            = Regular (Nest r)
nest (Irregular r)          = Irregular (Nest r)
nest (Tuple (r_1, ... , r_n)) = Tuple (nest r_1, ... , nest r_n)
```

This function, captures a key insight of the flattening transform. Namely, that a term of type `e` flattened under the context $\texttt{sh}_{\mathcal{L}}\texttt{++sh}$ would have the same possible flattened representations as a term of type `Array sh e` flattened under the context $\texttt{sh}_{\mathcal{L}}$.

We are left with the final case. This occurs when flattening cannot be avoided for the first argument. In such a case, irregular nesting is being introduced. We must extract segment descriptors from the flattened first argument with `segmentsOf`

```
segmentsOf :: Shape sh
           ⇒ F sh_L sh sh^flat
           → Expr Γ sh^flat
           → Expr Γ (Segments sh)
segmentsOf Avoid_S sh             = segmented (unit sh)
segmentsOf (Regular Scalar) sh   = segmented sh
segmentsOf (Irregular Scalar) sh = prj 1 sh `cross` segmented (prj 2 sh)
```

We then use these segments, and the current lifting shape to construct an segmented vector of enumerations.

```
enum_Ir :: Either (Expr Γ sh_L) (Expr Γ (Segments sh_L))
        → Expr Γ (Segments sh)
        → Expr Γ (Segments (sh_L++sh), Vector sh)
enum_Ir (Left ctx)  segs = let segs' = segments (unit ctx) `cross` segs
                           in (segs', generate_seg segs' (λv_0. indexRight v_0))
enum_Ir (Right ctx) segs = let segs' = ctx `cross` segs
                           in (segs', generate_seg segs' (λv_0. indexRight v_0))
```

The backtracking we rely on for lifting `generate` could, in theory, lead to exponential work complexity. In the implementation described in chapter 5, this is not a problem, as there is no deep nesting of `generate`-expressions. Keller et al. [25], who have to distinguish between only two cases, circumvent backtracking by splitting the transformation into an analysis phase which first labels the expression, followed by a separate lifting phase. Such an approach would be compatible with what is described here and likely be necessary for a language where deep nesting was more prevalent.

4.4.3 Array reduction

For `fold`, the type system ensures that the function it is applied to is a sequential computation over scalars, and the initial value is a scalar as well. This does not however, preclude the possibility that one of them may contain parallelism. This presents a problem for us as a fold with a reduction function containing parallelism is not typically supported by data parallel models of computing. For this reason, we leave the transform as partial in this regard. Flattening will fail for programs that rely on parallelism in reduction functions. This is true of other nested data parallel languages (e.g. Data Parallel Haskell [7] and NESL [5]),

Flattening `folds` is relatively straightforward compared to `generates`. It all depends on the 3rd argument. If the flattening of it can be avoided, then the entire fold can be avoided. If it has a regular flattened representation, then it can be reduced just by using a normal `fold`. If the representation is irregular, then $\texttt{fold}_{seg}$ is used.

Chapter 5

Implementing and Optimising

Now that we have a foundation for regularity aware flattening of nested data parallel programs we will explore what is required to use our program transform in an already implemented, expressive, powerful, high-performance array language and compiler. We will provide a high-level overview of the the implementation. For more detailed information we refer to the publicly available implementation.[1]

In this chapter, we describe:

- How sequences fit into Accelerate's existing compiler infrastructure (Section 5.1).
- Extensions to Accelerate's optimisation passes that help achieve competitive absolute performance for our sequence programs (Section 5.4).
- A dynamic scheduler for executing sequence computations (Section 5.5).

5.1 Architecture

As discussed in Chapter 2, Accelerate has a fairly standard compiler architecture: a multi-stage pipeline. Prior to our extension, this pipeline consisted of these stages.

- **Sharing recovery** – Converts the higher order abstract syntax (HOAS) into first order syntax with Debruijn indices and also recovers sharing.
- **Fusion and optimisation** – Tries to minimise unnecessary array traversals through fusion, folds constant expressions, performs Constant Subexpression Elimination (CSE), and shrinks terms.
- **Code generation** – Generates LLVM IR from AST and passes it off to the LLVM compiler. Binaries produced are either x86 (with vector instructions) or PTX, for CUDA capable GPUs.

[1]This is available at `https://github.com/AccelerateHS/accelerate/tree/feature/sequences`

- **Execution** – Schedules and executes the binaries on the target hardware.

To support sequence computations, we both have to extend the AST with a set of sequence combinators and change the pipeline to support them. The changes needed, at the high-level, are:

- An update sharing recovery to support the additional sequence AST constructors. This will not be explored in detail as it is not novel or different to how Accelerate handles other AST forms. See McDonell et al. [30] for details on how sharing recovery works.
- An additional flattening stage after sharing recovery to flatten the computations embedded in sequence computations (Section 5.3).
- A fusion system extension to exploit additional fusion opportunities presented by sequence computations (Section 5.4).
- An additional runtime system component for scheduling sequence computations in a way where memory usage can be minimised while maximising utilisation (Section 5.5).

5.2 What is a sequence?

As introduced in Section 3.2, a sequence computation `Seq [Array sh b]` adds a second level of nested, irregular data-parallelism on top of the flat regular parallelism of an array computation `Acc (Array sh b)`. Hence, we can generally regard such a sequence computation as a mapping of a flat, regular function `f` over an irregular sequence `xs`:

```
Seq [Array sh b]  ~  mapSeq f xs
  where
    f  :: Acc (Array sh' a) → Acc (Array sh b)
    xs :: Seq [Array sh' a]
```

In addition to the *inner-function* parallelism in `f`, we want to exploit as much of the parallelism of the outer `mapSeq` (the *intra-function* parallelism) as possible on a given architecture to achieve optimal performance. That is, we execute multiple —but not necessarily *all*— elements of the `mapSeq` in parallel. Avoiding the need to utilise all outer parallelism helps us to support out-of-core datasets on GPUs, by only loading into device memory those elements of the sequence `xs` that are processed together. By dynamically adjusting the number of elements operated on at once, we aim to minimise memory usage while keeping all processing elements fully utilised. The flattening transformation takes the definition of the function `f` and rewrites it to a definition for $\texttt{f}^{\uparrow}$, such that semantically $\texttt{f}^{\uparrow}$ `= mapSeq f`; in other words, $\texttt{f}^{\uparrow}$ can process multiple elements of `xs` at once, in parallel.

5.3 Flattening

The design of the flattening compiler stage closely follows the what is described in 4.4. However, it differs in a few key areas.

- There are considerably more combinators in Accelerate than Acc$_{NDP}$. This means there needs to be lifted versions of nearly all combinators. In a lot of cases there is both a regular and irregular lifted equivalent.
- There are no nested arrays and sequences are only one level of nesting. This means that the implementation of the transform can be simplified in a few places, but fundamentally remains the same.
- Accelerate uses representation types to handle tuples of multiple arity. In Acc$_{NDP}$ we just assumed that we could easily generalize over tuples of arbitrary arity.

We address these points as we give an overview of the flattening implementation. Like we did with Acc$_{NDP}$, we will begin by first describing the type level component before moving on to the actual program transform.

5.3.1 Types

Recall that, to represent values in our language at a higher dimension, we had to first normalise then flatten types. This was captured by these relations:

```
data 𝒩 t t' where
  Scalar :: IsScalar e
         ⇒ 𝒩 e (Array Z e)
  Nest   :: 𝒩 e (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)
         → 𝒩 (Array sh e) (Array (sh ++ sh_1) e_1, Array (sh ++ sh_2) e_2, … , Array (sh ++ sh_n) e_n)
  Tuple_𝒩 :: (𝒩 t_1 t_1', 𝒩 t_2 t_2', … , 𝒩 t_n t_n')
         → 𝒩 (t_1, t_2, … , t_n) (t_1', t_2', … , t_n')

data ℱ sh_ℒ t t^flat where
  Avoid_𝒮   :: IsScalar t
            ⇒ ℱ sh_ℒ t t                                           -- avoid vectorisation
  Avoid_𝒜   :: IsScalar e
            ⇒ ℱ sh_ℒ (Array sh e) (Array sh e)
  Regular   :: 𝒩 t (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)   -- regular context
            → ℱ sh_ℒ t ( Array (sh_ℒ++sh_1) e_1
                       , Array (sh_ℒ++sh_2) e_2
                       , …
                       , Array (sh_ℒ++sh_n) e_n)
  Irregular :: 𝒩 t (Array sh_1 e_1, Array sh_2 e_2, … , Array sh_n e_n)   -- irregular context
            → ℱ sh_ℒ t ( (Segments (sh_ℒ++sh_1), Vector e_1)
                       , (Segments (sh_ℒ++sh_2), Vector e_2)
                       , …
                       , (Segments (sh_ℒ++sh_n), Vector e_n))
  Tuple_ℱ   :: (ℱ sh_ℒ t_1 t_1^flat, ℱ sh_ℒ t_2 t_2^flat, … , ℱ sh_ℒ t_n t_n^flat)
            → ℱ sh_ℒ (t_1, t_2, .., t_n) (t_1^flat, t_2^flat, … , t_n^flat)
```

Because of the differences between Accelerate and Acc$_{NDP}$, in the implementation we combine both of these relations into a single GADT.

```
data LiftedType t t' where
   UnitT       ::                      LiftedType ()           ()
   LiftedUnitT ::                      LiftedType ()           (Scalar Int)
   AvoidedT    :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Array sh e)
   RegularT    :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Array (sh:.Int) e)
   IrregularT  :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Segments sh, Vector e)
   TupleT      :: (IsProduct Arrays t, IsProduct Arrays t')
               ⇒ LiftedTupleType (TupleRepr t) (TupleRepr t')
               → LiftedType t t'
```

It is important to note that this structure is not parameterised by the lifting shape. This is because of the lack of deep nesting in Accelerate. As our only nested structure is the sequence, and sequences themselves cannot be nested, the only possible nested context is being an element of a sequence. Hence, we can represent all possible lifting shapes with an `Int`.

The first two constructors deal with the unit type. This is something we didn't encounter in Chapter 4. In short, the unit type (`()`) in a lifted context can just be represented by the lifting shape. There is only one possible unit type, so a collection of elements of that type can be represented by the shape of the collection. For a sequence, that context is just an `Int` indicating the length of the sequence (or the size of the current chunk of the sequence; more on that below).

The `AvoidedT` constructor is equivalent to $\mathtt{Avoid}_A$. In the array language strata of Accelerate (`Acc`), scalar values do not occur so $\mathtt{Avoid}_S$ is not needed.

`RegularT` and `IrregularT` are equivalent to `Regular` and `Irregular` respectively, but without being generalised over tuples. For the regular case, we simply add the lifting shape to the dimensionality of the array. This is equivalent to $\mathtt{sh}_\mathcal{L}\mathtt{++sh}$ but $\mathtt{sh}_\mathcal{L}$ in this case is just `Z:.Int`. For the irregular case, we have a slightly different notion of segment descriptors than before. The lifting shape is implicit in the constructor. What this means is that `Segments (sh:.Int)` in Acc$_{NDP}$ is equivalent to `Segments sh` in Accelerate. The outer dimension is assumed to always be present.

To correctly handle tuples, we have the `TupleT` constructor. It captures the `IsProduct` constraint on both the original type `t` and the transformed type `t'`. In essence, it means they they are both products of which all components are members of the `Arrays` type class. To specify the lifted type of each component in this product we use $\mathcal{L}$`edTupleType`.

```
data LiftedTupleType t t' where
   NilLtup  :: LiftedTupleType () ()
   SnocLtup :: (Arrays a, Arrays a')
            ⇒ LiftedTupleType t t'
            → LiftedType a a'
            → LiftedTupleType (t,a) (t',a')
```

Recall from Section 2.3 that the way Accelerate uses representation types is that tuples are represented as left nested tuples.

5.3.2 Segment representation

In Chapter 4 we left the issue of how to represent segment descriptors as abstract and implementation specific. Now that we are discussing the implementation, we must address that. We start by giving the most general definition of segment descriptors and then show how, by specialising it under the properties of sequences, we can arrive at a more optimal definition. To start with, let's suppose we want to represent segment descriptors for an irregular 3D structure. To do that, a simple definition is this.

```
Segments_NDP (Z:.Int:.Int:.Int) = (Vector Int, Vector Int)
```

Given a 3D index, we can determine the position of the corresponding value in the value vector with #:

```
(#) :: Segments_NDP (Z:.Int:.Int:.Int) → Z:.Int:.Int:.Int → Int
(k,j) @ (Z:.z:.y:.x) = let kz = k !! z
                           jy = j !! (kz + y)
                           ix = jy + x
                       in ix
```

Suppose our structure has an outer dimension of size l, the average size of the middle dimension is m, and the average size of the inner dimension is n. This representation for segment descriptors is a vector of length l and a vector of max length $l \times m$.

However, the best representation depends on precisely where the irregularity is. For example, the following representation,

```
Segments_NDP (Z:.Int:.Int:.Int) = Vector (Z:.Int:.Int, Int)
```

works as long as the 3D structure is a vector of 2D arrays. The final offset can be calculated like so

```
(#) :: Segmments_NDP (Z:.Int:.Int:.Int) → Z:.Int:.Int:.Int → Int
segs # (Z:.z:.y:.x) = let (sh,off) = segs !! z
                      in off + toIndex sh (Z:.y:.x)
```

This is the multidimensional equivalent of the size-offset representation in Chapter 2. This representation is of size l. Therefore it's space requirement is significantly less than the more general representation. Furthermore, implementing `(#)` only requires a single indexing operation.

Returning to our concept of sequences, we see that we only need this second way of representing the segment descriptors. Sequences can only contain regular arrays, so we only have irregularity in the outer dimension. Therefore our definition for `Segments` is

```
Segments_NDP (sh:.Int) = Vector (sh, Int)
```

For Accelerate, we make some small changes. Firstly, as we only ever have segment descriptors of dimension at least one, we can ignore that part of the parameter. Secondly, as shown in Section 2.1, there is benefit to storing the offsets alongside the extents. Lastly, we often need access to the total size of all segments. This can be calculated by taking the last offset and the size of the last shape and adding them together, but doing this calculation each time we need this value interferes with other optimisations. Hence, we store this value separately.

```
Segments sh = (Vector sh, Vector Int, Scalar Int)
```

Our final definition is a vector containing the extent of each segment, another vector containing the offset of each segment, and an `Int` representing the total size of all segments.

5.3.3 Transformation

With a concrete representation for segment descriptors, we are able to fully implement the lifting transform in Accelerate.

First we will consider the environment. Recall Acc$_{NDP}$'s environment:

```
data Env sh_ℒ Γ Γ^flat where
  []     :: Env Z [] []
  (:)    :: 𝓕 t t^flat                    -- standard free variable
         → Env sh_ℒ Γ Γ^flat
         → Env sh_ℒ (t : Γ) (t^flat : Γ^flat )
  (:_R)  :: Shape sh                     -- regular flattening context
         ⇒ Expr Γ^flat sh
         → Env sh_ℒ Γ Γ^flat
         → Env (sh_ℒ++sh) Γ Γ^flat
  (:_Ir) :: Shape sh                     -- irregular flattening context
         → Expr Γ^flat (Segments sh)
         → Env sh_ℒ Γ Γ^flat
         → Env (sh_ℒ++sh) Γ Γ^flat
```

In Accelerate, we have the same structure. The main difference is that we don't making the lifting shape explicit in the type.

```
data Context acc aenv aenv' where
  BaseC     :: Context acc aenv aenv

  PushC     :: Arrays t'
            ⇒ Context acc aenv aenv'
            → LiftedType t t'
            → Nesting acc (aenv', t')
            → Context acc (aenv, t) (aenv', t')

data Nesting acc aenv = NoNest
                      | RegularNest (acc aenv (Scalar Int))
                      | ∀ sh. Shape sh ⇒ IrregularNest (acc aenv (Segments sh))
```

We also combine the different cons-like constructors into one with an additional *nesting indicator* that indicates whether this cons cell also captures nesting details. The `acc` type parameter exists as a consequence of the *tying the recursive knot* trick described in Section 2.3.

The transform itself proceeds in much the same way as for Acc_{NDP}. The result of the lifting transform, which in Acc_{NDP} was

$$\exists\ \mathtt{t}^{flat}\ .\ (\mathcal{F}\ \mathtt{sh}_{\mathcal{L}}\ \mathtt{t}\ \mathtt{t}^{flat}\ ,\ \mathtt{Expr}\ \Gamma^{flat}\ \mathtt{t}^{flat})$$

can be directly converted into this GADT.

```
data LiftedAcc acc aenv t where
  LiftedAcc :: Arrays t' ⇒ LiftedType t t' → acc aenv t' → LiftedAcc acc aenv t
```

5.3.4 Lifted operations

In Acc_{NDP} we had to have $\mathtt{fold}_{seg}$ and $\mathtt{generate}_{seg}$ as the segmented versions of existing operations. In Accelerate we require both of these, as well as segmented versions of a number of other combinators. Some of these are available as language primitives, meaning each backend has its own implementation. Others, however, have to be implemented with the existing language constructs.

As described in Section 2.3, the internal syntax of Accelerate is first-order and uses typed De Bruijn indices. Consequentially, the lifted version of operations which are specialised and spliced into the output program, must also be written in this way. In this section, for simplicity, take we take the liberty of writing them in the familiar Accelerate frontend language instead.

We divide Accelerate's array combinators into these categories:

- Producers – This includes `generate`, `map`, `zipWith` and `backpermute`. All of these can be implemented in terms of `generate` with indexing.
- Folds – This includes `fold` and `fold1`.
- Scans – This includes `scanl`, `scanr`, `scanl'`, `scanr'`.
- Stencil – Accelerate has built-in support for stencil operations. We don't support these with sequence computations, leaving it for future work.

What follows is an explanation for the implementations of the *lifted irregular* versions of each category. For the regular versions, the combinators are already rank-polymorphic so little additional work is necessary. For lifted irregular, or segmented, operations we typically have to do more involved computations involving the segment descriptors.

Producers

As all of these combinators can be implemented in terms of `generate`, their segmented versions can be implemented in terms of a segmented `generate`.

```
generateSeg :: (Shape sh, Elt e)
            ⇒ Acc (Segments sh)
            → (Exp Int → Exp sh → Exp e)
            → Acc (Vector e)
generateSeg (unlift → (shapes, offsets, totalSize)) f
  = map (λ(unlift → (n,i)) → f n (toIndex (shapes ! n) i)) indices
  where
    ones     = fill (index1 totalSize) (-1,1)
    zeroes   = generate (shape offsets) (λix → lift (unindex1 ix, 0))
    heads    = scatter offsets ones zeroes
    indices  = scanl1 (⊕) heads
    (unlift → (a,b)) ⊕ (unlift → (a',b')) = a == a' ? (lift (a, b+b'), lift (a',0))
```

This definition may appear inefficient. This is true, and gives a further justification to a central point of this work: writing flattened arrays programs by hand is hard but recognising and taking advantage of regularity is also important.

We first create a vector of pairs of the total size of the segmented array (`ones`). The first and second component of each pair are a -1 and a 1 respectively. We use -1 here because this value needs to be outside the range of valid indices. We then create an indexed vector that is the same length as the offsets vector (`zeroes`). It contains the index of the element as the first component of each pair and 0 as the second. Using these two vectors, we then compute a vector of head flags (`heads`). This vector has a pair of (-1,1) at all positions except for those at the start of each segment where there is the number of the segment and a 0. We do this via `scatter`.

```
scatter :: Elt e
        ⇒ Acc (Vector Int)          -- destination indices to scatter into
        → Acc (Vector e)            -- default values
        → Acc (Vector e)            -- source values
        → Acc (Vector e)
```

By performing a scan over this flags vector, we are able to construct the `indices` for the segmented generate. It is a vector of pairs where the first component the segment that element occurs in and the second component is the index of that element within the segment. Using these indices we can then compute the result of the segmented generate.

Folds

Accelerate has two non-segmented versions of fold, `fold` and `fold1`. Both of these need flattened equivalents. Fortunately, this is simpler than flattening generate as Accelerate provides native `foldSeg` and `foldSeg1`. We still have to do some work to get our input in the right form for these operations, but we do not have to fully implement it or make scheduling decisions in the process of flattening.

Here is the type signature for `foldSeg` in Accelerate.

```
foldSeg :: (Shape sh, Elt a, Elt i, IsIntegral i)
```

```
          ⇒ (Exp a → Exp a → Exp a)
          → Exp a
          → Acc (Array (sh:.Int) a)
          → Acc (Vector i)
          → Acc (Array (sh:.Int) a)
```

First observe that it expects a vector of integral segment descriptors, but a potentially multidimensional array of values. This version of a segmented fold allows for multiple segmented reductions to be performed over the inner dimension of an array. This is more general than we require, so we specialise it to

```
foldSeg1D :: Elt a
          ⇒ (Exp a → Exp a → Exp a)
          → Exp a
          → Acc (Vector a)
          → Acc (Vector Int)
          → Acc (Vector a)
foldSeg1D = foldSeg
```

This gives us something close to the segmented folds described in Section 2.1. Using this, we can build the segmented fold we had in Section 4.4. First, we need to convert our higher dimensional segment descriptors into the integral segment descriptors.

```
segSizes :: Shape sh ⇒ Acc (Segments sh) → Acc (Vector Int)
segSizes (unlift → (shapes, _, _)) = map shapeSize shapes
```

Then, we apply `foldSeg1D` to the integral segments and the values vector. This gives us the values vector of the result, but we still need to compute the segment descriptors.

We do that with the following function.

```
foldSegments :: Shape sh ⇒ Acc (Segments (sh:.Int)) → Acc (Segments sh)
foldSegments (unlift → (shapes, _, _)) = segsFromShapes (map indexTail shapes)
```

Here, `segsFromShapes` computes segment descriptors from a vector of shapes.

```
segsFromShapes :: Shape sh ⇒ Acc (Vector sh) → Acc (Segments sh)
segsFromShapes shapes = let (offs, totalSize) = scanl' shapes
                        in (shapes, offs, totalSize)
```

The `scanl'` combinator computes the left-scan of its input, but rather than returning a vector of length $n + 1$, where n is the size of the input, it returns a vector of length n and a scalar value containing the that final element.

Scans

For the segmented version of the various scans, we define them in terms of a segmented generate and a non-segmented scan. The approach taken is similar for all the different variants. We described the simplest variant here `scanl1` and refer to the representation for the implementations of segmented `scanl`, `scanr`, `scanr1`, `scanl'`, `scanr'`, etc.

```
scanl1Seg :: (Exp e → Exp e → Exp e)
          → Acc (Segments (sh:.Int))
          → Acc (Vector e)
          → Acc (Vector e)
scanl1Seg f z segs vals
  = map fst (scanl1 f' (zip vals flags))
  where
    flags = generate_seg segs (λn (unlift → sh:.i) → i == 0)
    f' (e,_) (e',flag) = (flag ? (e', f e e'), false)
```

We first compute a flags vector of the same length as the values vector. It contains `true` at the start of each segment and `false` everywhere else. Then we create a version of the binary function `f` that only applies it when this flag is `false`. This technique of tagging the values with a flag indicating whether they are at the start of a segment is similar to previous approaches[4].

5.4 Fusion

While we have shown that it is possible to take our sequence programs with nesting and transform them so that the nesting is removed, doing so has introduced one particular performance problem. In the process of flattening, we introduce *numerous* extra intermediate arrays. The naïve compilation of programs that contain so many intermediate arrays quickly drags down performance as much of the programs run time is spent reading and writing values to and from memory. Fusion (or deforestation) attempts to remove this overhead by combining adjacent transformations on data structures in order to remove intermediate results. This has been studied extensively [14, 26, 28, 30, 40].

5.4.1 Array fusion

The core idea underlying the existing Accelerate array fusion system [30] is well known: simply represent an array by its size and a function mapping indices to their corresponding values. Fusion then becomes an automatic property of the data representation. This method has been used successfully to optimise purely functional array programs in Repa [24, 26], although the idea of representing arrays as functions is well known [12, 18, 21].

However, a straightforward implementation of this approach results in a loss of *sharing*, which was a problem in early versions of Repa [26]. For example, consider the following program:

```
let xs = use (Array … )
    ys = map f xs
in
zipWith g ys ys
```

Every access to an element `ys` will apply the (arbitrarily expensive) function `f` to the corresponding element in `xs`. It follows that these computations will be done *at least* twice, once for each argument in `g`, quite contrary to the programmer's intent. In the standard Accelerate fusion system, the solution to this problem is to not fuse terms, such as `ys`, whose results are used more than once.

However, this approach does not take into account what the term `ys` actually *is*; it simply sees that `ys` occurs twice in `zipWith` and so refuses to fuse it further. Consider the following example:

```
let xs = use (Array … )
    ys = (map f₁ xs, map f₂ xs)
in
zipWith g (fst ys) (snd ys)
```

Although the term `ys` still occurs twice in the `zipWith`, we can see that the individual components of the tuple each occur only once, and thus should still be subject to fusion.

This lack of fusion in the regular Accelerate optimisation system is particularly problematic for us, since we represent irregular sequences as a pair consisting of the segment descriptors together with a vector of the values. Thus, in the prior Accelerate fusion system, irregular sequences would never fuse, severely impacting performance. We extend the Accelerate fusion system in order to fuse the individual components of a tuple independently, which improves the performance of both our sequence computations as well as regular Accelerate programs.

To explain this, we will first describe more concretely how Accelerate's prior fusion system works, then cover the *fusion-through-tuples* implemented as part of Accelerate sequences. To start with, consider this program here

```
λxs → map (+1) (map (*2) xs)
```

we fuse it by a bottom up traversal. We must first *delay* `xs` into

```
(shape xs, (xs !))
```

Here we've created the mapping function with indexing and using the extent for the size. Proceeding up the AST, the `map (*2)` can now be delayed by taking the delayed form of `xs` and composing it with `*2`.

```
(shape xs, (λix → (xs ! ix)*2))
```

In general, in arrays are represented in delayed form:

```
map f (sh, g) = (sh, f . g)
```

We can then do the same with the `map (+1)`, giving us a final form of

```
λxs → (shape xs, (λix → (xs ! ix)*2))
```

This can be brought back into array form by turning the delayed array into a generate.

```
λxs → generate (shape xs) (λix → (xs ! ix)*2)
```

This process works for all producer combinators, however the conumer combinators (`fold`, `scanl`, etc.) cannot be delayed, so those must be left as *manifest*: still introducing intermediate arrays.

The other time intermediate arrays must be introduced, as already described, is when an array is accessed more than once. In Accelerate, this can only occur with let bindings. Looking at this example here,

```
let a = generate sh f
in map g a
```

we see that it would be beneficial, and obvious, to embed (inline) the binding `a` into the body to expose fusion opportunities there. In general though, that is not always desired, as highlighted above. So, when should we embed bindings like this? There are many possible measures that could be used, but we opt for a simple one. Accelerate inlines whenever a variable is only "used" once. By "used" here, we mean how often the variable occurs in the body of the let binding. We will extend this definition in the next section.

There is an additional problem as well. Consider this term.

```
let a = let b = scanl f z
        in map g b
in map h a
```

In this case, it is possible to fuse the term into

```
let b = scanl f z
in map (h . g) b
```

However, by doing this, we've had to float the binding for `b` outside of its previous scope. This extends the lifetime of `b` potentially affecting the programs memory usage. In general, Accelerate will perform this let floating, but only provided it gives additional fusion opportunities.

5.4.2 Fusion through tuples

Going back to our example.

```
let xs = use (Array ... )
    ys = (map f₁ xs, map f₂ xs)
in
zipWith g (fst ys) (snd ys)
```

Under the previous system, this wouldn't fuse, despite the fact that even though the tuple was "used" twice each component of the tuple is only accessed once.

By extending the concept of what it means for a binding to be used, we can enable terms like this to be fused. We simply need to keep a separate occurrence count for each component of a tuple.

In the above example, that would let us see that even though `t` occurs twice in the body, each component of `t` is only "used" once. Hence the binding can be embedded into the body using a specialised form of embedding that immediately simplifies when it encounters tuple projection. This results in

```
zipWith h (generate sh f) (generate sh' g)
```

which can be fused using the rules described above. If we were to embed via conventional inlining, just replacing `ys` with its definition at all use sites, we would end up with

```
zipWith h (fst (generate sh f, generate sh' g)) (snd (generate sh f, generate sh' g))
```

which, while still fusing after further simplification, is much larger than the original term. In general, simple inlining can cause an explosion in the size of the term, slowing down subsequent optimisation.

Let's now look at a more complicated example:

```
let t = (generate sh f, scanl g z arr, map h arr)
in ( zipWith h (prj 1 t) (prj 3 t)
   , zipWith k (prj 3 t) (prj 2 t) )
```

This brings up the question of what we should do when some components of a tuple are delayable (the `generate`) and are only used once, but others are either not delayable (the `scanl`) or occur multiple times (the `map`)? In this case we inline the first component of `t` but leave the second and 3rd components bound.

```
let t' = (scanl g 0 arr, map h arr)
in ( zipWith h (generate sh f) (prj 2 t')
   , zipWith k (prj 2 t') (prj 1 t) )
```

In general, when we encounter a let binding of a tuple like this, we split the tuple into two tuples, one containing all the components that are delayable and only occur once, the other containing all the components that should remain let-bound. The delayable tuple gets inlined using the specialised inline, the bound one remains let-bound.

Of course, like above, we also have to consider nested bindings:

```
let a = let b = scanr j 1 arr
        in (map f b, scanl g 0 b)
in zipWith h (fst a) (snd a)
```

This gets transformed into

```
let b = scanr j 1 arr in
let a = scanl g 0 b
in zipWith h (map f b) a
```

Here we are floating `b` out, possibly extending its lifetime. However, by doing this, we are able to fuse the first component of `a`. As above, we consider that the benefit of fusion outweighs the cost of let floating.

While initially these fusion rules relating to tuples may seem to only work in specific situations, it is worth noting that they in fact stop tuples being a barrier to fusion in all cases: simply due to the restricted nature of the Accelerate language. The two key properties that make this happen are that:

1. Tuple constructors can only occur in a few places. In the argument to tuple projections, in let bindings and in other tuple constructors. None of the combinators take tuples as arguments.

2. Unnecessary indirection of the form `let` v_0 = v_n `in` c[v_0] is eliminated during fusion as well.

We've already shown how tuple constructors in let bindings can be fused. The case where a tuple constructor is passed to tuple projection is trivial. The only remaining case is tuple constructors in other tuple constructors. For example,

```
let x = (a, (b, c, d))
in …
```

Supposing that `b` is delayable and is only "used" once by the body, but that `a`, `c` and `d` need to remain bound, we have to be careful. We capture the exact usage pattern of such tuples by maintaining usage information in the following representation:

```
data Uses a where
  UsesArray :: Int                            -- How many times the contents of the array is accessed
            → Int                             -- How many times the shape of the array is used
            → Uses (Array sh e)
  UsesTuple :: (Uses t_0, … , Uses t_n)
            → Uses (t_0,..,t_n)
```

As a result, we capture how every component or sub-component of a tuple is accessed.

When we inline with nested tuples, we similarly split each tuple into two smaller tuples. In the above example, we get

```
let x = (a, (c, d))
in …
```

with `b` now fused into the body.

5.5 Scheduling

While we have described how sequences are implemented in the language and compiler, we are still left with the problem of deciding how many elements of a sequence we should compute at any one time. This problem was not the focus of this work, but is still necessary to enable efficient execution. One approach, as described in Madsen et al. [29], is to analyse whole sequence computations and chose a static number based on an analysis of their parallel degree. While this works for regular computations, in the presence of irregularity, statically determining the parallel degree is not always feasible. In addition, where the size of individual elements may vary greatly, there is in general no good fixed static size. Hence, we use a dynamic scheduling approach, constantly adjusting the number of elements of the sequence to execute at once. For example, consider the following sequence computation:

```
consume (elements (mapSeq f (mapSeq g xs)))
```

Executing a step of the sequence computation consists of (1) computing some chunk of the input `xs`; (2) applying the lifted version of `g` to the chunk; (3) applying the lifted version of `f` to that result; and (4) store the result. Now, suppose that $\mathtt{g}^{\uparrow}$ achieves best performance when processing N elements at a time, but $\mathtt{f}^{\uparrow}$ prefers a size of $2N$ for best performance. We could conceivably compute two N-sized chunks with $\mathtt{g}^{\uparrow}$, then combine these into a $2N$-sized chunk for $\mathtt{f}^{\uparrow}$. However, even though the size of the chunk may be known, the actual size of the data in each chunk (of an irregular computation) is not. For this reason, any sort of multi-rate scheduling requires either considerable copying of intermediates or unbounded buffers. We avoid this issue by choosing to only adjust the chunk size after each complete step of the sequence computation.

We have two competing considerations for determining how many elements we should process in a step of a sequence computation: (1) we want to maximise processor utilisation, to ensure that all processing elements are busy; and (2) minimise the amount of time taken for each element of the sequence, which also acts as an approximate measure to minimising overall system resource requirements, such as memory usage. Sequence computations initially execute a single element of the sequence, then subsequently select a chunk size for the next step based on the following strategy:

- If the overall processor utilisation (time spent executing sequence computations compared to the elapsed wall time) is below a target threshold (80%), increase the chunk size. This is particularly important when the computation is bootstrapping from small chunk sizes and the elements of the sequence are small.

- Once the processor is sufficiently utilised, we continue to increase the chunk size only if it decreases the average time per element. If the time per element instead increases, decrease the chunk size. Otherwise, maintain the chunk size.

  In particular, the ability to decrease the chunk size is necessary to deal sequences containing elements that grow in size. Additionally, this acts as a proxy to minimise usage of other system resource, such as memory.

  If the processor is sufficiently utilised; i.e. above the minimum threshold, we continue to increase the chunk size only if it yields a decrease in the amount of time, on average, to process each element in the chunk. If the time taken does not decrease, we decrease the chunk size. This last part is necessary for sequences containing elements that grow in size.

As practical considerations, our implementation uses weighted moving averages of sampled variables, and increases the chunk size at twice the rate that we decrease it, which reduces warm-up time and biases towards ensuring the processors remains saturated.

Chapter 6

Enhancements

This chapter describes two additional extensions to the Accelerate language and runtime system which significantly improve the practical value of the Accelerate framework. They are:

- A foreign function interface (FFI) supporting both importing foreign functions into Accelerate, and exporting Accelerate programs for access via conventional CUDA C/C++ programs.
- A GPU aware garbage collector. This is necessary for cases where the amount of data needed by the program exceeds the available GPU memory, but, whether by streaming or otherwise, the data is not needed all at once.

6.1 Foreign Function Interface

Accelerate is an expressive high-level language framework that allows programmers to write programs for massively parallel GPU and CPU architectures, without requiring the expert knowledge needed to achieve good performance. This is particularly true for programming GPU architectures, as that typically requires specific knowledge of the architecture, as highlight in Chapter 2. However, there are existing highly optimised libraries, for example, for high performance linear algebra and fast Fourier transforms. For Accelerate to be practically useful, we need to provide a means to use those libraries. Moreover, access to native code also provides a developer the opportunity to drop down to low level C or CUDA C in those parts of an application where the code generated by Accelerate is not sufficiently efficient. We achieve access to CUDA libraries and native CUDA components with the *Accelerate Foreign Function Interface* (or FFI).

The Accelerate FFI works in both directions: (1) it enables calling native CUDA C code from embedded Accelerate computations and (2) it facilitates calling Accelerate computations from non-Haskell code. Overall, a developer can implement an application in a mixture of

Accelerate and other languages in a manner that the source code is portable across multiple Accelerate backends.

Given that Accelerate is embedded in Haskell, it might seem that Haskell's standard FFI should be sufficient to enable interoperability with foreign code. Unfortunately, this is not the case. With Haskell's standard FFI, we can call C functions from Haskell host code. However, we want to call functions from within embedded Accelerate code and, in the case of CUDA, pass data structures located in GPU memory directly to native CUDA code and vice versa. The latter is crucial, as transferring data from CPU memory to GPU memory and back is very expensive.

6.1.1 Importing foreign functions

Calling foreign code in an embedded Accelerate computation requires two steps: (1) the foreign function must be made accessible to the host Haskell program and (2) the foreign function must be lifted into an Accelerate computation to be available to embedded code. For the first step, we use the standard Haskell FFI. The second step requires an extension to Accelerate. We will focus on interfacing with CUDA, as it also requires us to consider data transfer. Interfacing with CPU-based libraries is much the same, but somewhat simpler as they work in the same memory space.

As a concrete example, let us use the vector dot product of the highly optimised *CUDA Basic Linear Algebra Subprograms (CUBLAS)* library [32]. This CUBLAS function is called `cublasSDot()`; it computes the vector dot product of two arrays of 32-bit floating point values. To access it from Haskell, we use this Haskell FFI import declaration:

```
foreign import ccall "cublas_v2.h␣cublasSdot_v2" cublasSdot
  :: Handle
   → Int                           -- Number of array elements
   → DevicePtr Float → Int         -- The two input arrays, and...
   → DevicePtr Float → Int         -- ...element stride
   → DevicePtr Float               -- Result array
   → IO ()
```

The `Handle` argument is required by the foreign library and created on initialisation. The `DevicePtr` arguments are pointers into GPU memory. As mentioned before, one of the primary aims of the Accelerate FFI is to ensure that we do not unnecessarily transfer data between GPU and CPU memory.

To manage device pointers, the Accelerate FFI provides a GPU memory allocation function `allocateArray` and a function `withDevicePtr` to perform a computation that depends on a device pointer of an Accelerate array. We can use these functions to invoke `cublasSdot` with GPU-side data:

```
dotp_cublas :: Handle
            → (Vector Float, Vector Float)
```

```
           → LLVM PTX (Scalar Float)
dotp_cublas handle (xs, ys) = do
  let n  = arraySize (arrayShape xs)   -- number of input elements
  result ← allocateArray Z             -- allocate a new Scalar array
  withDevicePtr xs          $ λxptr →  -- get device memory pointers
    withDevicePtr ys        $ λyptr →
      withDevicePtr result $ λrptr →
        liftIO $ cublasSdot handle n xptr 1 yptr 1 rptr
  return result
```

The `LLVM PTX` monad is simply the `IO` monad enriched with some information used by the PTX backend to manage devices, memory, and caches.

6.1.2 Executing foreign functions with Accelerate

The function `dotp_cublas` invokes native CUDA code in such a manner that it directly uses arrays in GPU memory. This leaves us with two challenges: (1) we need to enable calling functions, such as `dotp_cublas`, in embedded code and (2) we need to account for Accelerate supporting multiple backends, while Accelerate programs should be portable across backends.

We address this challenge by extending the AST with a new node type `Aforeign` representing foreign calls. One instance of an `Aforeign` node encodes the code for one backend, but it also contains a fallback implementation in case a different backend is being used. The AST data constructor is defined as follows:

```
Aforeign    :: (Arrays as, Arrays bs, Foreign asm)
            ⇒ asm                  (as → bs)    -- The foreign function for a given backend
            → PreAfun      acc     (as → bs)    -- Fallback implementation(s)
            → acc              aenv as          -- Arguments to the function
            → PreOpenAcc   acc aenv bs
```

When the tree walk during code execution encounters an `Aforeign` AST node, it dynamically checks whether it can execute the foreign function. If it can't, it instead executes the fallback implementation. A fallback implementation might be another `Aforeign` node with native code for a different backend (e.g., for CPU instead of PTX), or it can simply be a vanilla Accelerate implementation of the same functionality that is provided by the foreign code. With a cascade of `Aforeign` nodes, we can provide an optimised native implementation of a function for a range of backends and still maintain a vanilla Accelerate version of the same functionality for execution in the Accelerate interpreter.

The dynamic check for the suitability of a foreign function is facilitated by the class constraint `Foreign f` in the context of `Aforeign`. The class `Foreign` is a subclass of `Typeable` with instances for data types that represent foreign functions for specific backends. For the CUDA backend, we have the following:

```
class Typeable2 f ⇒ Foreign f where …
instance Foreign ForeignAcc where …
data ForeignAcc f where
```

```
ForeignAcc :: String
           → (Stream → a → LLVM PTX b)
           → ForeignAcc (a → b)
```

`ForeignAcc` wraps calls to foreign CUDA code executed in the `LLVM PTX` monad. When the CUDA backend encounters an AST node `Aforeign foreignFun alt arg`, it attempts to `cast`[1] the value of `foreignFun` to type `ForeignAcc f`. If that `cast` succeeds, it can unwrap the `CUDAForeignAcc` and invoke the function it contains. Otherwise, it needs to execute the alternative implementation `alt`.

Finally, we can define an embedded vector dot product that uses CUBLAS when possible and, otherwise, falls back to the version defined in Section 2.3:

```
dotp' :: Acc (Vector Float) → Acc (Vector Float)
      → Acc (Scalar Float)
dotp' xs ys = Aforeign (CUDAForeignAcc (dotp_cublas handle))
                       (uncurry dotp)
                       (lift (xs, ys))
```

Foreign calls are not curried; hence, they only have got one argument, which is an instance of the class `Arrays` of tuples of Accelerate arrays.

6.1.3 Exporting functions

Accelerate simplifies writing high performance code as it lessens the need to understand most low-level details of, higher level programming. Hence, we would like to use Accelerate from other languages. As with importing foreign code into Accelerate, the foreign export functionality of the standard Haskell FFI is not sufficient for efficiently using Accelerate from languages, such as C. In the following, we describe how the Accelerate FFI supports exporting Accelerate code as standard C calls.

Exporting Accelerate programs

To export Accelerate functions as C functions, we make use of Template Haskell [39]. For example, we might export our Accelerate dot product:

```
dotp :: Acc (Vector Float, Vector Float) → Acc (Scalar Float)
dotp = uncurry $ λxs ys → fold (+) 0 (zipWith (*) xs ys)

exportAfun 'dotp "dotp_compile"
```

The function `exportAfun` is defined in Template Haskell and takes the name of an Accelerate function, here `dotp`, as an argument. It generates the necessary export declarations by inspecting the properties of the name it has been passed, such as its type.

Compiling a module that exports Accelerate computations in this way (say, `M.hs`) gener-

[1]See Haskell's `Data.Typeable` library for details on `cast`.

ates the additional file `M_stub.h` containing the C prototype for the foreign exported function. For the dot product example, this header contains:

```
#include "HsFFI.h"
extern AccProgram dotp_compile(AccContext a1);
```

A C program needs to include this header to call the Accelerate dot product.

Running embedded Accelerate programs

One of the functions to execute an Accelerate computation in Haskell is:

```
run1In :: (Arrays as, Arrays bs)
       ⇒ Context → (Acc as → Acc bs) → as → bs
```

This function comprises two phases: (1) program optimisation and instantiation of skeleton templates of its second argument and (2) execution of the compiled code in a given CUDA context (first argument). The implementation of `run1In` is structured such that, partially applying it to only its first and second argument, yields a new function of type `as → bs`, where Phase (1) has been executed already — in other words, it precompiles the Accelerate code. Repeated application of this function of type `as → bs` executes the CUDA code without any of the overheads associated with just-in-time compilation.

The Accelerate export API retains the ability to precompile Accelerate code. The C function provided by `exportAfun` compiles the Accelerate code, returning a reference to the compiled code. Then, in a second step, `runProgram` marshals input arrays, executes the compiled program, and marshals output arrays:

```
OutputArray   out;
InputArray    in[2]   = { … };
AccProgram    dotp    = dotp_compile( context );

runProgram( dotp, in, &out );
```

The function `dotp_compile` was generated by `exportAfun 'dotp "dotp_compile"`.

Marshalling input and output arrays

Accelerate uses a non-parametric representation of multi-dimensional arrays: an array of tuples is represented as a tuple of arrays. The type `InputArray` follows this convention. It is a C struct comprising an array of integers indicating the extent of the array in each dimension together with an array of pointers to each underlying GPU array of primitive data.

```
typedef struct { int* shape; void** adata; } InputArray;
```

`OutputArray` includes an extra field, a stable pointer, that maintains a reference to the associated Haskell-side `Array`. This keeps the array from being garbage collected until the `OutputArray` is explicitly released with `freeOutput`.

```
typedef struct { int* shape; void** adata;
                 HsStablePtr stable_ptr; } OutputArray;
```

6.2 GPU aware garbage collection

One major design consideration of high-level languages is how to manage memory. Forcing the programmer to manually allocate and free memory hardly satisfies the property of being high-level, but any alternative involves some compromise. The most common solution to this problem is automatic garbage collection where the runtime system associated with a language will periodically scan the heap and determine what allocated memory is reachable and free any allocations that are not. There are many different methods for doing this, but what we propose here is agnostic to which method is used.

The Glasgow Haskell Compile (GHC) supports garbage collection already. As Accelerate is embeddded in Haskell, we can introduce garbage collection to it by hooking into GHC's garbage collector. To do this however, we have to address the question of when data should be transferred both to and from the GPU.

6.2.1 When to transfer data?

As we have already highlighted, the limited working memory provided by GPUs makes it difficult to write programs that have inputs of sizes exceeding this memory. We have shown how array sequences can remove this limitation, for certain classes of programs. However, even with sequences, there are still cases where this is a problem. To see why, suppose we have program pipeline with 2 stages. The first stage uses sequences to compute values intended for the second stage. The second stage, however, is not able to process its input as a stream but instead requires all of the results before it can do anything. Such a program might have this structure:

```
secondStage . streamOut . firstStage
```

For programs like this, the Accelerate runtime system needs to decide when to copy arrays to the *device* memory on the GPU, when to copy them back to *host* memory and when to free the device memory allocated for them. For our example, the first stage could do as follows for each chunk of the sequence:

1. Compute the chunk from `firstStage`.
2. Copy it back to main memory.
3. Free it from GPU memory.

This has the advantage of minimizing space usage on device memory, but it is not always the most optimal thing to do. If `secondStage` is purely CPU based processing, then we have not lost anything by transferring the chunk as soon as we have computed it. If, however, `secondStage` does its own GPU processing, with some or all of its input, then we might not want to transfer immediately. If it needs that data to once again be in device memory, then it makes little sense to transfer to host memory and back again.

6.2.2 The previous solution

Prior to our work, Accelerate already had a solution to this problem. The transfer of arrays from device memory were, and still are, delayed till an attempt is made to access the values in the array from the CPU. For freeing the array, Accelerate hooks into the Garbage Collector (GC) of the Glasgow Haskell Compiler (GHC). It attaches a finaliser that frees the array in GPU memory when the host-side version of the array is collected.

This solves the problem of premature data transfer. Arrays won't be transferred back to host memory till they're needed there and won't be left in device memory once the host-side array is no longer needed. However, it introduces another problem as device memory usage will grow with host memory usage. Results from `firstStage` aren't from device memory till they are all computed. Naturally, this leads to memory exhaustion and takes away one of the advantages of streaming (at least for examples like this one.)

6.2.3 Array eviction

As part of this work, we implemented a more advanced form of GPU aware garbage collection. It works in much the same way as the previous solution, but allows for arrays to be copied (*evicted*) back from device memory when memory is exhausted. Specifically, allocation of arrays works as follows:

1. Attempt to allocate space on the device. If that is successful, use it for the device-side representation of the array.
2. If it fails due to an out-of-memory-error, trigger a GHC garbage collection, try to yield to the GC thread, and then try to allocate again.
3. If that fails, find the *least recently used* array in device memory and, if necessary, copy it back to host memory before freeing it. After that, try to allocate again.
4. If that fails, repeat the above step.

For (1), we use CUDAs version of `malloc()` which will fail with an error if is unable to allocate on device memory. With (2) we rely on `performGC` from `System.Mem`, which causes a GC to occur in a separate thread. We can't yield to this thread exclusively, but, in practice, calling `yield` from `Control.Concurrent` typically achieves that.

To do (3) requires us to keep track of when arrays were used. To do this we attach a timestamp to each record of a device-side array. In Section 6.1.1 we introduced `withDevicePtr`;

internally Accelerate uses that function whenever it needs to access an array on the device. We extended that function to update the timestamp associated with the array.

6.2.4 Discussion

This new method of memory management is not only advantageous to programming with sequences, but also more generally. Programs that do not use sequences, but do require inputs larger than available memory also benefit from automatic eviction. That said, an LRU based eviction is not always the most optimal. One example is if a program works by doing multiple passes over the same input, then evicting the least recently used array may not always be the best choice, as it may represent future input. It may be better in that case to evict the result, assuming it is not needed for the next pass of the algorithm. We have found, however, that in most cases the LRU strategy is the best, but there is room for further exploration in this area.

Chapter 7

Evaluation

The objective of our work is to improve the expressive power of a flat data-parallel array language with the introduction of irregular structures and a limited form of nested parallelism. However, to be of practical relevance, this extra expressiveness may only impose a reasonable cost on performance. We demonstrate the performance of this work through a series of benchmarks, which are summarised in Table 7.1.

Benchmarks were conducted using a single Tesla K40c GPU (compute capability 3.5, 15 multiprocessors = 2880 cores at 750MHz, 11GB RAM) backed by two 12-core Xeon E5-2670 CPUs (64-bit, 2.3GHz, 32GB RAM, hyperthreading is enabled) running GNU/Linux (Ubuntu 14.04 LTS). We used GHC-8.0.2, LLVM-3.9.1, and NVCC-8.0.44. Haskell programs are run with RTS options to set thread affinity and match the allocation size to the processor cache size.[1] We use `numactl` to pin threads to the physical CPU cores. Execution times are measured using criterion[2] via linear regression.

Name	Input Size	Competitor		Accelerate		Accelerate +Sequences	
SMVM (Queen_4147)	330M	62.0	(MKL)	78.1	(126%)	110.7	(179%)
Audio processor	(continuous)	2.1	(C)	0.36	(18%)	0.11	(5.2%)
MD5 Hash	14M	49.9	(Hashcat)	92.0	(184%)	285.5	(572%)
PageRank	130M	5840	(Repa)	2220	(38%)	4240	(73%)

Table 7.1: Benchmark summary. Execution times in milliseconds.

[1] `+RTS -qa -A30M -N`*n*

[2] `http://hackage.haskell.org/package/criterion`

	Non-zeros	Competitor				Accelerate				Accelerate+Sequences			
Name	(nnz/row)	N=1	N=12	N=24	GPU	N=1	N=12	N=24	GPU	N=1	N=12	N=24	GPU
pdb1HYS	4.3M (119)	2.09	28.36	47.90	21.61	1.81	13.71	16.36	7.96	0.86	1.50	1.15	0.07
consph	6.0M (72)	2.11	20.72	26.38	15.44	1.82	11.60	9.12	7.79	1.18	1.61	1.26	0.09
cant	4.0M (64)	2.18	30.77	52.21	13.99	2.02	14.52	15.52	7.12	0.82	1.32	0.97	0.06
pwtk	11.6M (53)	2.07	5.41	17.21	13.20	1.97	7.91	4.90	8.36	1.01	2.19	1.74	0.16
rma10	2.4M (50)	2.70	20.48	38.60	13.31	2.09	8.94	10.33	5.68	0.77	0.88	0.65	0.04
shipsec1	7.8M (55)	2.06	13.72	17.66	12.23	2.02	9.61	9.00	7.91	0.90	1.60	1.43	0.11
rail4284	11.3M (10)	1.06	3.68	5.10	7.08	0.83	2.14	3.31	4.58	0.59	1.09	1.18	0.21
TSOPF_FS_b300_c2	8.8M (154)	2.04	5.47	6.57	8.47	1.81	4.32	4.33	5.27	1.21	1.86	1.66	0.14
FullChip	26.6M (9)	0.97	1.96	3.36	0.10	1.19	2.73	2.89	0.32	0.53	1.29	1.28	0.17
dielFilterV2real	48.5M (42)	1.40	4.31	8.06	14.50	1.76	4.21	7.59	7.89	1.34	3.10	3.22	0.54
Flan_1565	117.4M (75)	1.49	4.86	8.40	22.43	1.44	12.77	11.70	9.36	1.35	5.06	5.43	1.20
Queen_4147	329.5M (80)	1.78	9.07	9.71	23.00	1.30	8.09	6.83	9.02	1.24	6.68	5.28	2.36
nlpkkt240	774.5M (28)	1.51	6.90	6.27	16.88	1.39	6.49	8.15	7.02	0.85	4.10	4.37	3.19
HV15R	283.0M (140)	1.50	10.66	5.16	22.45	1.40	13.77	25.08	9.75	1.35	6.24	6.43	2.30

Table 7.2: Overview of sparse matrices tested and results of the benchmark. Measurements in GFLOPS/s (higher is better). Columns labelled N=*n* are CPU implementations using *n* threads. Competitor implementations are Intel MKL on the CPU and NVIDIA cuSPARSE on the GPU.

7.1 Sparse-Matrix Vector Multiplication (SMVM)

SMVM multiplies a sparse general matrix in compressed row format (CSR) [11] with a dense vector (see Section 3.2.1). Table 7.2 compares Accelerate to the Intel Math Kernel Library[3] (MKL v11.3.2, `mkl_cspblas_dcsrgemv`) on the CPU, and NVIDIA cuSPARSE[4] (v8.0, `cusparseDcsrmv`) on the GPU. Test matrices are derived from a variety of application domains [16] and are available online.[5] Matrices use double precision values and 32-bit integer indices. GPU implementations do not include host/device data transfer time.

In a balanced machine, SMVM should be limited by memory bandwidth. Accelerate is at a disadvantage in this regard, since MKL and cuSPARSE require some pre-processing to construct the segment descriptor, which is not included in their timing results. Our Sequences implementation has some additional bookkeeping work over standard Accelerate in order to track sequence chunks, further reducing overall throughput.

Figure 7.1 shows the strong scaling when computing the sparse-matrix multiply of the Queen_4147 dataset on the CPU. The work in this paper achieves 56% the performance of the highly-tuned reference implementation (or a 30% slowdown compared to the base Accelerate implementation).

[3]`https://software.intel.com/en-us/intel-mkl`

[4]`https://developer.nvidia.com/cusparse`

[5]`http://www.cise.ufl.edu/research/sparse/matrices/list_by_id.html`

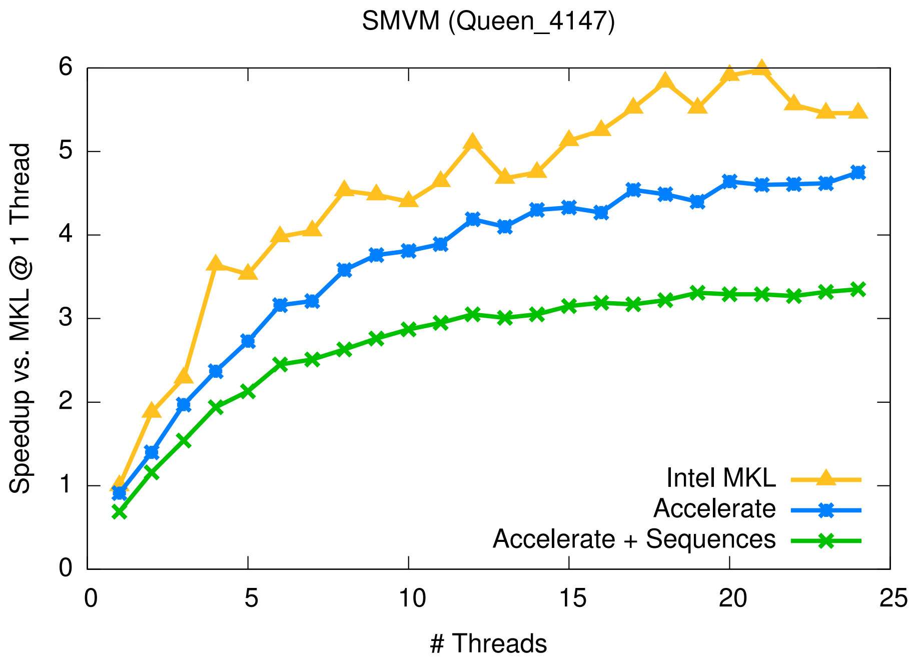

Figure 7.1: SMVM of Queen_4147 dataset

7.2 Audio processor

The audio processing benchmark tests the algorithm described in Section 3.2.2, computing the `zc_stream` part of the algorithm which processes the input audio data along a sliding window. Since standard Accelerate is limited to flat data parallelism only, it can only take advantage of a single source of parallelism, and applies the `processWindow` operation to a single element of the stream at a time. However, with sequences, we can also take advantage of another source of parallelism and process multiple windowed elements at a time. This is particularly important for the GPU, since we must have enough work to fully saturate all the cores of the device, as well as to amortize the overhead of streaming data between the CPU and GPU. On our test machine the runtime chooses to process 512 elements at a time on the GPU, resulting in a speedup of 3.3× over regular Accelerate.

Figure 7.2 shows the strong scaling performance compared to the implementation in regular Accelerate[6] on the CPU. As seen in the other benchmark programs, our implementation currently has some overhead compared to regular Accelerate (which has had more time to mature). However, since we can also take advantage of the extra parallelism of processing multiple windowed elements in parallel, we are able to make up the difference in this benchmark.

[6]Unfortunately we do not have a parallel reference implementation of the algorithm to compare to.

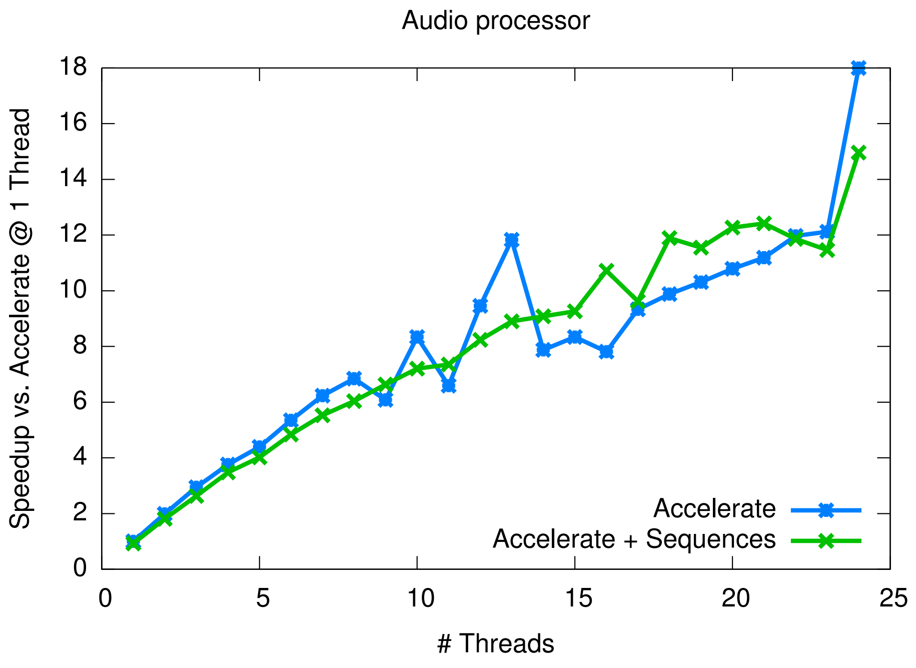

Figure 7.2: Audio processor

7.3 MD5 Hash

The MD5 message-digest algorithm [37] is a cryptographic hash function producing a 128-bit hash value that can be used for cryptographic and data integrity applications. The MD5 benchmark attempts to find the plain text of an unknown hash with a database of known plaintexts. We compare to Hashcat,[7] the self-proclaimed world's fastest CPU-based password recovery tool. Results from Hashcat are as reported by its inbuilt benchmark mode.

Our sequences based implementation achieves a maximum throughput of 50 million words per second, compared to Hashcat at 287Mword/sec and standard Accelerate at 155Mword/sec. One key difference between our version using sequences and the implementation in standard Accelerate is that we stream in and process small chunks of the input dictionary at a time, rather than loading the entire database into memory at once. However, our runtime system currently does not overlap computation with pre-loading the next chunk of input into memory, which would help close this performance gap. The strong scaling graph is shown in Figure 7.3.

[7] `https://hashcat.net/hashcat/`

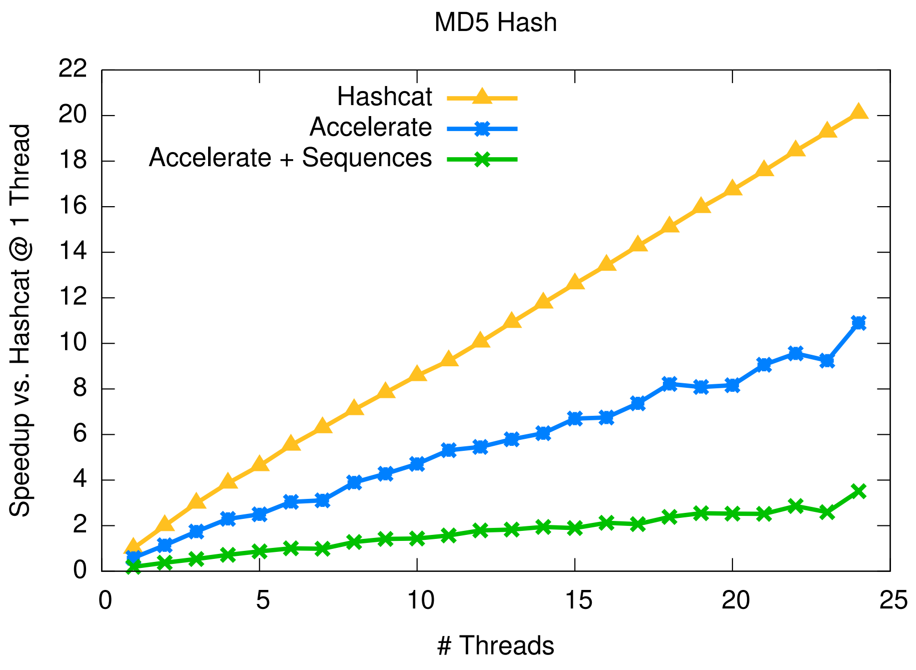

Figure 7.3: MD5 hash recovery

7.4 PageRank

PageRank [34] is a link analysis algorithm which estimates the relative importance of each element of a linked set of documents. As input we use a link graph of Wikipedia (English) consisting of 5.7 million pages and 130 million links. Figure 7.4 compares the performance against an implementation written in Repa [24, 26], a data-parallel array programming language similar to Accelerate. Both Accelerate and Repa represent the link graph as index pairs `(Int,Int)`. While this representation is not as space efficient as an adjacency list, it is the one typically chosen for this algorithm as it is more suitable for parallelism.

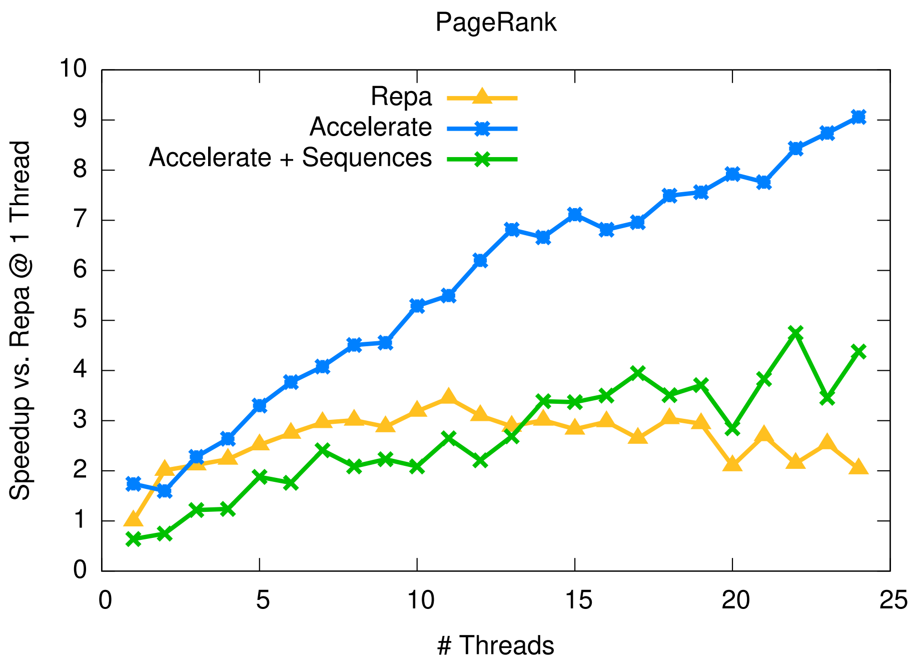

Figure 7.4: PageRank analysis of Wikipedia (English)

Chapter 8

Conclusion

In this work we argue that adding even one level of nesting to array programs offers considerable benefits to modularity. Moreover, if that form of nesting is in sequences we get the additional benefit of reduced memory constraints.

In Chapter 3 we present the irregular array sequence as an abstract structure and how it adds an additional layer of expressiveness to the array language Accelerate. Even with a relatively small set of operations on sequences, we can express programs that were awkward and non-modular to write before. As well as programs that would otherwise require specific knowledge of the hardware to write without exhausting available GPU memory.

In Chapter 4 we describe a novel extension to Blelloch's flattening transform, that we have used to enable irregular arrays sequences, but which can be applied more generally to all nested array structures. This extension is able to identify computations which are regular and generate code that is optimal for that regularity while generating conventional flattened array code for irregular computations.

In Chapter 6 we describe two further novel components of our work. We introduce the Accelerate Foreign Function Interface (FFI), a new way of allowing for an embedded language to support foreign functions. We also introduce a method of garbage collection that allows for GPU memory to be treated like a cache for arrays that are used in GPU computations.

Finally, Chapter 7 evaluates our work with a series of benchmarks.

8.1 Contribution

The work contains 2 major contributions and 2 minor contributions. The two major ones are regularity identifying flattening and irregular array sequences. They are closely related as the former is necessary to implement the latter with expected performance gains. However, it is much more widely applicable. Regularity identifying flattening can be applied to any context where efficient nested arrays are needed.

The minor contributions further extend what we can practically express in an array language, by dynamically reducing memory usage through GPU-aware garbage collection and by giving access to highly-optimised 3rd party libraries through a foreign function interface.

8.1.1 Regularity identifying flattening

While program flattening via Blelloch's flattening transform is well understood, languages that implement it have either assumed everything is irregular [1, 5, 7, 15, 19], or, more recently, that everything is regular [22]. While the former is most general, it misses opportunities to take advantage of efficiency of regular scheduling. The latter, however, is unable to express irregular computations.

By having a flattening transform that is able to identify regular (sub)programs and generate flat array code for them, while still the flat array code suitable for irregular structures otherwise, we get the best of both worlds. This enable us to support not only irregular nested programs and regular nested programs, but also programs that combine both.

8.1.2 Irregular array sequences

Given two, almost entirely independent, limitations on flat array programs, in this work we show that both can be overcome with the addition of one feature. Irregular sequences of arrays allow us to tk the loss of modularity in flat array programs, without having to tk the challenges of more deeply nested structures. Moreover, they gives us portability between GPUs and CPUs, in particular, their different memory limitations. The same program can be executed on both architectures and, even if the GPU does not have sufficient memory to perform the computation in-core, it will have tk performance on both.

8.1.3 GPU-aware garbage collection

A feature that is useful for both sequence programs and array programs alike is for the garbage collector to be aware of the GPU and its, typically, much smaller memory than the main memory of the system it is in. While the CPU is able to swap pages out to disk, prior to our work, there was no was no system for swapping memory out of the GPUs memory to host memory. By both hooking into GHC's garbage collector, and keeping track of what arrays are in GPU memory, we are able to dynamically manage the memory of the GPU such that the programmer no longer has to concern themselves with exhausting it. A high-level language should abstract away such details. With our work, a programmer only has to concern themselves with whether individual arrays will fit in GPU memory, not whatever the working memory of their program is, which is typically much harder.

8.1.4 A Foreign Function Interface for an embedded language

Foreign function interfaces have proven to be invaluable to the adoption of high-level languages[8]. Prior to our work, this was an area unexplored in the domain of embedded languages. By introducing an FFI to Accelerate, we made it possible to call out to already highly-optimised low-level libraries from within high-level programs. In addition, different backends can be supported by having different foreign functions for each backend.

8.2 Future work

What we describe in this dissertation could be extended in a number of different ways. Here, we present a few key research possibilities.

8.2.1 More sequence operations

Even though we can express realistic sequence programs with the operations we have, like with different array combinators there is room for more elaborate sequence combinators. In our work, we wanted to avoid the problem of unbounded buffers, but if such a restriction were relaxed (say by supporting data structures that allow for fast concatenation) a much wider range of operations is possible.

TODO: *Example?*

8.2.2 More nested structures

Given the regularity identifying flattening transform in Chapter 4 an obvious piece of future work is to support arbitrary nested arrays in Accelerate. While we have demonstrated that it is possible to do so in a way that takes advantage of regularity, there are still questions that arise. In particular, scheduling of deeply nested programs on GPUs has proven to be challenging[1]. Additionally, to really support nested structures, sum and recursive data types are needed. This can be achieved, in a limited fashion, with such types in the meta language, but if that is sufficient or whether Accelerate itself needs them is still an open question.

8.2.3 Multi-GPU parallelism

With sequences, we introduce a new level of parallelism. We've already shown how pipeline parallelism can be used to take advantage of both the CPU and GPU by executing two different stages at the same time, one on the GPU and one on the CPU. However, doing this on multiple GPUs is yet to be explored. It's already possible, with Accelerate's existing support for multiple GPUs, to manually schedule different computations on different GPUs, but to do so automatically is unexplored. Being able to define a sequence pipeline and have the different stages allocated across the available GPUs in such a way as to maximise efficiency (both in terms of work and in terms of data transfer) would be a powerful language feature.

Bibliography

[1] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP: International Conference on Functional Programming.* ACM, 2012.

[2] G. Blelloch and G. Narlikar. A practical comparison of *n*-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[3] G. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel delaunay algorithm. In *Proceedings ACM Symposium on Computational Geometry*, May 1996.

[4] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Nov. 1990.

[5] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[6] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *The 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 575–585. IEEE, 1988.

[7] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[8] M. M. T. Chakravarty. The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report, 2003.

[9] M. M. T. Chakravarty and G. Keller. More Types for Nested Data Parallel Programming. In *In Proceedings ICFP 2000: International Conference on Functional Programming*, pages 94–105. ACM Press, 2000.

[10] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP: Declarative Aspects of Multicore Programming*, pages 3–14. ACM, 2011.

[11] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proc. of Supercomputing*, pages 666–675. IEEE Computer Society Press, 1990.

[12] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects and Applications of Multicore Programming*. ACM, 2012.

[13] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. Technical report, NVIDIA, 2013.

[14] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*. ACM, 2007.

[15] J. P. Daniel Palmer and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE, 1995.

[16] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011. URL `http://www.cise.ufl.edu/research/sparse/matrices`.

[17] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[18] C. Elliott. Functional Images. In *The Fun of Programming*. Palgrave, 2003.

[19] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The manticore project. In *ML'07: Workshop on ML*, pages 15–24. ACM, 2007. doi: 10.1145/1292535.1292539.

[20] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[21] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *POPL '78: Principles of Programming Languages*, pages 1–8, 1978.

[22] T. Henriksen, N. G. 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*, pages 556–571. ACM, 2017.

[23] K. E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM. doi: 10.1145/1460833.1460872. URL `http://doi.acm.org/10.1145/1460833.1460872`.

[24] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP: International Conference on Functional Programming*, pages 261–272. ACM, 2010. doi: 10.1145/1863543.1863582.

[25] G. Keller, M. M. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones. Vectorisation avoidance. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[26] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*. ACM, 2012.

[27] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher-order vectorisation. In *ICFP'12: International Conference on Functional Programming*, pages 259–270. ACM, 2012. doi: 10.1145/2364527.2364564.

[28] B. Lippmeier, M. M. Chakravarty, G. Keller, and A. Robinson. Data flow fusion with series expressions in haskell. In *ACM SIGPLAN Notices*, volume 48, pages 93–104. ACM, 2013.

[29] F. M. Madsen, R. Clifton-Everest, M. M. T. Chakravarty, and G. Keller. Functional array streams. In *FHPC'15: Workshop on Functional High-Performance Computing*, pages 23–34. ACM, 2015.

[30] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*, pages 49–60, Sept. 2013.

[31] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212, New York, New York, USA, 2015. ACM Press.

[32] NVIDIA. CUBLAS Library, 2018.

[33] NVIDIA. CUDA C Programming Guide, 2018.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[35] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP'06: International Conference on Functional Programming*, pages 50–61, 2006.

[36] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science*, Oct. 2008.

[37] R. Rivest. The md5 message-digest algorithm. 1992.

[38] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Symposium on Graphics Hardware*, pages 97–106. Eurographics Association, 2007.

[39] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[40] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.