University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Proving the Correctness of Rewrite Rules in LIFT's Rewrite-Based System

Xueying Qin

February 17, 2020

Abstract

Rewrite rules in LIFT are used for generating optimised code but their correctness is not fully justified. In this project, mechanical proofs are developed in Agda for the verification of their correctness. Semantics of LIFT's data types and primitives are defined and their correctness are justified. Most of LIFT's rewrite rules are proven to be correct while some incorrect and inaccurate rules are fixed and clarified. Strategies of proving rewrite rules over array and multi-dimensional array operations are also discussed.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Xueying Qin Date: 30 March 2020

Acknowledgements

I would like to thank my supervisor Dr. Michel Steuwer, for his support and help in this project. Also, I would like to thank Uma Zalakain and Rongxiao Fu, who are PhD students at University of Glasgow, for sharing their experiences in developing verification programs in Agda, which are very helpful to this project.

Contents

1 Introduction

1.1 Motivation

The rewrite system of LIFT systematically transforms programs composed of high-level algorithmic patterns into low-level high performance OpenCL code with equivalent functionality. During this process, a set of rewrite rules are applied, each encoding an implementation of optimisation choice. Ensuring the correctness of these rules is important for ensuring the program's original functionality is not altered during the rewrite process.

Currently, the correctness of these rules is only proven on paper and only for a subset of the used rules. Although the correctness of some rules can be pointed out straightforwardly, for example, the map fusion rule provided by Steuwer (2015):

$$map\ f \circ map\ g \rightarrow map\ (f \circ g)$$

which fuses two consecutive `map` operations in function composition; the correctness of some rules is obscure, take the tiling rule (Hagedorn et al. 2018) as an example:

$$map\ f \circ slide\ size\ step \rightarrow join \circ map\ (\lambda\ tile.\ map\ f \circ (slide\ size\ step\ tile))\ slide\ u\ v$$

it is difficult to tell the left-hand side of the arrow has the same functionality as the right-hand side of the arrow from sheer observation.

Moreover, existing paper proofs do not use standard proof techniques such as proof by induction and proof by contradiction. For example, the map fusion rule is proven as (Steuwer 2015):

Let $xs = [x_1, ..., x_n]$.

$$\begin{aligned}
(map\ f \circ map\ g)\ xs &= map\ f\ (map\ g\ xs) \\
&= map\ f\ [g\ x_1, ..., g\ x_n] && \text{definition of map} \\
&= [f\ (g\ x_1), ..., f\ (g\ x_n)] && \text{definition of map} \\
&= [(f \circ g)\ x_1, ..., (f \circ g)\ x_n] && \text{definition of } \circ\text{, definition of map} \\
&= map\ (f \circ g)\ xs
\end{aligned}$$

Although conceptually this is a correct proof, it does not have formal proof structures or stress on what conditions should f, g and xs satisfy. In general, these paper proofs are not convincing enough and sometimes can be wrong.

Thus, in this project a set of mechanical proofs in Agda will be developed to show the correctness of the existing rules implemented in LIFT system and published in paper.

1.2 Aims

First of all, we would like to design concise and well-structured operational semantics for LIFT's patterns, which are introduced in chapter 3. Secondly, the validation of rewrite rules are produced by equality reasoning in Agda. Different kinds of rules and strategies of developing proofs are discussed (chapter 4). Base on the developed proofs, we further discuss if these rules are correct and accurate, and revise the problematic ones.

2 Background

2.1 LIFT

LIFT (Steuwer et al. 2015) is a high-level programming language which provides high performance and code portability. LIFT achieves this via a rewrite-based system to transform and optimise programs. Programmers express operations using a set of high-level functions, such as `map`, `reduce`, `split`, `join` etc. The are called patterns in LIFT. During the compilation, expressions containing these high-level patterns are rewritten into expressions with algorithmic optimisation strategies, for example, divide and conquer for parallel operations, and then high-performance hardware specific code is generated accordingly in the code generation phase. These strategies are defined as rewrite rules. The there different kinds of rewrite rules, such as rewrite rules for algorithmic optimisations and rules for changing the orders of patterns in expressions allowing algorithmic optimisations to be applied.

2.2 Curry-Howard Correspondence

Since we would like to use a set of computer programs to verify mathematical proofs, there needs to be an equivalence relation between these two kinds of formalism. The one serves as the basis of this project is Curry-Howard Correspondence, which was firstly discovered by Curry (1934): a theory of functions can be related to a theory of implication; and later revisited by Howard (1980): there exists correspondence between natural deduction and simply-typed lambda calculus. This correspondence was then synthesised by Wadler (2015): propositions as types; proofs as programs; simplification of proofs as evaluation of programs.

Following this paradigm, we are able to encode equality relations for stating rewrite rules as types and develop programs which serve as proofs to justify these proposed equality relations.

2.3 Agda

Agda (Ulf et al.) is a dependently typed programming language, where the definition of each of its types can depend on a value. It can be used as a proof assistant based on the proposition-as-type paradigm. Data types are defined inductively in Agda with dependently typed pattern matching, i.e., checking if a sequence of token matches a given pattern, which eliminates type errors of the execution of a function by restricting the type checking on its input and output data types. This feature is demonstrated in the following example.

Firstly, we define two data types, one is the natural number type and the other is an indexed collection, where its type depends on its size:

```
-- Define the natural number type
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
-- Define an indexed collection
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

It can be pointed out that there exists correspondence in the patterns of these two definitions: 1) in the base case of Vec, the pattern of its size matches the pattern of the base case zero in the definition of ℕ; 2) the constructor _::_ is defined with the pattern matching on size using suc in the definition of ℕ.

Now we would like to define a function head, which returns the first element of an indexed collection. Note that this function does not take an empty collection as input, as it is impossible to get the first element from it since it does not contain any. To emphasise this restriction in the definition, head is provided as:

```
head : {A : Set} → {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

In this definition, Agda normalise the input argument, which is an indexed collection with type (Vec A (suc n)), into (x :: xs), since only the constructor _::_ matches the given pattern but not []. Hence the restriction is reinforced in this definition.

More detailed information about defining data types and functions, as well as developing equality reasoning will be introduced in chapter 3 and chapter 4, alongside developing semantics of primitives and verification of rewrite rules.

3 Semantics of LIFT in Agda

Introduced by Steuwer et al. (2015); Atkey et al. (2017), LIFT patterns have been formalised into a set of mathematical expressions, for instance, the function `map`, which is a LIFT primitive, is defined as:

$$\textbf{map}: \quad \{n : \mathsf{nat}\} \to \{s\ t : \mathsf{data}\} \to (s \to t) \to n_\bullet s \to n_\bullet t \tag{3.1}$$

where `nat` is the natural number type and `data` can represent any data types, $(s \to t)$ represents a function which takes in an arguments with data type s and returns a value with data type t. $n_\bullet s$ is an array which contains elements with data type s and has size n. In addition, braces and parentheses are used for indicating implicit arguments and explicit arguments.

We define the operational semantics of LIFT patterns as functions in Agda base on those mathematical expressions, which describes the meaning of data types and the computation steps of primitives.

3.1 Encoding LIFT data types in Agda

To define the semantics of primitives, we firstly need to encode some core LIFT data types in Agda. In section 3.1.1, the representation of the set of data types, i.e., `data`, is introduced. In section 3.1.2 and 3.1.3, the inductive definitions of the natural number type and the array type are discussed.

3.1.1 The set of data types

In LIFT's type system, `data` represents the set of data types, including the natural number type, the array type, the index type etc. In Agda, we use the universe `Set` to encode LIFT's `data` type.

Agda uses universes to resolve Russell's paradox, which states that the collection of all sets cannot be a set itself (Agda Developer Team). The universe `Set` is a collection of Agda types, for example, both `Bool` and `ℕ` have the type `Set`. However, the `Set` itself cannot have the type `Set`, to give `Set` a type, we can use a higher level of universe Set_1. There are infinite number of universes, each of which is an element of the next higher one.

3.1.2 Natural number type

We use the Agda built-in inductive definition of natural numbers to represent the natural number type (Agda Developer Team 2020). The definition is shown as below:

```
-- The definition of natural numbers in Agda
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

It says to construct a natural number, we need: 1) a base case stating `zero` is a natural number and 2) an inductive case stating if n is a natural number, then (`suc` n) is also a natural number.

Following this definition of natural numbers, the addition option between two natural numbers can be defined inductively:

```
-- The definition of natural number addition in Agda
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)
```

Then natural number multiplication can be defined as a repetition of addition:

```
-- The definition of natural number multiplication in Agda
_*_ : ℕ → ℕ → ℕ
zero * m = zero
suc n * m = m + n * m
```

3.1.3 Array type

The array type in LIFT represents an indexed collection of elements which have the same data type. For example, the notation $n_\bullet s$ in 3.1 means an array of type s elements with size n, where n is a natural number. Formally, the array type is defined as:

$$\frac{\Delta \vdash N : \mathsf{nat} \qquad \Delta \vdash T : \mathsf{data}}{\Delta \vdash N_\bullet T}$$

Although there is no array type defined in the Agda standard library, the vector type Vec defined as below satisfies the definition of LIFT's array type:

```
-- The definition of an indexed collection data type in Agda
data Vec (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

Note that in this definition, a is an arbitrary universe level and Set a is a universe at level a, a vector can contain elements in any universe level hence the generality of the definition is preserved. The constructors are able to construct vectors with the consideration of their indices, i.e., the number of elements in a vector. The base case constructs a vector of zero elements and the inductive case says that when never a new element is added, the indices are increased by one, while all the elements in the vector remain unchanged.

Some functions are also defined to manipulate vectors. Firstly, a function to build a vector with only element is introduced below:

```
-- Construct a vector with only one element
[_] : A → Vec A 1
[ x ] = x :: []
```

For example, [1] builds a vector containing only one element which is the natural number 1.

Vector concatenation is defined by performing induction on the first vector as:

```
-- The definition of vector concatenation
_++_ : {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

To avoid confusion in terminologies, although LIFT's array type is encoded as the vector type in Agda, we continue using "array" to refer to this data type in the rest of this paper.

3.2 Encoding LIFT primitives in Agda

Based on the data types introduced in section 3.1, a set of LIFT's primitives can be defined. In section 3.2.1, the definition of some basic primitives including `id`, `map`, `take`, `drop`, `split` and `join` are introduced. In section 3.2.2, primitives for reducing an array such as `reduce` and `partRed` are defined. The transpose operation of a two-dimensional (2D) array is defined in section 3.2.3 and finally in section 3.2.4, we formalise some functions for stencil computation, which is an optimisation strategy for updating values in a multi-dimensional array by creating neighbour values with fixed pattern (Hagedorn et al. 2018).

3.2.1 Basic primitives

The `id` primitive in LIFT can be viewed as an operation which copies a value. Its definition is straightforward:

```
-- The definition of primitive id
id : {T : Set} → T → T
id t = t
```

It says the `id` primitive takes in an argument with an arbitrary type and returns the argument itself. Note that in Agda braces indicate implicit arguments in declarations which are able to be figured out by the compiler, thus we do not need to explicitly provide T in the definition.

The `map` primitive, which is formally defined in 3.1, is an operation applying a given function to every element in an array. It can be defined as below:

```
-- The definition of primitive map
map : {n : ℕ} → {S T : Set} → (S → T) → Vec S n → Vec T n
map f [] = []
map f (x :: xs) = f x :: map f xs
```

It can be pointed out that the first line in the Agda code, which is the declaration, matches that the mathematical formalisation in 3.1. Moreover, the computation steps are specified inductively with a base case stating that applying `map` of a function to an empty array results an empty array; and an inductive case stating that applying `map` of a function to a non-empty array yields applying the function to the first element and then mapping the function to the rest of the array.

`take` and `drop` are a pair of opposite operations. Formally `take` is defined as:

$$\textbf{take} : \quad (n : \mathsf{nat}) \to \{m : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to (n + m)_\bullet t \to n_\bullet t \tag{3.2}$$

which means taking the first n elements from an array with size $(n + m)$; and `drop` is defined as:

$$\textbf{drop} : \quad (n : \mathsf{nat}) \to \{m : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to (n + m)_\bullet t \to m_\bullet t \tag{3.3}$$

which is an operation discarding the first n elements from an array with size $(n + m)$. We define them in Agda as below:

```
-- The definition of primitive take
take : (n : ℕ) → {m : ℕ} → {T : Set} → Vec T (n + m) → Vec T n
take zero xs = []
take (suc n) {m} (x :: xs) = x :: (take n {m} xs)

-- The definition of primitive drop
drop : (n : ℕ) → {m : ℕ} → {T : Set} → Vec T (n + m) → Vec T m
drop zero xs = xs
drop (suc n) (x :: xs) = drop n xs
```

Again, the computation steps are defined by induction, with base cases stating that taking/dropping zero element from an array and inductive cases specifying taking/dropping at least one element from an array.

Making use of the definition of `take` and `drop`, we can give the primitive `split` concise semantics. `split` is an operation producing a multi-dimensional array by slicing an array into chunks with a given size, which is formally defined as:

$$\textbf{split}: \quad (n : \mathsf{nat}) \to \{m : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to mn_\bullet t \to m_\bullet n_\bullet t \tag{3.4}$$

where the notation $mn_\bullet t$ indicates an array with size $(m * n)$ and $m_\bullet n_\bullet t$ is a multi-dimensional array, i.e., a size m array of arrays with size n.

In Agda, we encode this operation as:

```
-- The definition of primitive split
split : (n : ℕ) → {m : ℕ} → {T : Set} → Vec T (m * n) → Vec (Vec T n) m
split n {zero} xs = []
split n {suc m} xs = take n {m * n} xs :: split n (drop n xs)
```

The inductive case is defined using `take` and `drop`, which means to split an array into at least one chunk of size n, we take a size n chunk from it, and then build it with the result of splitting the rest of the array.

The primitive `join` can be viewed as an opposite operation of `split`. It is an operation joining the two outer dimensions of a multi-dimensional array, reducing the total number of its dimensions by one. Formally it is defined as:

$$\textbf{join}: \quad \{n\ m : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to n_\bullet m_\bullet t \to mn_\bullet t \tag{3.5}$$

Again, its computation steps can be defined inductively in Agda:

```
-- The definition of primitive join
join : {n m : ℕ} → {T : Set} → Vec (Vec T n) m → Vec T (m * n)
join [] = []
join (xs :: xs₁) = xs ++ join xs₁
```

The base case is straightforward, joining an empty array gives an empty array; the inductive case states that joining an non-empty array means concatenating the first chunk of the array with the result of joining the rest of the array.

3.2.2 Reduction primitives

In general, a reduction operation is to reduce an array into a single value. There are three reduction primitives in LIFT: `reduce`, `reduceSeq`, and `partRed` (i.e., partial reduction).

The `reduce` primitive performs a reduction on an array with a given binary operator which is associative and commutative, taking two arguments of the same type. `reduceSeq` is a more general operation than `reduce`, which takes a binary operator (not required to be associative or commutative) that is able to operating on two arguments with different types. The `partRed` operation is more complicated, instead of reducing an array into a single element, it reduces an array into an array, where its size is a given fraction of the original size.

We start with looking into the most general reduction operation, i.e., `reduceSeq`, which is formally defined as:

$$\textbf{reduceSeq}: \quad \{n : \mathsf{nat}\} \to \{s\ t : \mathsf{data}\} \to (s \to t \to t) \to t \to n_\bullet s \to t \tag{3.6}$$

Note that $(s \to t \to t)$ is a user specified binary operator, and t denotes the initial argument with data type t. We define it in Agda as:

```
-- The definition of primitive reduceSeq
reduceSeq : {n : ℕ} → {S T : Set} → (S → T → T) → T → Vec S n → T
reduceSeq f init [] = init
reduceSeq f init (x :: xs) = reduceSeq f (f x init) xs
```

The base case is when operating on an empty array, the resulting value will be the initial argument *init*. The inductive case states that firstly the binary operator is applied on *init* and the first element of the array, the resulting value will be used as the initial argument for the next `reduceSeq` operation on the rest of the array.

Since the most general reduction primitive is defined, we can build the semantics of `reduce` upon it. In the definition of `reduce`:

$$\textbf{reduce}: \qquad \{n : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to (t \to t \to t) \to t \to n_\bullet t \to t \tag{3.7}$$

the binary operator $(t \to t \to t)$ has a restriction to be both associative and commutative. Moreover, the initial argument need to be an *id* element of the operator. An *id* element for a binary operator means whenever it is taken by the operator with another argument, the result is always the other argument. For instance, the natural number 0 is the *id* element of the natural number addition $(+)$, and the natural number 1 is the *id* element of the natural number multiplication $(*)$.

In order to define `reduce`, we firstly need to define the type of these associative and commutative operators with *id* elements:

```
-- The definition of associative and commutative operator
record CommAssocMonoid (A : Set) : Set₁ where
  field
    ε     : A
    _⊕_   : A → A → A
    id^l  : ∀ x → ε ⊕ x ≡ x
    id^r  : ∀ x → x ⊕ ε ≡ x
    assoc : ∀ x y z → (x ⊕ y) ⊕ z ≡ x ⊕ (y ⊕ z)
    comm  : ∀ x y → x ⊕ y ≡ y ⊕ x
```

We define this binary operator type as an Agda `record` type called `CommAssocMonoid`. The record type in Agda is a group of values. In this `CommAssocMonoid`, ε is the *id* element for a binary operator $\oplus$, id^l and id^r indicate the identity property of the *id* element ε; `assoc` and `comm` indicate the associative and commutative properties of the operator $\oplus$.

Now the `reduce` primitive can be defined in Agda as:

```
-- The definition of primitive reduce
reduce : {n : ℕ} → {T : Set} → (M : CommAssocMonoid T) → Vec T n → T
reduce M xs = let _⊕_ = _⊕_ M; ε = ε M
              in reduceSeq _⊕_ ε xs
```

Note that the let-expression is an abbreviation for name binding. This definition says the `reduce` is an `reduceSeq` operation with the binary operator being associative and commutative. Also, the initial argument has to be the *id* element of the operator.

Partial reduction is a more complicated operation, a definition is provided by Steuwer (2015):

$$partRed\ (\oplus)\ id_\oplus\ xs : [\alpha]_{\frac{n}{m}} \tag{3.8}$$

where both m and n are natural numbers. $(\oplus)$ is an associative and commutative binary operator with the type $(\alpha \to \alpha \to \alpha)$ and $id_\oplus$ denotes its id element having the type α. xs is an array of type α elements with size n. The resulting array has the size $\frac{n}{m}$.

More specifically, how the operation is done is defined as:

$$partRed\ (\oplus)\ id_\oplus\ n\ [x_1, x_2, ..., x_n] = [x_{\sigma(1)} \oplus ... \oplus x_{\sigma(n)}, ..., x_{\sigma(n-m+1)} \oplus ... \oplus x_{\sigma(n)}] \tag{3.9}$$

where σ is a permutation function over the indices, meaning that the partial reduction is a operation reducing an array without considering the elements' order in it.

There are some limitations of this definition. Firstly, the size of the resulting array is expressed as an fraction $\frac{n}{m}$, however, it is not explicitly showing the requirement that $\frac{n}{m}$ being a natural number, i.e., n is divisible by m. Secondly, the important restriction that the resulting array needs to be a non-empty array is not stated.

We define the partial reduction operation as `partRed` in Agda as below. For the simplicity of the definition, we do not consider the reordering of elements in the array at this stage:

```
-- The definition of primitive partRed
partRed : (n : ℕ) → {m : ℕ} → {T : Set} →
          (M : CommAssocMonoid T) → Vec T (suc m * n) → Vec T (suc m)
partRed zero {zero} M [] = let ε = ε M
                           in [ ε ]
partRed (suc n) {zero} M xs = [ reduce M xs ]
partRed zero {suc m} M [] = let ε = ε M
                            in ε :: partRed zero {m} M []
partRed (suc n) {suc m} M xs = [ reduce M (take (suc n) {suc (n + (m * suc n))} xs) ]
                               ++ partRed (suc n) {m} M ((drop (suc n) xs))
```

In this definition, instead of defining the size of the resulting array as a fraction, we define the size of the input array as an multiplication of $suc\ m$ and n, ensuring it is divisible by n according to the requirement. Also, defining the size of resulting array as $suc\ m$ indicates that the resulting array cannot be empty.

3.2.3 Transpose

The `transpose` primitive is defined as the matrix transpose in linear algebra, which is an operation producing a matrix by switching the row and column indices of the original matrix. Note that if the input multi-dimensional array has more than two dimensions, the transpose operation will be done on the two outer most dimensions. Formally, it is defined as:

$$\textbf{transpose}: \quad \{n\ m : \mathsf{nat}\} \to \{t : \mathsf{data}\} \to n_\bullet m_\bullet t \to m_\bullet n_\bullet t \tag{3.10}$$

Before defining the transpose operation in Agda, we need to introduce three helper functions: `fill`, `head` and `tail` as below:

```
-- Helpers for defining transpose
fill : {T : Set} → (n : ℕ) → T → Vec T n
fill zero x = []
fill (suc n) x = x :: fill n x

head : {T : Set} → {n : ℕ} → Vec T (suc n) → T
head (x :: xs) = x

tail : {T : Set} → {n : ℕ} → Vec T (suc n) → Vec T n
tail (x :: xs) = xs
```

fill is an operation constructing an array of n copies of an given argument x. head and tail are a pair of opposite operations, head returns the first element of an given array while tail returns the given array without the first element.

Making use of these three helper functions, we can define the `transpose` primitive as:

```
-- The definition of primitive transpose
transpose : {n m : ℕ} → {T : Set} → Vec (Vec T m) n → Vec (Vec T n) m
transpose {suc n} {zero} xss = []
transpose {zero} {m} [] = fill _ []
transpose {suc n} {suc m} xss = map head xss :: transpose (map tail xss)
```

Slightly different for the definition of matrix transpose in linear algebra, we consider the situations where the inner and outer dimensions have size zero. The first base case states that transposing an array of empty arrays gives an empty array. The second base case says, if we would like to transpose an empty array, the result will be an array of m empty arrays, where m is the given size of the original inner dimension. Finally, the inductive case defines that when the sizes of both dimensions are not zero, we take the heads of all the inner arrays, and build them with the result of transposing tails of all the inner arrays.

3.2.4 Stencil computation primitives

In this section, some primitives introduced by Hagedorn et al. (2018) for enabling overlapped tiling optimisation for stencil computations, which are algorithms widely used in machine learning, medical imaging etc., are formalised in Agda.

We firstly discuss about the primitive `slide`, which is an operation creating neighbour values. It operates a sliding window of a given size moving past number of elements with a given step to traverse till the end of an array. For example, if we apply `slide` to the array $[0, 1, 2, 3, 4]$ with a given size 3 and a given step 1, we will get the result:

$$[[0, 1, 2], [1, 2, 3], [2, 3, 4]]$$

A formal definition of `slide` is shown below:

$$\textbf{slide}: \quad \{n : \mathsf{nat}\} \to (sz\ sp : \mathsf{nat}) \to \{t : \mathsf{data}\} \to (sp \cdot n + sz - sp)_\bullet t \to n_\bullet sz_\bullet t \tag{3.11}$$

where sz is the size of a sliding window and sp is a given step, note that both n and sp are larger than zero. The `split` primitive can be viewed as a special case of `slide` when the size and step are the same.

Similar to defining `split`, slide is also inductively defined with `take` and `drop`:

```
-- The definition of primitive slide
slide : {n : ℕ} → (sz : ℕ) → (sp : ℕ)→ {T : Set} → Vec T (sz + n * (suc sp)) →
        Vec (Vec T sz) (suc n)
slide {zero} sz sp xs = [ xs ]
slide {suc n} sz sp xs =
      take sz {(suc n) * (suc sp)} xs ::
      slide {n} sz sp (drop (suc sp) (cast (slide-lem n sz sp) xs))
```

The base case is very intuitive, which means when the size of the array is sz, the result is an array of the input array. The inductive case is similar to the inductive case of `split`, however, we need to cast the size of the array into a suitable pattern to perform the `drop` operation. The cast operation does not alter the size or the elements in an array, all it does is merely changing the numerical pattern of the array size in the type. It is defined as:

```
-- Helper cast, casting the size of a given array
cast : {T : Set} → {m n : ℕ} → .(_ : m ≡ n) → Vec T m → Vec T n
cast {T} {zero} {zero} eq [] = []
cast {T} {suc m} {suc n} eq (x :: xs) = x :: cast {T} {m} {n} (cong pred eq) xs
```

It states that if it is possible to provide an equality relation between two natural numbers m and n, where m is the size of a given array, then we can construct an array of size n which is equal to the original array.

The `slide-lem` used in the definition of `slide` is such an equality relation:

```
slide-lem : (n sz sp : ℕ) →
            suc (sz + (sp + n * suc sp)) ≡ suc (sp + (sz + n * suc sp))
```

The proof of this lemma is omitted for brevity. With this lemma, we are able to cast the size of an array with pattern $suc\ (sz + (sp + n * suc\ sp))$ into the pattern $suc\ (sp + (sz + n * suc\ sp))$, to satisfy the pattern (`drop` $(suc\ sp)$) requires.

Padding is an operation for extending the boundary of an array by adding given numbers of elements at the beginning and end of an array. There are two types of padding in LIFT, `padCst` and `padClamp`. The difference is that the elements `padCst` adds to a given array have a constant value; however in `padClamp`, the values are generated by a function. We only discuss `padCst` in this paper.

The primitive `padCst` is defined as:

$$\textbf{padCst}: \quad \{n : \text{nat}\} \to (l\ r : \text{nat}) \to \{t : \text{data}\} \to t \to n._{\bullet}t \to (l + n + r)._{\bullet}t \tag{3.12}$$

where l and r are the numbers of elements to be added at the beginning and end of the given array. t indicates the constant value of these elements. An example of the `padCst` operation is given below, note that we choose $l = 2$, $r = 3$, and the constant value is 0:

$$padCst\ 2\ 3\ 0\ [3, 4, 5] = [0, 0, 3, 4, 5, 0, 0, 0]$$

We define `padCst` in Agda by breaking the operation down into adding elements at the beginning and adding elements add the end. Combining these two operations together gives `padCst`:

```
-- Adding elements at the begining
padCst^l : {n : ℕ} → (l : ℕ) → {T : Set} → T → Vec T n → Vec T (l + n)
padCst^l zero x xs = xs
padCst^l (suc l) x xs = padCst^l l x ([ x ] ++ xs)

-- Adding elements at the end
padCst^r : {n : ℕ} → (r : ℕ) → {T : Set} → T → Vec T n → Vec T (n + r)
padCst^r zero x xs = xs
padCst^r (suc r) x xs = padCst^r r x (xs ++ [ x ])

-- The definition of primitive padCst
padCst : {n : ℕ} → (l r : ℕ) → {T : Set} → T → Vec T n → Vec T (l + n + r)
padCst l r x xs = padCst^r r x (padCst^l l x xs)
```

Both `slide` and `padCst` are one-dimensional operations for neighbour value creation and array boundary handling, we can extend them to handle operations on multiple dimensions. According to Hagedorn et al. (2018), the 2D slide operation is defined as:

$$slide_2\ (size,\ step,\ input) = map\ (transpose,\ slide\ (size,\ step,\ map\ (slide\ (size, step),\ input))) \tag{3.13}$$

We formalise it in Agda as:

```
-- The definition of 2D slide
slide₂ : {n m : ℕ} → (sz : ℕ) → (sp : ℕ)→ {T : Set} →
         Vec (Vec T (sz + n * (suc sp))) (sz + m * (suc sp)) →
         Vec (Vec (Vec (Vec T sz) sz) (suc n)) (suc m)
slide₂ {n} {m} sz sp xs = map transpose (slide {m} sz sp (map (slide {n} sz sp) xs))
```

Comparing to the definition in 3.13, this definition indicates: 1) the 2D slide operation adds two more dimensions to the input multi-dimensional array and 2) the restrictions of the size of each dimension of input multi-dimensional array and the shape of the result.

Although there no straightforward definition of a 2D `padCst` operation, we can adapt the definition from a general form of n-dimensional padding:

$$pad_n\ (l,\ r,\ h,\ input) = map_{n-1}\ (pad\ (l,\ r,\ h),\ pad_{n-1}\ (l,\ r,\ h,\ input)) \tag{3.14}$$

where map_n is defined as:

$$map_n\ (f,\ input) = map_{n-1}\ (map(f),\ input) \tag{3.15}$$

Based on this definition, the 2D `padCst` operation is defined in Agda as:

```
-- The definition of 2D padCst
padCst₂ : {n m : ℕ} → (l r : ℕ) → {T : Set} → T → Vec (Vec T n) m →
          Vec (Vec T (l + n + r)) (l + m + r)
padCst₂ {n} l r x xs = map (padCst l r x) (padCst l r (fill n x) xs)
```

where `fill` is defined in section 3.2.3, It says the inner dimension is extended with the given constant value x and outer dimension is extended with arrays of that given constant value.

3.2.5 Simplifying semantics with REWRITE

Since Agda is dependently typed, the type of an array depends not only on the type of the elements inside the array but also the size of the array. Sometimes this can introduce complexities in defining semantics of array operations as well as developing proofs. We have to come up with some strategies to cope with this feature. For instance, in the definition of `slide`, we have to use `cast` to adjust the pattern of an array's size to satisfy the definition of `drop`. Moreover, we will later discuss about a mechanism in proving the equality between two different types in section 4.2.5. In this section, we introduce a way to increase the flexibility of Agda's type system to simplify the primitives.

Take the definition of `padCst` as an example, originally we needed to define it as:

```
-- The orignal definition of padCst
-- Adding elements at the begining
padCstˡ : {n : ℕ} → (l : ℕ) → {T : Set} → T → Vec T n → Vec T (l + n)
padCstˡ zero x xs = xs
padCstˡ (suc l) x xs = cast lem₁ (padCstˡ l x ([ x ] ++ xs))

-- Adding elements at the end
padCstʳ : {n : ℕ} → (r : ℕ) → {T : Set} → T → Vec T n → Vec T (n + r)
padCstʳ {n} zero x xs rewrite +-comm n zero = xs
padCstʳ (suc r) x xs = cast lem₂ (padCstʳ r x (xs ++ [ x ]))
```

```
-- The definition of primitive padCst
padCst : {n : ℕ} → (l r : ℕ) → {T : Set} → T → Vec T n → Vec T (l + n + r)
padCst l r x xs = padCstʳ r x (padCstˡ l x xs)
```

where lem_1 and lem_2 are defined as:

```
lem₁ : {l n : ℕ} → l + suc n ≡ suc (l + n)
lem₂ : {l n : ℕ} → n + 1 + l ≡ n + suc l
```

Compare to the definition we introduced in section 3.2.4, in this definition, the semantics of padCst^l and padCst^r are more complex requiring extra reasoning on the sizes of arrays. The reason causing this additional complexity is the definition used for natural number addition in section 3.1.2. In that definition, natural number addition is defined by induction on the first argument, therefore, in the base case of padCst^r:

```
padCstʳ {n} zero x xs rewrite +-comm n zero = xs
```

Although $(zero + n)$ can be reduced to n, Agda fails to reduce the pattern $(n + zero)$ to n, since it does not match the type of the base case in the definition. We have to explicitly provide the information that natural number addition is commutative to satisfy the pattern in the declaration. The rationale behind the definitions of the inductive cases is the same.

Consider an alternative definition of natural number addition by induction on the second argument:

```
-- An alternative definition of natural number addition
_+′_ : ℕ → ℕ → ℕ
n +′ zero = n
n +′ suc m = suc (n +′ m)
```

This is also a valid definition and allows the pattern matching in the definition of `padCst`. We would like Agda to allow both definitions in the type system using a built-in `REWRITE` feature (Cockx et al. 2019).

```
+zero : {m : ℕ} → m + zero ≡ m
+zero {zero} = refl
+zero {suc m} = cong suc +zero

+suc : {m n : ℕ} → m + (suc n) ≡ suc (m + n)
+suc {m} {n} =
  begin
    m + (1 + n)
  ≡⟨ sym (+-assoc m 1 n) ⟩
    m + 1 + n
  ≡⟨ cong (_+ n) (+-comm m 1) ⟩
    refl
{-# REWRITE +zero +suc #-}
```

We first prove these two equality relations to be valid, and then add them into Agda type system using the `REWRITE` pragma to make the pattern matching of the natural number addition allow both definitions. Also, we need to ensure the rules added do not break the confluence of the type system using the flag `--confluence-check` in the option enabling the built-in `REWRITE`:

```
{-# OPTIONS --rewriting --prop --confluence-check #-}
```

Afterwards we can simplify the definition of `padCst` into the one introduced in section 3.2.4.

4 Equality Reasoning for Rewrite Rules

The core of LIFT's optimisation approach is a set of rewrite rules, which systematically transform high-level expressions written using primitives introduced in section 3.2, into semantically equivalent expressions. The main aim of this project is to develop mechanical proofs to show the correctness of these rewrite rules.

4.1 Algorithmic rules

Algorithmic rules produce representations of different algorithmic choices of the high-level expression, regardless the choice of low-level programming models. We verified six categories of algorithmic rules published in papers (Steuwer 2015; Steuwer et al. 2015) or implemented in source code (Hagdorn et al. 2020), which are: fusion rules, identity rules, simplification rules, the split-join rule, the reduction rule and partial reduction rules.

4.1.1 Fusion rules

The two rules discussed in this section are used for fusing consecutive operations. The first one fuses two consecutive `maps`. The second one fuses a general reduction operation `reduceSeq` and a `map`. Formally they are defined as:

$$map\ f \circ map\ g \rightarrow map\ (f \circ g) \tag{4.1}$$

$$reduceSeq\ (\oplus)\ id_{\oplus} \circ map\ f \rightarrow reduceSeq\ (\lambda(a, b).\ a \oplus (f\ b))\ id_{\oplus} \tag{4.2}$$

Proving the correctness of them are straightforward. We first use the proof of 4.1 as an illustration of prove by induction and the reasoning chain in Agda:

```
-- The complete reasoning chain of
-- proving the first fusion rule by induction
fusion₁ : {n : ℕ} → {S T R : Set} → (f : T → R) → (g : S → T) → (xs : Vec S n) →
          (map f ∘ map g) xs ≡ map (f ∘ g) xs
fusion₁ f g [] =
  begin
    (map f ∘ map g) []
  ≡⟨⟩
    map f (map g [])
  ≡⟨⟩
    map f []
  ≡⟨⟩
    []
  ≡⟨⟩
    map (f ∘ g) []
  ∎
```

```
fusion₁ f g (x :: xs) =
  begin
    (map f ∘ map g) (x :: xs)
  ≡⟨⟩
    map f (map g (x :: xs))
  ≡⟨⟩
    map f (g x :: map g xs)
  ≡⟨⟩
    f (g x) :: map f (map g xs)
  ≡⟨⟩
    (f ∘ g) x :: map f (map g xs)
  ≡⟨ cong ((f ∘ g) x ::_ ) (fusion₁ f g xs) ⟩
    (f ∘ g) x :: map (f ∘ g) xs
  ■
```

The construction of the equality reasoning is based on Curry-Howard Correspondence (section 2.2), we show the semantic equivalence of *map f ∘ map g* and *map (f ∘ g)* by proving the propositional equality (denoted as ≡) of their types. The reasoning chain starts with begin and ends with ■ (i.e., Q.E.D); steps are separated by ≡⟨⟩; within each ≡⟨⟩, reasoning of achieving the next step can be provided.

The equality is proven by induction on the input array *xs*. The base case is: when the input array is empty, the right-hand side of the equation is trivially equal to the left-hand side. The inductive case states given the equality holds for *xs*, we are able to prove it holds for (*x* :: *xs*) .

Note that the cong function in the inductive case represents congruence, which is relation for a given function when it is preserved by applying a given function, it is defined in Agda as:

```
cong : {A B : Set} → ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
```

Equality satisfies congruence. For a given function *f* and equality relation $x \equiv y$, the congruence is a relation $f\ x \equiv f\ y$.

In this proof, cong ((*f* ∘ *g*) *x* ::_) (fusion₁ *f g xs*) means given the equality relation: (fusion₁ *f g xs*) and the function (*f* ∘ *g*) *x* ::_), we can conclude (*f* ∘ *g*) *x* :: map *f* (map *g xs*) ≡ (*f* ∘ *g*) *x* :: map (*f* ∘ *g*) *xs*.

This proof is rather long, however, we can make it concise by removing the steps which are able to be figured out by Agda automatically. Note that when there is no required justification inside a ≡⟨⟩, the next step is trivial to be derived in Agda. This proof can be rewritten as:

```
-- The concise proof of the first fusion rule
fusion₁ : {n : ℕ} → {S T R : Set} → (f : T → R) → (g : S → T) → (xs : Vec S n) →
          (map f ∘ map g) xs ≡ map (f ∘ g) xs
fusion₁ f g [] = refl
fusion₁ f g (x :: xs) = cong ((f ∘ g) x ::_ ) (fusion₁ f g xs)
```

Note that refl is the reflexivity of equality. Proving a term equals to itself can be written as refl.

Similar to the first fusion rule, the second fusion rule can also be intuitively proven by induction:

```
-- The proof of the second fusion rule
fusion₂ : {n : ℕ} → {S T R : Set} →
          (f : S → T) → (bf : T → R → R) → (init : R) → (xs : Vec S n) →
          reduceSeq (λ (a : S) (b : R) → (bf (f a) b)) init xs ≡
          (reduceSeq bf init ∘ map f) xs
```

```
fusion₂ f bf init [] = refl
fusion₂ f bf init (x :: xs) = fusion₂ f bf (bf (f x) init) xs
```

Note that this rule only holds for the general reduction pattern `reduceSeq`, which does not require the binary operator being associative or commutative. The initial value is not required to be an *id* element defined in section 3.2.2 either.

4.1.2 Identity rules

The first two identity rules state that composing any function operating on an array with `id` does not change the output of the function. Formally, they are defined as (Steuwer 2015):

$$f \rightarrow f \circ map\ id \mid map\ id \circ f \tag{4.3}$$

To prove these two rules, we firstly need to introduce a lemma map-id, which is an equality relation map id $xs \equiv xs$:

```
map-id : {n : ℕ} → {T : Set} → (xs : Vec T n ) → map id xs ≡ xs
map-id [] = refl
map-id (x :: xs) = cong (x ::_) (map-id xs)
```

Using this lemma, we can easily prove the correctness of the first identity rule:

```
-- The first identity rule
identity₁ : {n : ℕ} → {S T : Set} → (f : Vec S n → Vec T n) → (xs : Vec S n) →
            (f ∘ map id) xs ≡ f xs
identity₁ f xs = cong f (map-id xs)
```

Again, we make use of congruence and the proven lemma here. cong f (map-id xs) says that given a function f and an equality relation map id $xs \equiv xs$, we are able to derive the equality relation f (map id) $xs \equiv f\ xs$.

Similarly, we can prove the second identity rule:

```
-- The second identity rule
identity₂ : {n : ℕ} → {S T : Set} → (f : Vec S n → Vec T n) → (xs : Vec S n) →
            (map id ∘ f) xs ≡ f xs
identity₂ f xs = map-id (f xs)
```

Originally, this rule has been proven in paper (Steuwer 2015):

$$\begin{aligned}
&\text{Let } xs = [x_1, ..., x_n].\\
&\quad (map\ id \circ f)\ xs = map\ id\ (f\ xs)\\
&\qquad\qquad \{\text{definition of map}\}\\
&\qquad\qquad = [id\ (f\ xs)_1, id\ (f\ xs)_2, ..., id\ (f\ xs)_n]\\
&\qquad\qquad \{\text{definition of id}\}\\
&\qquad\qquad = f\ xs
\end{aligned}$$

This proof is not correct. Since f is a function applying to an array but not to each element of the array, in the second step, f should not be applied to elements in xs. The proof in Agda is free from this mistake, as the type of f is defined as (Vec $S\ n \rightarrow$ Vec $T\ n$) and the array xs has type (Vec $S\ n$), xs is a valid argument of f. The compiler will complain about the type error if we try to apply f to each element in xs.

The third identity rule states if we transpose a 2D array twice, the resulting array is equal to the input array itself. It can be formally defined as:

$$transpose \circ transpose \rightarrow id \tag{4.4}$$

The proof in Agda is more complex than the first two:

```
-- The third identity rule: (A^T)^T ≡ A
identity₃ : {n m : ℕ} → {T : Set} → (xss : Vec (Vec T m) n) →
            transpose (transpose xss) ≡ xss
identity₃ {suc n} {zero} ([] :: xss) = cong ([] ::_) (fill-empty xss)
identity₃ {zero} {m} [] = empty (transpose (fill _ []))
identity₃ {suc n} {suc m} ((x :: xs) :: xss) =
  begin
    (x :: map head (transpose (xs :: map tail xss))) ::
    transpose (map head xss :: map tail (transpose (xs :: map tail xss)))
  ≡⟨ cong (_:: transpose (map head xss :: map tail (transpose (xs :: map tail xss))))
    (cong (x ::_) (transpose-head xs xss)) ⟩
    (x :: xs) :: transpose (map head xss :: map tail (transpose (xs :: map tail xss)))
  ≡⟨ cong (λ yss → (x :: xs) :: transpose yss) (transpose-tail xs xss) ⟩
    (x :: xs) :: transpose (transpose xss)
  ≡⟨ cong ((x :: xs) ::_) (identity₃ xss) ⟩
    refl
```

The three cases in the proof match the three cases in the definition of `transpose`. Each case is broken down into smaller lemmas, which are equality reasoning about the expressions involving helper functions used for defining `transpose`, i.e., `fill`, `head` and `tail`. In the third case, we conclude this proof by induction.

4.1.3 Simplification rules

Each simplification rule specifies a pair of consecutive operations that can be eliminated. We discuss about the split-join pair and join-split pair in this section. Formally they are defined as:

$$join \circ split\ n \rightarrow id \tag{4.5}$$

$$split\ n \circ join \rightarrow id \tag{4.6}$$

Again, both of them can be proven by induction, some smaller lemmas of equality reasoning on expressions involving `take` and `drop` are necessary since `split` is defined using them. The proof of the first rule 4.5 is shown below:

```
-- The proof of the first simplification rule
simplification₁ : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec T (m * n)) →
                  (join ∘ split n {m}) xs ≡ xs
simplification₁ n {zero} [] = refl
simplification₁ n {suc m} xs =
  begin
    take n xs ++ join (split n {m} (drop n xs))
  ≡⟨ cong (take n xs ++_) (simplification₁ n {m} (drop n xs)) ⟩
    take n xs ++ drop n xs
  ≡⟨ take-drop n xs ⟩
    refl
```

where take-drop is an lemma says the concatenation of the result of taking n elements from an array and the result of dropping n elements from that array is the original array itself, we prove it in Agda:

```
take-drop : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec T (n + m)) →
              take n {m} xs ++ drop n {m} xs ≡ xs
take-drop zero xs = refl
take-drop (suc n) (x :: xs) = cong (x ::_) (take-drop n xs)
```

The proof of the second simplification rule 4.6 has more steps comparing to the first one but is still straightforward:

```
-- The proof of the second simplification rule
simplification₂ : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec (Vec T n) m) →
                    (split n ∘ join) xs ≡ xs
simplification₂ n {zero} [] = refl
simplification₂ n {suc m} (xs :: xs₁) =
  begin
    take n (xs ++ join xs₁) :: split n (drop n (xs ++ join xs₁))
  ≡⟨ cong (_:: split n (drop n (xs ++ join xs₁))) (take-++ n xs (join xs₁)) ⟩
    xs :: split n (drop n (xs ++ join xs₁))
  ≡⟨ cong (xs ::_) (cong (split n) (drop-++ n xs (join xs₁))) ⟩
    xs :: split n (join xs₁)
  ≡⟨ cong (xs ::_) (simplification₂ n xs₁) ⟩
    refl
```

It requires a pair of lemmas take-++ and drop-++, which are defined as:

```
take-++ : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec T n) → (xs₁ : Vec T m) →
            take n {m} (xs ++ xs₁) ≡ xs
take-++ zero [] xs₁ = refl
take-++ (suc n) {m} (x :: xs) xs₁ = cong (x ::_) (take-++ n {m} xs xs₁)

drop-++ : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec T n) → (xs₁ : Vec T m) →
            drop n (xs ++ xs₁) ≡ xs₁
drop-++ zero [] xs₁ = refl
drop-++ (suc n) (x :: xs) xs₁ = drop-++ n xs xs₁
```

According to the proofs in section 4.1.2 and section 4.1.3 It can be pointed out that breaking proof down into smaller reusable proofs is an effective strategy for developing verification programs.

4.1.4 Split-join rule

The split-join rule allows a `map` to be partitioned into several nested `maps`, possibly handled by multiple threads. The rule is defined as:

$$map\ f \longrightarrow join \circ map\ (map\ f) \circ split\ n \tag{4.7}$$

To prove this rule, we need two lemmas, one is the first simplification rule 4.5 defined in section 4.1.3, the other one is a movement rule `splitBeforeMapMapF`. We will discuss more about movement rules is section 4.2.

The rule `splitBeforeMapMapF` is defined as:

```
-- The proof of split movement rule
splitBeforeMapMapF : (n : ℕ) → {m : ℕ} → {S T : Set} →
                     (f : S → T) → (xs : Vec S (m * n)) →
                     map (map f) (split n {m} xs) ≡ split n {m} (map f xs)
splitBeforeMapMapF n {zero} f xs = refl
splitBeforeMapMapF n {suc m} f xs =
  begin
    map f (take n xs) :: map (map f) (split n (drop n xs))
  ≡⟨ cong (map f (take n xs) ::_) (splitBeforeMapMapF n f (drop n xs)) ⟩
    map f (take n xs) :: split n (map f (drop n xs))
  ≡⟨ cong (_:: split n (map f (drop n xs))) (map-take n f xs) ⟩
    take n (map f xs) :: split n (map f (drop n xs))
  ≡⟨ cong (take n (map f xs) ::_) (cong (split n) (map-drop n f xs)) ⟩
    take n (map f xs) :: split n (drop n (map f xs))
  ■
```

It basically says that applying nested map operations after an array being split is the same as applying a single map operation to that array and then split it. The proof uses smaller lemmas as the equality reasoning of moving `map` operations around `take` and `drop`, since `split` is defined based on them. More details of proving movements rules and their lemmas will be discussed in section 4.2 later.

Now we can make use of these two rules to prove split-join rule, which is shown below:

```
-- The proof of split-join rule
splitJoin : {m : ℕ} → {S T : Set} →
            (n : ℕ) → (f : S → T) → (xs : Vec S (m * n)) →
            (join ∘ map (map f) ∘ split n {m}) xs ≡ map f xs
splitJoin {m} n f xs =
  begin
    join (map (map f) (split n {m} xs))
  ≡⟨ cong join (splitBeforeMapMapF n {m} f xs) ⟩
    join (split n {m} (map f xs))
  ≡⟨ simplification₁ n {m} (map f xs) ⟩
    map f xs
  ■
```

Since the equality relation is symmetric, it satisfies the original definition 4.7 by switching the expressions on the left-hand side and right-hand side. Comparing to the paper proof provided by Steuwer (2015), this proof is more structured, since it clearly states how it is constructed by the rules we have proven to be correct.

4.1.5 Reduction rule

The reduction rule states that a full reduction operation can be expressed and an composition of full reduction and partial reduction, allowing the full reduction operation to be partitioned, which is defined as:

$$reduce\ (\oplus)\ id_{\oplus} \rightarrow reduce\ (\oplus)\ id_{\oplus} \circ partRed\ (\oplus)\ id_{\oplus} \tag{4.8}$$

The proof is developed according to the four cases in the definition of primitive `partRed`:

```
-- The proof of redction rule
reduction : {m : ℕ} → {T : Set} → (n : ℕ) →
            (M : CommAssocMonoid T) → (xs : Vec T (suc m * n)) →
            (reduce M ∘ partRed n {m} M) xs ≡ reduce M xs
--- Base cases
reduction {zero} zero M [] = let _⊕_ = _⊕_ M; ε = ε M in (idʳ M ε)
reduction {zero} (suc n) M xs = let _⊕_ = _⊕_ M; ε = ε M in (idʳ M (reduce M xs))
reduction {suc m} zero M [] = let _⊕_ = _⊕_ M; ε = ε M in
  begin
    reduceSeq _⊕_ (ε ⊕ ε) (partRed zero {m} M [])
  ≡⟨ cong (λ y → reduceSeq _⊕_ y (partRed zero {m} M [])) (idʳ M ε) ⟩
    reduce M (partRed zero {m} M [])
  ≡⟨ reduction {m} zero M [] ⟩
    refl
-- Inductive case
reduction {suc m} (suc n) M xs = let _⊕_ = _⊕_ M; ε = ε M in
  begin
    reduce M ([ reduce M (take (suc n) {suc (n + (m * suc n))} xs) ] ++
    partRed (suc n) {m} M (drop (suc n) xs))
  ≡⟨ sym (reduce-++ M [ reduce M (take (suc n) {suc (n + (m * suc n))} xs) ]
     (partRed (suc n) {m} M (drop (suc n) xs))) ⟩
    reduce M [ reduce M (take (suc n) {suc (n + (m * suc n))} xs) ] ⊕
    reduce M (partRed (suc n) {m} M (drop (suc n) xs))
  ≡⟨ cong₂ (λ x y → x ⊕ y) (idʳ M (reduce M (take (suc n) {suc (n + (m * suc n))} xs)))
     (reduction {m} (suc n) M (drop (suc n) xs)) ⟩
    reduce M (take (suc n) {suc (n + (m * suc n))} xs) ⊕
    reduce M (drop (suc n) {suc (n + (m * suc n))} xs)
  ≡⟨ sym (reduce-take-drop (suc n) {suc (n + (m * suc n))} M xs) ⟩
    refl
```

Note that the congruence function cong_2 used in the inductive case is similar to the function cong introduced in section 4.1.1, but allowing us to reasoning about the congruence with two function applications instead of one.

The first and second case can be easily proven by the property of the *id* element. The third case is proven using a lemma:

```
reduceSeq-reduce : {n : ℕ} → {T : Set} →
                   (M : CommAssocMonoid T) → (x : T) → (xs : Vec T n) →
                   let _⊕_ = _⊕_ M in
                   x ⊕ reduce M xs ≡ reduceSeq _⊕_ x xs
```

which means performing the operation $\oplus$ with an argument x with the type T and the result of the full reduction of an array xs whose elements have the type T, is equal to a general reduce operation with operator $\oplus$, initial value x and the input array xs.

The last case is proven using two lemmas which are stated below:

```
reduce-++ : {n m : ℕ} → {T : Set} →
            (M : CommAssocMonoid T) → (xs₁ : Vec T n) → (xs₂ : Vec T m) →
            let _⊕_ = _⊕_ M in
            reduce M xs₁ ⊕ reduce M xs₂ ≡ reduce M (xs₁ ++ xs₂)
```

```
reduce-take-drop : (n : ℕ) → {m : ℕ} → {T : Set} →
                   (M : CommAssocMonoid T) → (xs : Vec T (n + m)) →
                   let _⊕_ = _⊕_ M in
                   reduce M xs ≡
                   reduce M (take n {m} xs) ⊕ reduce M (drop n {m} xs)
```

Basically, the lemma reduce-++ states performing the operation $\oplus$ on the results of reduction of two arrays xs_1 and xs_2 is equal to the reduction of the concatenation of these two arrays. And reduce-take-drop shows the reduction of an array is equal to the reduction of the array containing n elements taken from it and the reduction of it after dropping the first n elements being combined together by the operator $\oplus$.

The proofs of lemmas are omitted for brevity, but can be found in the project's source code.

4.1.6 Partial reduction rules

There are two partial reduction rules we would like to verify in this section. The first one states the condition when a partial reduction operation can result in a full reduction, which is formally defined as:

$$partRed\ (\oplus)\ id_\oplus\ n \longrightarrow reduce\ (\oplus)\ id_\oplus \tag{4.9}$$

According to the definition of `partRed` shown in section 3.2.2, this rule trivially holds:

```
-- The proof of the first partial reduction rule
partialReduction₁ : {T : Set} → (n : ℕ) →
                    (M : CommAssocMonoid T) → (xs : Vec T n) →
                    partRed n M xs ≡ [ reduce M xs ]
partialReduction₁ zero M [] = refl
partialReduction₁ (suc n) M xs = refl
```

The second rule is more complicated, it states the partial reduction can be done in parallel by partitioning the input array first. The definition is shown below:

$$partRed\ (\oplus)\ id_\oplus\ n \longrightarrow join \circ map\ (partRed\ (\oplus)\ id_\oplus\ n) \circ split\ m \tag{4.10}$$

The idea of proving this rule is similar to the construction of the proof for the split-join rule in section 4.1.4, we need to move the map operation to a suitable place and then cancel the split and join operation pair using a simplification rule. The lemma we need is:

```
map-join-partRed : {m : ℕ} → {T : Set} → (n : ℕ) →
                   (M : CommAssocMonoid T) → (xss : Vec (Vec T n) (suc m)) →
                   join (map (partRed n {zero} M) xss) ≡
                   partRed n {m} M (join {n} {suc m} xss)
```

It simply says joining a 2D array together after mapping partial reduction operations onto it equals to applying the partial reduction operation after joining this 2D array. Then we can prove this rule by applying this lemma and the first simplification rule 4.5:

```
-- The proof of the second partial reduction rule
partialReduction₂ : {m : ℕ} → {t : Set} → (n : ℕ) →
                    (M : CommAssocMonoid t) → (xs : Vec t (suc m * n)) →
                    (join ∘ map (partRed n {zero} M) ∘ split n {suc m}) xs ≡
                    partRed n {m} M xs
```

```
partialReduction₂ {m} n M xs =
  begin
    join (map (partRed n {zero} M) (split n {suc m} xs))
  ≡⟨ map-join-partRed {m} n M (split n {suc m} xs) ⟩
    partRed n M (join (split n {suc m} xs))
  ≡⟨ cong (partRed n M) (simplification₁ n {suc m} xs) ⟩
    refl
```

Again, since the equality relation is symmetric, we can conclude the second partial reduction rule 4.10 to be correct.

4.1.7 Summary

In this section, we firstly introduce proof by induction in Agda, and then encode a set of algorithmic rules defined by Steuwer (2015) and verify their correctness. Mechanical proofs are effective for verifying these parallel optimisation strategies for array operations. Comparing to paper proofs, each step of reason processes is well constructed and the reinforcement of type restrictions ensures the correctness of those proofs.

4.2 Movement rules

A movement rule moves the primitives' orders in a composed expression, to make its pattern matches the shape which we can apply algorithmic rules on. They have been implemented in source code but not yet verified (Hagdorn et al. 2020). The rule `splitBeforeMapMapF` introduced in section 4.1.4 is one of the movement rules, technically there is a pair of them, defined symmetrical to each other as:

$$map\ (map\ f) \circ split\ n \rightarrow split\ n \circ map\ f \tag{4.11}$$

$$split\ n \circ map\ f \rightarrow map\ (map\ f) \circ split\ n \tag{4.12}$$

`splitBeforeMapMapF` is the rule 4.11, it changes the order of `split` and `map` operations, in order to group `split` and `join` together allowing them to cancel each other. In the following sections, we are going to discuss more movement rules. The rule 4.12 can be proven correct since equality is symmetric.

4.2.1 Moving `map` around other primitives

There are a set of rules similar to `splitBeforeMapMapF`, which move `map` operations before or after other primitive operations while preserving the final results. We simply address each rule by the primitive combined with `map` in its expression, such as addressing `splitBeforeMapMapF` as the split movement rule.

As we discussed in section 3.2.4, `split` can be viewed as a special case of the primitive `slide` when the *size* is equal to *step*, therefore we can form a pair of slide movement rules similar to the pair of split movement rules:

$$map\ (map\ f) \circ slide\ sz\ sp \rightarrow slide\ sz\ sp \circ map\ f \tag{4.13}$$

$$slide\ sz\ sp \circ map\ f \rightarrow map\ (map\ f) \circ slide\ sz\ sp \tag{4.14}$$

They state applying nested `map` operations after `slide` equals to applying a single `map` before `slide`. Again, since they are symmetric to each other, proving the correctness of one of them shows they both hold. We prove the first slide movement rule 4.13 in Agda:

```
-- The proof of the first slide movement rule
slideBeforeMapMapF : {n : ℕ} → (sz sp : ℕ) → {S T : Set} →
                     (f : S → T) → (xs : Vec S (sz + n * (suc sp))) →
                     map (map f) (slide {n} sz sp xs) ≡ slide {n} sz sp (map f xs)
slideBeforeMapMapF {zero} sz sp f xs = refl
slideBeforeMapMapF {suc n} sz sp f xs = let ys = (cast (slide-lem n sz sp) xs) in
  begin
    map f (take sz xs) :: map (map f) (slide {n} sz sp (drop (suc sp) ys))
  ≡⟨ cong (_:: map (map f) (slide {n} sz sp (drop (suc sp) ys))) (map-take sz f xs) ⟩
    take sz (map f xs) :: map (map f) (slide sz sp (drop (suc sp) ys))
  ≡⟨ cong (take sz (map f xs) ::_) (slideBeforeMapMapF {n} sz sp f (drop (suc sp) ys))⟩
    take sz (map f xs) :: slide sz sp (map f (drop (suc sp) ys))
  ≡⟨ cong (λ y → take sz (map f xs) :: slide sz sp y) (map-drop (suc sp) f ys) ⟩
    cong (λ y → take sz (map f xs) :: slide sz sp (drop (suc sp) y))
    (map-cast {sz + suc (sp + n * suc sp)} {suc (sp + (sz + n * suc sp))} f xs
    (slide-lem n sz sp))
```

The general idea is still proof by induction. The proof of the base case is trivial, as for the inductive case, we break it down into three smaller lemmas, map-take, map-drop and map-cast.

```
map-take : (n : ℕ) → {m : ℕ} → {S T : Set} → (f : S → T) → (xs : Vec S (n + m)) →
           map f (take n {m} xs) ≡ (take n {m} (map f xs))
map-take zero f xs = refl
map-take (suc n) {m} f (x :: xs) = cong (f x ::_) (map-take n {m} f xs)

map-drop : (n : ℕ) → {m : ℕ} → {S T : Set} → (f : S → T) → (xs : Vec S (n + m)) →
           map f (drop n {m} xs) ≡ (drop n {m} (map f xs))
map-drop zero f xs = refl
map-drop (suc n) f (x :: xs) = map-drop n f xs

map-cast : {m n : ℕ} → {S T : Set} → (f : S → T) → (xs : Vec S m) → .(eq : m ≡ n) →
           map f (cast eq xs) ≡ cast eq (map f xs)
map-cast {zero} {zero} f [] eq = refl
map-cast {suc m} {suc n} f (x :: xs) eq =
         cong (f x ::_) (map-cast {m} {n} f xs (cong pred eq))
```

Basically, map-take says mapping a function onto an array after taking n elements from it equals taking n elements from the resulting array of mapping the function onto the input array. Similarly, map-drop and map-cast are the equality reasoning of changing orders of `map` operations with `drop` and `cast` in composed expressions.

With these three lemmas, we first change the order of `take` and `map`, allowing the us to apply `slideBeforeMapMapF` inductively on the expression; then we change the order of `drop` and `map`, allowing us to further change the order of `cast` and `map`, making the expression matches the pattern on the right-hand side of the definition.

Since we defined movement rules for `split`, we can also form movement rules for its opposite operation `join`: joining a 2D array before applying a single `map` equals to applying nested `map`s on that 2D array before the `join` operation. Formally, the pair of join movement rules is defined as below:

$$map\ f \circ join \longrightarrow join \circ map\ (map\ f) \tag{4.15}$$

$$join \circ map\ (map\ f) \longrightarrow map\ f \circ join \tag{4.16}$$

To prove these rules, it is necessary to prove a lemma which indicates concatenating two arrays together and then mapping a function onto the resulting array is equal to mapping the function onto the two arrays separately before concatenating them together:

```
map-++ : {n m : ℕ} → {S T : Set} →
         (f : S → T) → (xs₁ : Vec S n) → (xs₂ : Vec S m) →
         map f (xs₁ ++ xs₂) ≡ map f xs₁ ++ map f xs₂
map-++ f [] xs₂ = refl
map-++ f (x :: xs₁) xs₂ = cong (f x ::_) (map-++ f xs₁ xs₂)
```

Now the rule 4.15 in can be proven by using the lemma `map-++`:

```
-- The proof of the first join movement rule
joinBeforeMapF : {S T : Set} → {m n : ℕ} →
                 (f : S → T) → (xs : Vec (Vec S n) m) →
                 map f (join xs) ≡ join (map (map f) xs)
joinBeforeMapF f [] = refl
joinBeforeMapF f (xs :: xs₁) =
  begin
    map f (xs ++ join (xs₁))
  ≡⟨ map-++ f xs (join xs₁) ⟩
    map f xs ++ map f (join xs₁)
  ≡⟨ cong (map f xs ++_) (joinBeforeMapF f xs₁) ⟩
    refl
```

The `map` operation is moved before the array concatenation, allowing `joinBeforeMapF` to be applied in the inductive case.

The last pair of rules to be discussed about in this section is a pair of transpose movement rules, stating that applying nested `map`s before transposing a 2D array is equal to transposing the 2D array before applying the nested `map` operations. The formal definitions are shown below:

$$transpose \circ map\ (map\ f) \rightarrow map\ (map\ f) \circ transpose \tag{4.17}$$

$$map\ (map\ f) \circ transpose \rightarrow transpose \circ map\ (map\ f) \tag{4.18}$$

The proof of the rule 4.17 has three cases matching the definition of transpose (section 3.2.3):

```
-- The proof of the first transpose movement rule
mapMapFBeforeTranspose : {n m : ℕ} → {S T : Set} →
                         (f : S → T) → (xss : Vec (Vec S m) n) →
  map (map f) (transpose xss) ≡ transpose (map (map f) xss)
mapMapFBeforeTranspose {zero} {m} f [] = fill-[]₁ m f
mapMapFBeforeTranspose {suc n} {zero} f xss = refl
mapMapFBeforeTranspose {suc n} {suc m} f ((x :: xs) :: xss) =
  begin
    (f x :: map f (map head xss)) :: map (map f) (transpose (xs :: map tail xss))
  ≡⟨ cong ((f x :: map f (map head xss)) ::_)
      (mapMapFBeforeTranspose f (xs :: map tail xss)) ⟩
    (f x :: map f (map head xss)) ::
    transpose (map f xs :: map (map f) (map tail xss))
  ≡⟨ cong (_:: transpose (map f xs :: map (map f) (map tail xss)))
      (cong (f x ::_) (map-head f xss)) ⟩
```

```
    (f x :: map head (map (map f) xss)) ::
    transpose (map f xs :: map (map f) (map tail xss))
  ≡⟨ cong (λ yss → (f x :: map head (map (map f) xss)) ::
      transpose (map f xs :: yss)) (map-tail f xss) ⟩
    refl
```

The first base case is proven with lemma fill-[]$_1$:

```
fill-[]₁ : {S T : Set} → (m : ℕ) → (f : S → T) →
           map (map f) (fill m []) ≡ fill m []
fill-[]₁ zero f = refl
fill-[]₁ (suc m) f = cong ([] ::_) (fill-[]₁ m f)
```

which says given a 2D array constructed by filling an array with m copies of empty arrays, if we apply nested `map` operations on that 2D array, the value of that 2D array remains unchanged.

The second base case is trivial. The inductive case is proven by two other lemmas defined as:

```
map-head : {n m : ℕ} → {S T : Set} →
           (f : S → T) → (xss : Vec (Vec S (suc m)) n) →
           map f (map head xss) ≡ map head (map (map f) xss)
map-head f [] = refl
map-head f ((x :: xs) :: xss) = cong (f x ::_) (map-head f xss)

map-tail : {n m : ℕ} → {S T : Set} →
           (f : S → T) → (xss : Vec (Vec S (suc m)) n) →
           map (map f) (map tail xss) ≡ map tail (map (map f) xss)
map-tail f [] = refl
map-tail f ((x :: xs) :: xss) = cong (map f xs ::_) (map-tail f xss)
```

They are both simply proven by induction. The map-head lemma says applying a `map` operation after mapping the `head` operation on a 2D array has the same result as mapping the `head` operation after applying nested `map`s on the 2D array; and map-tail shows the equality between applying nested `map`s before and after mapping a `tail` operation onto a 2D array.

The process of proving the inductive case is applying the rule itself inductively, and then applying map-head and map-tail to complete the proof.

4.2.2 Join-transpose rules

There are four join-transpose rules, each of them changes `join` and `transpose` operations' orders in a composed expression. Formally they are defined as:

$$transpose \circ join \rightarrow map\ join \circ transpose \circ map\ transpose \tag{4.19}$$

$$map\ join \circ transpose \rightarrow transpose \circ join \circ map\ transpose \tag{4.20}$$

$$join \circ map\ transpose \rightarrow transpose \circ map\ join \circ transpose \tag{4.21}$$

$$transpose \circ map\ join \rightarrow join \circ map\ transpose \circ transpose \tag{4.22}$$

These four rules are not symmetric to each other, nevertheless, their patterns are internally related to each other:

- The left-hand side of the rule 4.19 is the same as the first two patterns of the right-hand side of the rule 4.20 while the left-hand side of the rule 4.20 has the same pattern as the first two components of the right-hand side of the rule 4.19;

- the left-hand side of the rule 4.20 matches the second and third components of the right-hand side of the rule 4.21 while the left-hand side of the rule 4.21 is the same as the second and third components of the right-hand-side of the rule 4.20;
- the left-hand side of the rule 4.21 matches the first two components of the rule 4.22 and again the left-hand side of the rule 4.22 is the same as the first two patterns of the right-hand side of the rule 4.21.

Therefore, the strategy of proving these rules is, starting from proving the first rule 4.19, other rules can be sequentially proven based on a previous rule afterwards.

To prove the first join-transpose rule 4.19, several lemmas need to be introduced first, note that proofs are omitted for brevity, but available in the provided source code:

```
fill-[]₂ : (n : ℕ) → {m : ℕ} → {T : Set} → (xs : Vec T zero) →
        fill n xs ≡ map (join {m}) (fill n [])


map-join-[] : {n₁ n₂ m : ℕ} → {T : Set} →
                (xsss : Vec (Vec (Vec T zero) n₁) m) →
                (ysss : Vec (Vec (Vec T zero) n₂) m) →
                map join xsss ≡ map join ysss


map-map-head : {n m o : ℕ} → {T : Set} →
                    (xsss : Vec (Vec (Vec T (suc o)) (suc m)) (suc n)) →
                    map (map head) xsss ≡ map head (map transpose xsss)


map-map-tail : {n m o : ℕ} → {T : Set} →
                  (xsss : Vec (Vec (Vec T (suc o)) (suc m)) (suc n)) →
  map transpose (map (map tail) xsss) ≡ map tail (map transpose xsss)
```

The lemma fill-[]$_2$ is similar to the lemma fill-[]$_1$ introduced in section 4.2.1, stating that mapping a `join` operation onto a 2D array constructed by filling an array with n empty arrays does not change the 2D array's value. The lemma map-join-[] is straightforward, it means given two 3D arrays, both of them have innermost dimensions with size zero and outermost dimensions with size m, although the second dimensions have different sizes, mapping a `join` operation onto them gets the same result. map-map-head and map-map-tail are lemmas for reducing nested mapping `head` or `tail` operations into a single `map` with mapping `transpose` operations.

The proof the first join-transpose rule is constructed by induction on the size of each dimension of the input 3D array:

```
-- The proof of the first join-transpose rule
joinBeforeTranspose : {n m o : ℕ} → {t : Set} → (xsss : Vec (Vec (Vec t o) m) n) →
  transpose (join xsss) ≡ map join (transpose (map transpose xsss))
-- Base cases
joinBeforeTranspose {zero} {m} {o} [] = fill-[]₂ o {m} []
joinBeforeTranspose {suc n} {zero} {o} ([] :: xsss) =
  begin
    transpose (join xsss)
  ≡⟨ joinBeforeTranspose xsss ⟩
    map join (transpose (map transpose xsss))
  ≡⟨ map-join-[] (transpose (map transpose xsss))
      (transpose (map transpose ([] :: xsss))) ⟩
```

```
    refl
joinBeforeTranspose {suc n} {suc m} {zero} (xss :: xsss) = refl
-- Inductive case
joinBeforeTranspose {suc n} {suc m} {suc o} xsss =
  begin
    map head (join xsss) :: transpose (map tail (join xsss))
  ≡⟨ cong₂ (λ x y → x :: transpose y)
      (joinBeforeMapF head xsss) (joinBeforeMapF tail xsss) ⟩
    join (map (map head) xsss) :: transpose (join (map (map tail) xsss))
  ≡⟨ cong (join (map (map head) xsss) ::_)
      (joinBeforeTranspose (map (map tail) xsss)) ⟩
    join (map (map head) xsss) ::
    map join (transpose (map transpose (map (map tail) xsss)))
  ≡⟨ cong₂ (λ x y → join x :: map join (transpose y))
      (map-map-head xsss) (map-map-tail xsss) ⟩
    refl
```

The first base case can be proven using the lemma fill-[]$_2$, the second base case can be proven with induction on itself followed by the lemma map-join-[], and the third base case is trivial. There are three steps in the proof of the inductive case, firstly, applying the first join movement rule `joinBeforeMapF` (definition 4.15) which is proven in section 4.2.1 twice to the 3D array *xsss*, where *f* in the join movement rule is substituted by `head` and `tail` respectively. The second step is to apply `joinBeforeTranspose` by induction. Lastly, lemmas map-map-head and map-map-tail are applied to complete the proof.

Base on the proven rule `joinBeforeTranspose`, we are able to prove the second join-transpose rule 4.20. According to the pattern introduced at the beginning of this section, the left-hand side of `joinBeforeTranspose` matches the first two components of the right-hand side of the rule 4.20; and the left-hand side of the rule 4.20 matches the first two components of left-hand side of `joinBeforeTranspose`, it is easier for us to prove the symmetric definition of this rule and since equality is a symmetric relation, this rule can hence be proven correct.

The proof of the symmetric definition is constructed as:

```
-- The symmetric lemma for the second join-transpose rule
sym-lem₁ : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
           transpose (join (map transpose xsss)) ≡ map join (transpose xsss)
sym-lem₁ xsss =
  begin
    transpose (join (map transpose xsss))
  ≡⟨ joinBeforeTranspose (map transpose xsss) ⟩
    map join (transpose (map transpose (map transpose xsss)))
  ≡⟨ cong (λ y → map join (transpose y)) (double-map-transpose xsss) ⟩
    refl
```

It is simply proven by two steps, firstly applying the `joinBeforeTranspose` rule to justify the left-hand side of the equation into an expression matches the first two components of the right-hand side of the equation, and then cancelling the consecutive mappings of `transpose` operations by applying the lemma double-map-transpose:

```
double-map-transpose : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                       map transpose (map transpose xsss) ≡ xsss
```

It is similar to the third identity rule 4.4, the difference is that instead of cancelling consecutive `transpose` operations, it cancels consecutive mappings of `transpose`.

Then we can prove the rule 4.20 by stating the equality relation proven in sym-lem$_1$ is symmetric:

```
-- The symmetric lemma for the second join-transpose rule
transposeBeforeMapJoin : {n m o : ℕ} → {T : Set} →
                         (xsss : Vec (Vec (Vec T o) m) n) →
  map join (transpose xsss) ≡ transpose (join (map transpose xsss))
transposeBeforeMapJoin xsss = sym (sym-lem₁ xsss)
```

Note that sym is the function saying an propositional equality relation is symmetric in Agda.

The third 4.21 and fourth 4.22 rules can be proven using the same strategy: developing a proof of the symmetric definition base on a previous proven rule, and then justifying the correctness of the rule by the symmetric property of the equality relation.

4.2.3 Split-transpose rules

Three split-transpose rules introduced in this section are the rules moving `split` and transpose operations in expression, formally they are defined as:

$$split\ n \circ transpose \longrightarrow map\ transpose \circ transpose \circ map\ (split\ n) \tag{4.23}$$

$$transpose \circ map\ (split\ n) \longrightarrow map\ transpose \circ split\ n \circ transpose \tag{4.24}$$

$$map\ (split\ n) \circ transpose \longrightarrow transpose \circ map\ transpose \circ split\ n \tag{4.25}$$

Similar to the join-transpose rules, the is some regularity in their patterns:

- The left-hand side of the rule 4.23 matches the second and third components of the rule 4.24, and vice versa;
- the two components on the left-hand side of the rule 4.24 have reversed orders of them on the left-hand side of the rule 4.25, and on the right-hand side, *transpose* is the first function to be applied in 4.24 but the last function to be applied in the rule 4.25.

It indicates that the strategy to prove these three rules is to prove the rule 4.23 first, and then to develop the proof of the second rule 4.24 base on the first rule, lastly to prove the third rule 4.25 using the second rule.

The proof of the first split-transpose rule 4.23 is shown below:

```
-- The proof of the first split-transpose rule
transposeBeforeSplit : {m p q : ℕ} → {T : Set} → (n : ℕ) →
                       (xsss : Vec (Vec (Vec T p) (m * n)) q) →
  split n {m} (transpose xsss) ≡ map transpose (transpose (map (split n {m}) xsss))
transposeBeforeSplit {zero} n [] = refl
transposeBeforeSplit {zero} n (xss :: xsss) = refl
transposeBeforeSplit {suc m} n xsss =
  begin
    take n (transpose xsss) :: split n (drop n (transpose xsss))
  ≡⟨ cong₂ (λ x y → x :: split n y) (take-transpose n xsss) (drop-transpose n xsss) ⟩
    transpose (map (take n) xsss) :: split n (transpose (map (drop n) xsss))
  ≡⟨ cong (transpose (map (take n) xsss) ::_)
    (transposeBeforeSplit n (map (drop n) xsss)) ⟩
    transpose (map (take n) xsss) ::
    map transpose (transpose (map (split n) (map (drop n) xsss)))
  ≡⟨ sym (decompose-lem₁ n xsss) ⟩
    refl
```

The two base cases are trivial, we focus on discussing about the proof of the inductive case. Proving the inductive case requires three steps, the first step is applying two lemmas take-transpose and drop-transpose to change the orders of `take`, `drop` and `transpose` in the expression, allowing us to inductively apply `transposeBeforeSplit` in the second step, the two lemmas are straightforward:

```
take-transpose : (n : ℕ) → {m p : ℕ} →
                 {T : Set} → (xs : Vec (Vec T (n + m)) p) →
                 take n {m} (transpose xs) ≡ transpose (map (take n {m}) xs)

drop-transpose : (n : ℕ) → {m p : ℕ} →
                 {T : Set} → (xs : Vec (Vec T (n + m)) p) →
                 drop n {m} (transpose xs) ≡ transpose (map (drop n {m}) xs)
```

Basically take-transpose says taking n elements from the result of transposing a 2D array equals to mapping the `take` operation before transposing the 2D array; drop-transpose follows the logic but replaces `take` with `drop`.

After the first two step, the left-hand side of the equation is written into:

```
transpose (map (take n) xsss) ::
map transpose (transpose (map (split n) (map (drop n) xsss)))
```

We would like to prove the right-hand side of the equation can be decompose into this expression, by introducing the lemma decompose-lem_1:

```
decompose-lem₁ : {m p q : ℕ} → {T : Set} → (n : ℕ) →
                 (xsss : Vec (Vec (Vec T p) (suc m * n)) q) →
  map transpose (transpose (map (split n {suc m}) xsss)) ≡
  transpose (map (take n {m * n}) xsss) ::
  map transpose (transpose (map (split n) (map (drop n) xsss)))
```

It states the right-hand side of the equation is equal to the above expression. We plug the lemma with its symmetric property into the third step of the proof to complete the reasoning chain.

The idea of proving the second split-transpose rule is the same as proving the second join-transpose rule 4.24 introduced in section 4.2.2, which is proving the symmetric definition using a previously proven rule and then justifying the rule by the symmetric property of equality relations. The symmetric definition is proven as:

```
-- The symmetric lemma for the second split-transpose rule
sym-lem₄ : {m p q : ℕ} → {T : Set} → (n : ℕ) →
           (xsss : Vec (Vec (Vec T p) (m * n)) q) →
  map transpose (split n (transpose xsss)) ≡ transpose (map (split n {m}) xsss)
sym-lem₄ n xsss =
  begin
    map transpose (split n (transpose xsss))
  ≡⟨ cong (map transpose) (transposeBeforeSplit n xsss) ⟩
    map transpose (map transpose (transpose (map (split n) xsss)))
  ≡⟨ double-map-transpose (transpose (map (split n) xsss)) ⟩
    refl
```

The proof is clearly constructed with two steps: applying the first split-transpose rule and applying the lemma double-map-transpose defined in section 4.2.2. Then the second split-transpose rule can be justified by stating the equality proven in sym-lem_4 is symmetric:

```
-- The proof of the second split-transpose rule
mapSplitBeforeTranspose : {m p q : ℕ} → {T : Set} → (n : ℕ) →
                          (xsss : Vec (Vec (Vec T p) (m * n)) q) →
  transpose (map (split n {m}) xsss) ≡ map transpose (split n (transpose xsss))
mapSplitBeforeTranspose n xsss = sym (sym-lem₄ n xsss)
```

Proving the third split-transpose rule 4.25 is more tricky, although we would like to make use of the second split-transpose rule to prove it, its components do not exactly match the patterns in the second split-transpose rule. Thus, we make use of the third identity rule 4.4 introduced in section 4.1.2 to create and cancel consecutive `transpose` operations for pattern matching:

```
-- The proof of the third split-transpose rule
transposeBeforeMapSplit : {m p q : ℕ} → {T : Set} → (n : ℕ) →
                          (xsss : Vec (Vec (Vec T p) q) (m * n)) →
  map (split n {m}) (transpose xsss) ≡ transpose (map transpose (split n xsss))
transposeBeforeMapSplit n xsss =
  begin
    map (split n) (transpose xsss)
  ≡⟨ sym (identity₃ (map (split n) (transpose xsss))) ⟩
    transpose (transpose (map (split n) (transpose xsss)))
  ≡⟨ cong transpose (mapSplitBeforeTranspose n (transpose xsss)) ⟩
    transpose (map transpose (split n (transpose (transpose xsss))))
  ≡⟨ cong (λ y → transpose (map transpose (split n y))) (identity₃ xsss) ⟩
    refl
```

The first step of the proof, we apply the third identity rule symmetrically to introduce a pair of `transpose` operations, in order to write the expression into the pattern which allows the second split-transpose rule `mapSplitBeforeTranspose` to be applied in the second step. Finally the third identity rule is applied again to cancel the innermost pair of `transpose` operations, completing the proof.

4.2.4 Slide-transpose rules

Since `split` can be viewed as a special case of `slide`, we can generalise the split-transpose rules into slide-transpose rules:

$$slide\ n \circ transpose \longrightarrow map\ transpose \circ transpose \circ map\ (slide\ n) \tag{4.26}$$

$$transpose \circ map\ (slide\ n) \longrightarrow map\ transpose \circ slide\ n \circ transpose \tag{4.27}$$

$$map\ (slide\ n) \circ transpose \longrightarrow transpose \circ map\ transpose \circ slide\ n \tag{4.28}$$

The strategy of proving these three rules is exactly the same as the strategy used for proving three split-transpose rules discussed in section 4.2.3: proving the rule 4.26 first, and then constructing the proof of the second rule 4.27 using it, finally proving the third rule 4.28 using the second rule 4.27.

The steps of proving the first slide-transpose rule 4.26 is similar to proving the first split-transpose rule, however some new lemmas are required:

```
-- The proof of the first slide-transpose rule
transposeBeforeSlide : {n m o : ℕ} → {T : Set} → (sz sp : ℕ) →
                       (xsss : Vec (Vec (Vec T o) (sz + n * (suc sp))) m) →
  slide {n} sz sp (transpose xsss) ≡
  map transpose (transpose (map (slide {n} sz sp) xsss))
```

```
-- Base case 1
transposeBeforeSlide {zero} sz sp [] = refl
-- Base case 2
transposeBeforeSlide {zero} sz sp (xss :: xsss) =
  cong (λ y → transpose (xss :: y) :: []) (sym (map-head-id xsss))
-- Indctive case
transposeBeforeSlide {suc n} sz sp xsss =
  let ys = map (cast (slide-lem n sz sp)) xsss in
  begin
    take sz (transpose xsss) :: slide sz sp (drop (suc sp) (cast _ (transpose xsss)))
  ≡⟨ cong (λ y → take sz (transpose xsss) :: slide sz sp (drop (suc sp) y))
      (transpose-cast (slide-lem n sz sp) xsss) ⟩
    take sz (transpose xsss) :: slide sz sp (drop (suc sp) (transpose ys))
  ≡⟨ cong₂ (λ x y → x :: slide sz sp y)
      (take-transpose sz xsss) (drop-transpose (suc sp) ys) ⟩
    transpose (map (take sz) xsss) :: slide sz sp (transpose (map (drop (suc sp)) ys))
  ≡⟨ cong (transpose (map (take sz) xsss) ::_)
      (transposeBeforeSlide sz sp (map (drop (suc sp)) ys)) ⟩
    transpose (map (take sz) xsss) ::
    map transpose (transpose (map (slide sz sp) (map (drop (suc sp)) ys)))
  ≡⟨ sym (decompose-lem₂ sz sp xsss) ⟩
    refl
```

The second base case requires a lemma map-head-id, which is defined as:

```
map-head-id : {n m : ℕ} → {T : Set} → (xss : Vec (Vec T m) n) →
                map head (map (λ xs → xs :: []) xss) ≡ xss
```

where *xs* has type (Vec *T m*). It basically says constructing each inner element with an empty array and then taking the head of each inner element does not change the resulting value. Obviously, the result of (head *xs* :: []) is *xs*.

The process of proving the inductive case is similar to proving the inductive case of the first split-transpose rule, but need to firstly apply the lemma transpose-cast to change the order of `cast` and `transpose` in the expression for adjusting the size of the input 3D array to satisfy following pattern matching. The transpose-cast is defined as:

```
transpose-cast : {m m₁ n : ℕ} → {T : Set} →
                   .(eq : m ≡ m₁) → (xs : Vec (Vec T m) n) →
                   cast eq (transpose xs) ≡ transpose (map (cast eq) xs)
```

which indicates applying `transpose` before `cast` equals to applying `transpose` after mapping `cast` operations onto the input.

The second step is again applying lemmas take-transpose and drop-transpose introduced in section 4.2.3, to change orders of the operations so that the `transposeBeforeSlide` can be applied inductively. The resulting expression is:

```
transpose (map (take sz) xsss) ::
map transpose (transpose (map (slide sz sp) (map (drop (suc sp)) ys)))
```

Note that in this expression *ys* is the name binding of (map (cast (slide-lem *n sz sp*)) *xsss*).

Similarly, we need to decompose the expression on the right-hand side of the equation into the expression above, using the lemma decompose-lem₂:

```
decompose-lem₂ : {n m o : ℕ} → {T : Set} → (sz sp : ℕ) →
                 (xsss : Vec (Vec (Vec T o) (suc (sz + (sp + n * suc sp)))) m) →
  map transpose (transpose (map (λ xs → take sz {suc (sp + n * suc sp)} xs ::
  slide {n} sz sp (drop (suc sp) (cast (slide-lem n sz sp) xs))) xsss)) ≡
  transpose (map (take sz {suc (sp + n * suc sp)}) xsss) ::
  map transpose (transpose (map (slide sz sp) (map (drop (suc sp))
  (map (cast (slide-lem n sz sp)) xsss))))
```

By applying this lemma symmetrically in the last step, the proof can be completed.

The proofs of the second 4.27 and third 4.28 slide-transpose rules are constructed in the same ways respectively as the second and third split-transpose rules, since their patterns are the same but replacing `split` with `slide` in definitions of slide-transpose rules.

4.2.5 Proving `join` is associative using heterogeneous equality

The last pair of rules in this section are stating the `join` primitive is associative, which means given a 3D array, joining the two outer dimensions first and then joining the result with the innermost dimension is equal to joining the two inner dimensions first and then joining the result with the outermost dimension. Formally they are defined as:

$$join \circ join \longrightarrow join \circ map\ join \tag{4.29}$$

$$join \circ map\ join \longrightarrow join \circ join \tag{4.30}$$

The definitions are rather straightforward, however, if we try to define the equality relation of the rule 4.29 in Agda using propositional equality as we have done for all previous proofs:

```
joinBeforeJoin : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                 join (join xsss) ≡ join (map join xsss)
```

the compiler will raise an error indicating the type of the right-hand side does not match the type of the left-hand side of the equation:

```
n != n * m of type ℕ
when checking that the inferred type of an application
  Vec T (n * (m * o))
matches the expected type
  Vec T (n * m * o)
```

The left-hand side of the equation has type (Vec T (n * m * o)) and the right-hand side of the equation has type (Vec T (n * (m * o))). Although it is intuitive that multiplication is associative thus these two array has the same size, in Agda, they are two different types. Since propositional equality has the premise that the two sides of the equation must have the same type, this rule cannot be proven using propositional equality.

However, (join (join *xsss*)) and (join (map join *xsss*)) produces the same value, they have some kind of equality relation despite their types are different in Agda. We would like to form this equality relation as heterogeneous equality.

Heterogeneous equality is an equality relation for two expressions with different types, which is less restrictive than propositional equality. It has been shown useful for reasoning about the soundness of algorithmic equality, normalisation, consistency, etc., by Abel (2011). The crucial point of forming heterogeneous equality is reasoning about why the types are equal (Altenkirch

et al. 2007). For example, we can reason about the equality of type (Vec T (n * m * o)) and (Vec T (n * (m * o))) by the equality of their sizes; and the equality of their sizes can be justified since multiplication is associative.

The heterogeneous equality is denoted as ≅ in Agda, this equality relation is also reflexive, symmetric and transitive. Similar to propositional equality, a reasoning chain can be form with begin, ■ and ≅⟨⟩. To distinguish from propositional equality reasoning chains, we rename begin and ■ to hbegin and h■.

The congruence function in propositional equality is important for reasoning about given an equality relation $x \equiv y$ and a function f we can conclude $f\ x \equiv f\ y$. To facilitate developing proofs, a similar congruence function for heterogeneous equality is also introduced:

```
hcong′ : {α β γ : Level} {I : Set α} {i j : I}
         → (A : I → Set β) {B : {k : I} → A k → Set γ} {x : A i} {y : A j}
         → i ≡ j
         → (f : {k : I} → (x : A k) → B x)
         → x ≅ y
         → f x ≅ f y
hcong′ _ refl _ Heq.refl = Heq.refl
```

Where A is a constructor function, taking in an argument with type I in the universe Set α and constructing a type in the universe Set β; B is another constructor function which takes in an argument constructed by A and produces a type in the universe Set γ. Given values i and j with the same type I, x and y are values constructed by A using i and j respectively. f is a function using B to construct a value with an argument produced by the constructor A. If there exists propositional equality between i and j as well as heterogeneous equality between x and y, we can reach the conclusion $f\ x$ and $f\ y$ are heterogeneously equal.

Note that we defined this function as hcong′ to distinguish from the congruence function provided in the standard library. That congruence function does not stress on the propositional equality reasoning in the type constructor function, thus does not satisfy the important requirement on constructing heterogeneous equality: reasoning about why two types are equal.

The justification of the rule 4.29 can be expressed using a heterogeneous equality reasoning chain in Agda, the proof is shown below:

```
-- The proof of join is associative
joinBeforeJoin : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                 join (join xsss) ≅ join (map join xsss)
joinBeforeJoin [] = Heq.refl
joinBeforeJoin {suc n} {m} {o} {T} (xss :: xsss) =
  hbegin
    join (xss ++ join xsss)
  ≅⟨ join-++ xss (join xsss) ⟩
    join xss ++ join (join xsss)
  ≅⟨ hcong′ (Vec T) (*-assoc n m o) (λ y → join xss ++ y) (joinBeforeJoin xsss) ⟩
    join xss ++ join (map join xsss)
  h■
```

The base case can be simply justified since the heterogeneous equality is reflexive. The inductive case is proven using a lemma join-++ and the induction on itself with the heterogeneous congruence relation defined above. Note that when we apply the heterogeneous congruence in the proof, (Vec T) is the constructor function A, the associativity of multiplication *-assoc is the propositional equality relation, the lambda function ($\lambda\ y \rightarrow$ join xss ++ y) is the function f and the induction on $xsss$, i.e., joinBeforeJoin $xsss$, is the required heterogeneous equality relation.

The lemma join-++ can be viewed as distribute `join` over an array concatenation operation, which is defined as:

```
join-++ : {n m o : ℕ} → {T : Set} →
          (xs₁ : Vec (Vec T o) n) → (xs₂ : Vec (Vec T o) m) →
          join (xs₁ ++ xs₂) ≅ join xs₁ ++ join xs₂
join-++ [] xs₂ = Heq.refl
join-++ {suc n} {m} {o} {T} (xs :: xs₁) xs₂ =
  hbegin
    xs ++ join (xs₁ ++ xs₂)
  ≅⟨ hcong' (Vec T) (*-distrib^r-++ o n m) (λ y → xs ++ y) (join-++ xs₁ xs₂) ⟩
    xs ++ join xs₁ ++ join xs₂
  ≅⟨ hsym (++-assoc xs (join xs₁) (join xs₂)) ⟩
    (xs ++ join xs₁) ++ join xs₂
  h■
```

It states that concatenating two 2D arrays together and then joining the resulting 2D array have the same result as joining these two 2D arrays separately and then concatenating the results together. The left-hand side and right-hand side are heterogeneously equal since the natural number multiplication is distributive over addition. It is proven based on the fact the array concatenation operation is associative:

```
-- The proof of array concatenation is associative
++-assoc : {n m o : ℕ} → {T : Set} →
           (xs : Vec T n) → (ys : Vec T m) → (zs : Vec T o) →
           (xs ++ ys) ++ zs ≅ xs ++ (ys ++ zs)
++-assoc [] ys zs = Heq.refl
++-assoc {suc n} {m} {o} {T} (x :: xs) ys zs =
          hcong' (Vec T) (+-assoc n m o) (λ l → x :: l) (++-assoc xs ys zs)
```

Again the reasoning chain is established with heterogeneous equations, the equality of the sizes of left-hand side and right-hand side of the of the definition can be proven equal since the natural number addition is associative.

The second rule 4.30 can be then proven correct since heterogeneous equality is a symmetric relation:

```
-- Heterogeneous equality is symmetric
mapJoinBeforeJoin : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                    join (map join xsss) ≅ join (join xsss)
mapJoinBeforeJoin xsss = hsym (joinBeforeJoin xsss)
```

From the above proofs, it can be pointed out that array concatenation can be viewed as an addition operation on sizes of arrays, and the `join` operation can be viewed as the multiplication on the sizes of two dimensions of an array. and these two operations have some properties same as natural number addition and multiplication, i.e., being associative and distributive.

4.2.6 Summary

This this section, we first introduce a set of movement rules that changes the orders of `map` and other primitive operations in expressions (section 4.2.2), they can be broken down into lemmas when developing proofs. Then we introduces three set of rules that move `transpose` around primitives `join`, `split`, and `slide` in section 4.2.2, 4.2.3 and 4.2.4. Rules in each set are internally

related, thus if we prove one in each set to be correct, the proofs of other rules can be sequentially developed using proven ones. Lastly, we introduce an equality relation justifying expressions with different types, i.e., heterogeneous equality in section 4.2.5, and make use of it proving the `join` primitive is associative. We further discuss about the similarities between natural number operations and array operations.

4.3 Tiling: a stencil computation rule

In the last section of this chapter, we would like to formalise and verify the tiling rule, which is a stencil computation rule introduced by Hagedorn et al. (2018). It partitions applying `map` to the resulting 2D array produced by `slide`. The general idea is with the consideration of overlapping, firstly applying `slide` to get larger chunks, and then applying `slide` in each chunk with the given *size* and *step*, joining the larger chunks together at the end. Formally it is defined as:

$$map\ f \circ slide\ size\ step \longrightarrow join \circ map\ (\lambda\ tile.\ map\ f \circ (slide\ size\ step\ tile))\ slide\ u\ v \tag{4.31}$$

In the paper, restrictions of choices u and v in rule 4.31 are not specified. The only piece of specified information is $u - v = size - step$. Therefore, firstly we would like to give some restrictions on the choices of u and v.

According to the definition of `slide` in Agda (section 3.2.4), we know that the size of the input array should satisfy $(sz + n * (suc\ sp))$, where sz and sp are the chosen size and step. Hence a general choice of u should be $(sz + n * (suc\ sp))$, allowing the following `slide` to be operated on. Then the value of $(suc\ v)$ can calculate: $((suc\ n) * (suc\ sp))$, ensure the difference between u and $suc\ v$ equals to sz and $(suc\ sp)$. We use *suc* to ensure the value of step to be larger than zero.

The rule is proven in Agda as:

```
-- the proof of the tiling rule
tiling : {n m : ℕ} → {S T : Set} → (sz sp : ℕ) → (f : Vec S sz → Vec T sz) →
         (xs : Vec S (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
  join (map (λ (tile : Vec S (sz + n * (suc sp))) →
  map f (slide {n} sz sp tile)) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs)) ≡
  map f (slide {n + m * (suc n)} sz sp (cast (lem₁ n m sz sp) xs))
tiling {n} {m} {s} {t} sz sp f xs =
  begin
    join (map (λ (tile : Vec s (sz + n * (suc sp))) → map f (slide {n} sz sp tile))
    (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs))
  ≡⟨ cong join (map-λ {n} {m} sz sp f xs) ⟩
    join (map (map f) (map (slide {n} sz sp)
    (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)))
  ≡⟨ mapMapFBeforeJoin f (map (slide {n} sz sp)
      (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)) ⟩
    map f (join (map (slide {n} sz sp) (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)))
  ≡⟨ cong (map f) (slideJoin {n} {m} sz sp xs) ⟩
    refl
```

Note that in the declaration of the equality relation, we apply the lemma lem_1 to adjust the size of the input array *xs* to allow pattern matching for the operation (`slide {n + m * (suc n)} sz sp`) at right-hand side of the equation. lem_1 is defined as:

```
lem₁ : (n m sz sp : ℕ) →
       sz + n * suc sp + m * suc (n + sp + n * sp) ≡ sz + (n + m * suc n) * suc sp
```

The proof is developed using three rules: map-λ for changing the order of `map` and the lambda function in the expression, a movement rule `mapMapFBeforeJoin` (rule 4.16) introduced in section 4.2.1, and the rule `slideJoin`.

Basically the map-λ rule moves the (map f) out of the lambda function as nested `map` operations, allowing the `mapMapFBeforeJoin` to be applied to group the operations into the pattern which satisfies the rule `slideJoin`.

The rule map-λ is defined and proven inductively as below:

```
map-λ : {n m : ℕ} → {S T : Set} → (sz : ℕ) → (sp : ℕ) → (f : Vec S sz → Vec T sz) →
          (xs : Vec S (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
  map (λ (tile : Vec S (sz + n * (suc sp))) →
  map f (slide {n} sz sp tile)) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs) ≡
  map (map f) ((map (λ (tile : Vec S (sz + n * (suc sp))) →
  slide {n} sz sp tile)) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs))
map-λ {n} {zero} sz sp f xs = refl
map-λ {n} {suc m} sz sp f xs =
  begin
    map f (slide sz sp (take (sz + n * suc sp) xs)) ::
    map (λ tile → map f (slide sz sp tile)) (slide (sz + n * suc sp) (n + sp + n * sp)
    (drop (suc (n + sp + n * sp)) (cast _ xs)))
  ≡⟨ cong (map f (slide sz sp (take (sz + n * suc sp) xs)) ::_)
     (map-λ sz sp f (drop (suc (n + sp + n * sp)) (cast _ xs))) ⟩
    refl
```

The `slideJoin` rule is a partition strategy of the slide operation, formally defined as:

$$slide\ size\ step \rightarrow join \circ map\ (\lambda\ tile.\ slide\ size\ step\ tile)\ slide\ u\ v \tag{4.32}$$

Since the order of the application of $map\ f$ in tiling rule 4.31 can be handled by map-λ and the movement rule 4.16, i.e., `mapMapFBeforeJoin`, if we prove this partitioning strategy to be correct, we can conclude this tiling rule is correct.

This rule is originally defined as:

$$slide\ size\ step \rightarrow join \circ (\lambda\ tile.\ map\ (slide\ size\ step\ tile))\ slide\ u\ v$$

which is not correct. When we try to encode it in Agda, the right-hand side does not share the same type of the left-hand side; the orders of the function does not satisfy the pattern matching on the type of the input array. Hence we revise the definition as rule 4.32.

The proof of the `slideJoin` rule is shown below:

```
slideJoin : {n m : ℕ} → {T : Set} → (sz : ℕ) → (sp : ℕ) →
              (xs : Vec T (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
  join (map (λ (tile : Vec T (sz + n * (suc sp))) →
  slide {n} sz sp tile) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs)) ≡
  slide {n + m * (suc n)} sz sp (cast (lem₁ n m sz sp) xs)
slideJoin {n} {zero} sz sp xs =
  begin
    slide sz sp xs ++ []
  ≡⟨ ++-[] (slide sz sp xs) ⟩
    slide sz sp xs
  ≡⟨ cong (slide sz sp) (lem₂ {n} {sz} {sp} xs) ⟩
```

```
    refl
slideJoin {n} {suc m} sz sp xs =
  begin
    slide {n} sz sp (take (sz + n * suc sp) xs) ++
    join (map (slide {n} sz sp) (slide {m} (sz + n * suc sp) (n + sp + n * sp)
    (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs))))
  ≡⟨ cong (slide {n} sz sp (take (sz + n * suc sp) xs) ++_)
     (slideJoin {n} {m} sz sp (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs)) ) ⟩
    slide {n} sz sp (take (sz + n * suc sp) xs) ++
    slide {n + m * suc n} sz sp (cast (lem₁ n m sz sp)
    (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs)))
  ≡⟨ lem₄ {n} {m} sz sp xs ⟩
    refl
```

Again, lem_1 is used in the definition for adjusting the pattern of the size of the input array. Some other lemmas are used in the proof.

In the base case, we first use lemma `++-[]` to justify the concatenation of a given array with an empty array produces the array itself. And then we uses the lem_2 to reason about that the casting operation using lem_1 when the value m is zero does not change the value of the input array, which is defined as:

```
lem₂ : {n sz sp : ℕ} → {T : Set} → (xs : Vec T (sz + n * suc sp)) →
       xs ≡ cast (lem₁ n zero sz sp) xs
lem₂ {zero} {zero} {sp} [] = refl
lem₂ {zero} {suc sz} {sp} (x :: xs) = cong (x ::_) (lem₂ {zero} {sz} {sp} xs)
lem₂ {suc n} {zero} {sp} (x :: xs) = cong (x ::_) (lem₂ {n} {sp} {sp} xs)
lem₂ {suc n} {suc sz} {sp} (x :: xs) = cong (x ::_) (lem₂ {suc n} {sz} {sp} xs)
```

In the inductive case, lem_3 is another lemma for adjusting the size of input array used by `cast`:

```
lem₃ : (n m sz sp : ℕ) →
      suc (sz + n * suc sp + (n + sp + n * sp + m * suc (n + sp + n * sp))) ≡
      suc (n + sp + n * sp + (sz + n * suc sp + m * suc (n + sp + n * sp)))
```

It can be proven by the properties of natural number addition and multiplication. The proof of the inductive case also has two steps. The first step is inductively applying `slideJoin` with the adjusted pattern of the input array's size, in the second step, a lemma lem_4 is applied:

```
postulate lem₄ : {n m : ℕ} → {T : Set} → (sz sp : ℕ) →
                 (xs : Vec T (suc (sz + n * suc sp +
                 (n + sp + n * sp + m * suc (n + sp + n * sp))))) →
                 slide {n} sz sp (take (sz + n * suc sp)
                 {suc (n + sp + n * sp + m * suc (n + sp + n * sp))} xs) ++
                 slide {n + m * suc n} sz sp (cast (lem₁ n m sz sp)
                 (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs)))
                 ≡
                 take sz {suc (sp + (n + (n + m * suc n)) * suc sp)}
                 (cast (lem₁ n (suc m) sz sp) xs) ::
                 slide {n + (n + m * suc n)} sz sp
                 (drop (suc sp) {sz + (n + (n + m * suc n)) * suc sp}
                 (cast (slide-lem (n + (n + m * suc n)) sz sp )
                 (cast (lem₁ n (suc m) sz sp) xs)))
```

The left-hand side of the equation is a concatenation of the results of applying (`slide` *sz sp*) on the array of size *u* taken from the input array and applying (`slide` *sz sp*) on the input array after dropping size (*suc v*); the right-hand side of the equation is the normalised pattern of the (`slide` *sz sp*) operation on the input array by the definition of `slide` in section 3.2.4.

It can be written in a mathematical form that is easier to read:

$$slide\ sz\ sp \circ take\ u\ \texttt{++}\ slide\ sz\ sp \circ drop\ (suc\ v) \longrightarrow slide\ sz\ sp \tag{4.33}$$

We believe the left-hand side is equal to the right-hand side, since the overlap when performing `take` and `drop` is the difference between *u* and *suc v*, which equals to the overlap in the (`slide` *sz sp*), because the difference between *u* and (*suc v*) equals to the difference between *sz* and (*suc sp*) by our definition. However, due to the over complicated `cast` operations for adjusting the sizes of arrays for pattern matching, we have so far not able to develop the proof in Agda of this lemma.

An example for detailed illustration is shown in figure 4.1:

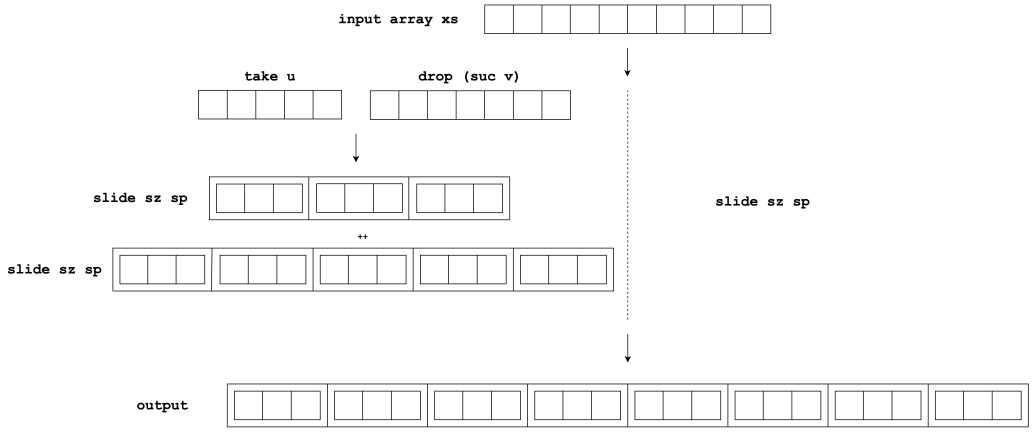

Figure 4.1: *An example for illustrating* lem_4

In this example, $sz = 3$, $suc\ sp = 1$, $u = 5$ and $suc\ v = 3$, it clearly shows that the result we get from the left-hand side of the lemma lem_4 equals to the result of the right-hand side.

4.3.1 Summary

In this section, we provide the verification of an complex stencil computation rule in Agda. Advantages of machine verification are obvious. Firstly, we make use of the dependently typed pattern matching in Agda to introduce general restrictions on the choices of *u* and *v* in the tiling rule 4.31. Comparing to the original piece of information we have, i.e., the difference between *u* and *v* should equal to the difference between *size* and *step*, this accurate formalisation can be helpful to eliminate errors in the implementation. Moreover, this complex rule is broken down into smaller rules and justified step by step in a clear structure.

However, the dependently typed pattern matching in Agda also has its limitations. We have to keep doing equality reasoning on arrays' sizes using the properties of natural number addition and multiplication while developing proofs. In the last lemma lem_4, since casting the input array's size over complicates the declaration, we have so far not been able to develop a proof although in mathematical form it can be justified.

5 Research Outcomes

In this project, some important LIFT data types and a large set of primitive operations have been encoded in Agda, and a set of mechanical proofs have been developed to verified the correctness of three categories of rewrite rules: algorithmic rules, movement rules and the stencil computation rule. In general, most of the rules have been proven correct, the incorrect proof and definition have been revised, and inaccurate definitions of arguments' patterns have been clarified. In general, we find machine verification is helpful for formalising and verifying these rewrite rules for optimising the efficiency of array and multi-dimensional array operations.

Induction is the core of the construction of semantics and proofs. It is effective for giving clear structural definitions of data types and primitives as well as constructing reasoning chains in proofs. Agda is helpful for formalising semantics and proofs following proposition-as-type interpretation since it is dependently typed, however this feature also introduces complexities in pattern matching. Some strategies are introduced for resolving these complexities, such as casting patterns at constructor level with equality reasoning with `cast` defined in section 3.2.4; using `REWRITE` to increase the flexibility of pattern matching while ensuring the soundness of the type system (section 3.2.5); and proving the equality between two expression with different types with heterogeneous equality (section 4.2.5).

Some strategies of constructing proofs are also discussed. A complex rule can be broken down into smaller rules, for example, the tiling rule 4.31 is proven using two rules moving the `map` f operation into suitable order, and the `slideJoin` rule to provide the algorithmic justification. Effective lemmas can be defined according to the definitions of primitives, for example, reasoning on expressions containing `split` can always be proven using lemmas specifying properties of `take` and `drop` since `split` is defined using them (3.2). Also, as the proofs discussed in section 4.2.2, section 4.2.3 and section 4.2.4, some rules are internally related to each other, when one of them is proven, the verification of others can be developed based on the proven ones.

6 Conclusion and Future Work

Effective mechanical verification in Agda is developed for justifying the correctness of the rewrite rules in LIFT. Most of the rewrite rules are proven to be correct while some incorrect or inaccurate rules are adjusted and clarified.

We give brief introduction in chapter 1 about the motivation and aims of the project, and provide background information of LIFT, Agda as well as the computation model Curry-Howard Correspondence in chapter 2. In chapter 3, encoding LIFT's data types and primitives into operational semantics in Agda is introduced. In chapter 4, we discuss in detail about the development of verification for algorithmic rules, movement rules and the stencil computation rule using proof by induction. The outcomes together with what we have learned in this project are discussed in chapter 5.

In current stage, we are not able to prove the last lemma used in proving the tiling rule (4.31) in section 4.3 due to the over-complicated pattern matching on the sizes of arrays. It would be interesting for us to research on strategies to simplify these tedious pattern matching in complex multi-dimensional array operations under dependently typed theorem provers like Agda. The `REWRITE` feature discussed in section 3.2.5 can be useful, however its usage is very limited since it can damage the confluence of the type system and affect the termination of programs.

We would also like to explore the restrictions and justify the correctness of applying similar tiling rules in 2D arrays, with stencil computation primitives for 2D arrays, e.g., $\texttt{slide}_2$ and $\texttt{padCst}_2$, and potentially generalise the verification on rules defined for n-dimensional arrays.

6 Bibliography

A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *International Conference on Foundations of Software Science and Computational Structures*, pages 57--71. Springer, 2011.

Agda Developer Team. Introduction to universes. URL `https://agda.readthedocs.io/en/latest/language/universe-levels.html`. Accessed 2 Apr. 2020.

Agda Developer Team. The agda standard library, 2020. URL `https://github.com/agda/agda-stdlib`. Accessed 2 Apr. 2020.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57--68, 2007.

R. Atkey, M. Steuwer, S. Lindley, and C. Dubach. Strategy preserving compilation for parallel functional code. *CoRR*, abs/1710.08332, 2017.

J. Cockx, N. Tabareau, and T. Winterhalter. How to tame your rewrite rules. *Types for Proofs and Programs, TYPES*, 2019.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

B. Hagdorn, M. Steuwer, R. Fu, and J. Lenfer. ELEVATE, 2020. URL `https://github.com/elevate-lang/elevate/tree/master/src/main/scala/elevate/rise`. Accessed 2 Apr. 2020.

B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 100--112, 2018.

W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479--490, 1980.

M. Steuwer. *Improving programmability and performance portability on many-core processors*. PhD thesis, University of Münster, 2015.

M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. *ACM SIGPLAN Notices*, 50(9):205--217, 2015.

N. Ulf, A. D. Nils, and A. Andreas. Agda. URL `https://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed 2 Apr. 2020.

P. Wadler. Propositions as types. *Communications of the ACM*, 58(12):75--84, 2015.