1 Introduction

The Discrete Fourier Transform (DFT) is a staple operation within Computer Science, Physics, and other fields with many applications. Fast Fourier Transforms are implementations of the DFT with improved performance characteristics. The FFT reduces the complexity of the DFT from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. This has many useful applications and enables the reduction in complexity for algorithms which would otherwise be bounded in time complexity by use of the DFT.

Most current implementations, such as FFTW[1], take the form of large libraries written in low-level languages. A key component of these libraries is the use of multiple implementations of the same algorithm, with each implementation (or kernel) containing optimisations suited towards specific input sizes and hardware profiles. When the user wants to compute the result of a Fourier Transform, the library uses the input size, and some heuristics on the users hardware, to construct a plan. This plan describes the decomposition of a given input, allowing FFTW to utilise a sequence of optimal kernels to compute the result.

The large number of kernels makes it very challenging to formally verify that a given FFT library provides results equal to those given by the DFT. This is because to do so would involve analysing the low-level implementation of each kernel, individually, and proving that it gives the same result as the naïve DFT for all possible inputs. An alternate approach is as follows. Instead of analysing existing code to confirm its correctness, we can create a single specification of the FFT such that it can be instantiated to any kernel, giving us a usable kernel and formal proof that said kernel computes the expected values.

To introduce formal guarantees, a theorem prover is needed, Agda is one such theorem prover which is based on dependent types. This basis on dependent types allows for arbitrary properties to be attached to programs as types. These properties allow us to describe the correctness of a given algorithm as a type, meaning that a given program cannot type check, and thus cannot run, if any of the correctness properties do not hold. This allows for strong guarantees to be placed on a program. This paper discusses the use of Agda to create a general case implementation of the FFT which is then proven to always compute the same value as the naïve implementation of the DFT.

Such an implementation would allow for future research in Agda to make use of the FFT in definitions, before substituting it with the DFT when it comes to generating proofs. This would be useful for research into algorithms which utilise the FFT, such as efficient polynomial multiplication. Such an implementation would also allow for future generation of low-level, efficient kernels, with a formally verified basis.

2 Background

2.1 Fourier Transforms

Described as "perhaps the most ubiquitous algorithm in use today" [2], Fourier Transforms are mathematical operations which transform functions between the time domain and the frequency domain. Fourier Transforms, and derivatives of, receive their name from the French mathematician and physicist Jean-Baptiste-Joseph Fourier who proposed in his 1822 [3] treatise that any given function can be represented as a harmonic series [4]. While bearing Fourier's name, some early forms of the Discrete Fourier Transform (DFT), a Fourier Transform which works on evenly spaced samples of a function, can be found before Fourier's time. As discussed by Heideman and Johnson in "Gauss and the History of the Fast Fourier Transform" [5], the earliest known example of this can be found in work published by Alexis-Claude Clairaut in 1754 [6]. Clairaut defined a variation of the DFT which exclusively used what we now refer to as the cosine component, thus restricting the input domain to the set of even functions[1][5]. Carl Friedrich Gauss extended Clairaut's definition to make use of both cosine and sine components, removing the need for the input domain to be restricted to the set of even functions and allowing for the analysis of any periodic function [9][5]. This definition was published posthumously in 1866, however, it is believed that it was originally written in 1805 [5].

We can use the historical definitions discussed above to create our modern definition for the DFT as follows. Given an input sequence $x = (x_0, x_1, \ldots, x_{n-1})$, where $x_i \in \mathbb{C}$, our transformed sequence $X = (X_0, X_1, \ldots, X_{n-1})$, where $X_i \in \mathbb{C}$, is given as follows.

$$X_j = \sum_{k=0}^{N-1} x_k \omega_N^{kj} \tag{1}$$

$$\text{where } \omega_N = e^{-\frac{2\pi i}{N}} = \cos\left(\frac{2\pi}{N}\right) - i\sin\left(\frac{2\pi}{N}\right) \tag{2}$$

The DFT Eq. 1 has applications in a variety of fields, such as digital signal processing [10]. When implemented naïvely, however, it has poor performance scaling, requiring "$\mathcal{O}(n^2)$ complex operations" [11]. Methods to reduce the number of complex operations required when computing the DFT were first investigated by Gauss in his 1805 treatise such that the "tediousness of mechanical calculations" [9] could be reduced [5]. In part due to his lack of research into the complexity scaling factor of his method, Gauss's research into how computation complexity could be reduced was not widely recognised until 1977 when H. H. Goldstine highlighted Gauss's research in an article for the Journal of Applied Mathematics and Mechanics [5][12]. While the DFT continued to be of great use to mathematicians through the 20th century, and with Gauss's work on complexity remaining hidden, some attempts (such as those by Danielson and Lanczos [13] and by Good [14]) were made to create Fast Fourier Transform algorithms (FFT algorithms) which could reduce the complexity of computation to $\mathcal{O}(n \log n)$. These algorithms, however, where only applicable to a

[1]The term "even function" refers to the set of functions $f(x)$ such that $f(-x) = f(x)$, that is to say, the set of functions which are symmetric over the y-axis. [7][8]

subset of the domain [14], succeeded only in reducing the constant on $\mathcal{O}(n^2)$, or did not directly perform the computational complexity [13].

In 1965 James William Cooley and John Tukey succeeded in discovering an FFT algorithm through the inadvertent reinvention of Gauss's algorithm for fast computation of the DFT; This would henceforth be known as the Cooley-Tukey FFT Algorithm [15][5]. This FFT Algorithm allows for a given DFT to be computed with $\mathcal{O}(n \log n)$ complex operations through recursive splitting of the input [15]. Although other FFT Algorithms were discovered before and after the Cooley-Tukey FFT, it is commonly considered to be "the most important FFT" [1]. This is because this improvement in time complexity allowed algorithms with time complexity previously bounded by use of the DFT, to reduce this complexity to, at the lowest, $\mathcal{O}(n^2)$. In the example of polynomial multiplication, this allowed for the computation to be moved into the frequency domain, reducing the time complexity from $\Theta(n^2)$ to $\Theta(n \log n)$ [16].

The Cooley-Tukey FFT can be derived from the DFT Eq. 1 by splitting any non-prime input n into the composite $n = r_1 r_2$ and expressing the indices k and j as follows.

$$\begin{aligned} j &= j_1 r_1 + j_0 & \qquad k &= k_1 r_2 + k_0 \\ \text{where } \; j_0 &= (0, 1, \ldots, r_1 - 1) & \text{where } \; k_0 &= (0, 1, \ldots, r_2 - 1) \\ j_1 &= (0, 1, \ldots, r_2 - 1) & k_1 &= (0, 1, \ldots, r_1 - 1) \end{aligned} \tag{3}$$

Eq. 1 can then be arranged to take the following form.

$$X_{j_1 r_1 + j_0} = \sum_{k_0=0}^{r_2-1} \left[\left(\sum_{k_1=0}^{r_1-1} x_{k_1 r_2 + k_0} \omega_{r_1}^{k_1 j_0} \right) \omega_{r_1 r_2}^{k_0 j_1} \right] \omega_{r_2}^{k_0 j_1} \tag{4}$$

When written in this form our recursive step, and thus the core idea of the Cooley-Tukey FFT, be easily observed by noting that the inner sum takes the form of a DFT of length r_1.

2.2 Agda

Agda[2] is a functional programming language which implements Martin-Löf Type Theory [17][18]. Martin-Löf type theory provides the definition of, and Agda allows for the construction of, dependent types [17]. These types allow for the definition of invariant properties which are checked at compile time [17]. As well as making a variety of common errors, such as out-of-bound indexing, unreachable, invariance properties can be used to guarantee functional properties [17]. When evidence that properties hold is non trivial, proofs that the properties hold must be provided. This allows for strong guarantees to be formed on any program defined in Agda. These proofs allow systems to be built which are provably correct allowing for a high confidence in their reliability [17].

Agda is not the only such proof assistant, and there are several other successful, dependently typed, languages which are similar to Agda, including Rocq [19], Lean [20]

[2]Reference to "Agda" throughout this report will always refer to version 2 unless explicitly stated otherwise

and Idris [21]. Rocq takes an especially interesting approach, and considers programs and proofs separately, allowing for the complexity of proofs - or as it refers to them, tactics - to be somewhat hidden once they are proven to hold [19]. Because of the similarities between these language, work in this project can be translated into any of these proof assistants, benefiting each respective community.

2.3 Related work

FFTW [1] is a `C` code library which is generally accepted within academia and industry as the fastest method with which the FFT can be correctly computed [22]. It achieves this title by implementing its own "special-purpose compiler" [22], `genfft`, this compiler accepts the size of the transform as input and outputs a kernel - a `c` code implementations of some known algorithm (i.e. the Cooley-Tukey FFT [15] Eq .4) optimised for that sized transform and the current hardware [22]. Although this has been tested, there currently exists no formal verification of its correctness, meaning that bugs which have not yet been found could exist within it.

As FFTW does not come with such formal guarantees separate definitions of various FFTs have been created before in proof assistance such as Coq [19] and Hol [23] with various methods and goals. In the paper "Certifying the Fast Fourier Transform with Coq" [24], Capretta makes use of binary trees to create a definition of the Cooley-Tukey FFT [15] for the radix-2 case (when $r_1 = 2$). This definition is then proven to be extensionally equal to that of the DFT. This provides a good definition for the radix-2 case of the FFT, allowing for it to be built on to create future proofs should they require the FFT, however, it does not cover the generalisation on the radix restricting the proof to specific splitting strategies.

In another paper, "A Methodology for the Formal Verification of FFT Algorithms in HOL", [25] Akbarpour and Tahar create two definitions of the Cooley-Tukey FFT [15] in Hol for the radix-2 and radix-4 cases. With a primary focus on the radix-2 case Akbarpour and Tahar go on to show equivalence to the DFT across various levels of abstraction [25]. At one stage of this abstraction, Akbarpour and Tahar introduce floating and fixed point arithmetic, showing an analysis of the resultant errors [25]. Much like Capretta [24], Akbarpour and Tahar [25] do not make use of a general radix, however, the paper does highlight how its methodology can be used to analyse general radix FFT implementations.

Currently, all previous work to formally verify the Cooley Tukey FFT has used fixed radices, while most common implementations utilise mixed radices which "are adapted to the hardware" [1]. This show a gap in the existing research, as no verification on these mixed radix cases is present.

3 Implementation

Before the DFT and FFT can be reasoned on, it is important to invest into the definition of some data structures which can accurately encode all required data and operations upon that data. Well defined data structures allow us to abstract useful properties on the data held within them. This allows for reasoning to be performed on a high level where:

- Variations of the Fourier Transform, such as the Inverse Fourier Transform, can be instantiated without a need to modify the underlying code.
- Definition can be made highly compact.
- The possibility for out-of-bound indexing errors can be eliminated by construction.
- The FFT can be defined for an input tensor of arbitrary shape, allowing multiple kernels to be defined using just one definition.
- The FFT can be described over a number system of arbitrary structure, allowing it to be instantiated for any number system with the correct properties

3.1 Complex Numbers

It is well known [26] that the DFT and FFT can be implemented on an arbitrary field-*like* [3] structure with roots of unity. Agda allows this this idea to be captured precisely though the creation of a structure, Cplx, which axiomatizes this field and its properties which the FFT and correctness proof rely on. This is similar to Java interfaces, defining the carrier and operations, but also allows for the properties (such as the associativity of addition) of this field to be defined.

This isolation allows the definition of the DFT, FFT and proofs to be instantiated for any implementation of Cplx. This generality allows the use of any modular field of sufficient size which holds the required properties, allowing operations such as fast multiplication to be performed upon these fields.

With the required operations and properties in mind, a structure can be formed to encapsulate complex. This structure first defines the carrier set, $\mathbb{C}$, and the of basic operations any implementation of complex must contain, each defined in a similar way to _+_ below.

```
record Cplx : Set₁ where
field
  ℂ : Set
  _+_ : ℂ → ℂ → ℂ
  -- ...
```

Addition, multiplication and negation must be proven to form a commutative ring, meaning that a set of properties, such as multiplication distributes over addition must hold. [27]

[3] This structure is only field-*like* because it does not require multiplicative inverses

```
-- ...
+-*-isCommutativeRing : IsCommutativeRing _+_ _*_ -_ 0ℂ 1ℂ
-- ...
```

Roots of unity as described for Complex numbers in Equation 2, must be defined for some non-zero divisor N and some power K, along with some properties on them. To ensure that the divisor N is never zero, a NonZero proof argument is required on N, guaranteeing division by zero to be impossible. This NonZero property is an instance argument, allowing an instance resolution algorithm[28] to perform automatic resolution on it, simplifying further proofs.

```
-- ...
-ω : (N : ℕ) → .{{ nonZero-n : NonZero N }} → (k : ℕ) → ℂ
ω-N-0       : -ω N 0                ≡ 1ℂ
ω-N-mN      : -ω N (N *ₙ m)         ≡ 1ℂ
ω-r₁x-r₁y   : -ω (r₁ *ₙ x) (r₁ *ₙ y) ≡ -ω x y
ω-N-k₀+k₁ : -ω N (k₀ +ₙ k₁)       ≡ (-ω N k₀) * (-ω N k₁)
```

3.2 Tensors

In Equations 1 and 4, the DFT and FFT are both defined for any input vector x of length N and length $r_1 \times r_2$ respectively. This implies that it would be possible to represent the input structure for both the DFT and the FFT in vector form, possibly using the Agda standard libraries functional vector definition, `Data.Vec.Functionals`.

Although this structure is ideal for the DFT, the FFTs relies on index splitting, as described in Equation 3, to decompose the input vector into r_1 parts. For vectors this would require low level index manipulation, for a single layer of splitting, this is not unreasonable, but can still complicate any definitions. For multiple layers however, where the input is split into n factors, this quickly becomes complex as the multipliers and split position for each factor must be carried through. This would make an kind of reasoning on the FFT, as well as generalisation, where the FFT is called iteratively, difficult as both would be pulled down to require the same low level of index manipulation.

The need for this low level manipulation can be removed, by creating some definition for shaped tensors, and allowing the FFT to accept these tensors as inputs. These shaped tensors can also be considered as Multi-dimensional arrays. As well as removing the need these low level manipulations, using this definition will also abstract the splitting of the input vector out of the FFT making any definition radix independent.

The shape of any given tensor can be described as a full binary tree of natural numbers. Each leaf, ι n, is one such natural number, one leaf can be considered to add one dimension to the overall shape. Each parent note, s ⊗ p, joins two subtrees. A given shape tree encodes the split of N into m many multipliers.

```
data Shape : Set where
  ι : ℕ → Shape
  _⊗_ : Shape → Shape → Shape
```

Defining shapes as trees in place of lists allows for more information to be encoded about the structure of the shape. This data loss can be identified by converting the below tensor shapes into their list forms, both of which are s :: p :: r :: q :: [].

$$s_1 = (s \otimes p) \otimes (r \otimes q)$$
$$s_2 = s \otimes (p \otimes (r \otimes q))$$

Array indices can then be inductively defined as a dependant type on Shapes. This definition takes the same form as that of shapes and defines the position of a non-leaf nodes as being constructed by the positions of its two children nodes, while leaf nodes are bound by the length of that leaf. This binding on the length of the leaf, allows the type checker to require evidence that a positions index is not greater than the length, removing the possibility for runtime out of bounds errors.

```
data Position : Shape → Set where
  ι : Fin n → Position (ι n)
  _⊗_ : Position s → Position p → Position (s ⊗ p)
```

Position can then be used to define the tensor data encoding, such that tensors form indexed types accepting a position and returning the value at that position.

```
Ar : Shape → Set → Set
Ar s X = Position s → X
```

This means any given tensor of Shape s and type X accepts a Position of shape s and returns a value of type X. This is a similar definition to that used in [29], and provides a basis on which tensors can be discussed

3.2.1 Tensor length

Given the shape of an array, we can compute the number of elements it contains by multiplying all components of the shape tree

```
# : Shape → ℕ
# (ι x) = x
# (s ⊗ s₁) = # s * # s₁
```

When computing the DFT and FFT, the number of elements in a given tensor is used to determine the base, or principal, root of unity. This base, however, cannot be zero as to avoid division be zero. Therefore, ω requires that a non zero proof argument to be provided.3.1. This can be easily achieved by restricting the DFT and FFT to operate only on tensors the number of elements is greater than zero, This means that any implementation of the DFT and FFT must be provided, or generate, a proof argument that no leaf is of zero length. For the simplicity of this paper we use the notation Ar^+ to indicate that a tensor is provided such a proof argument. This notation cannot be used in the final implementation where the non zero property must be provided explicitly, however, this obfuscates the key points and so this improved notation is used here.

3.2.2 One-dimensional tensors

Given the definition of tensors, we can begin by defining some basic operations which can be used upon them. The first operations we shall define will operate exclusively on single dimensional tensors, which are often referred to for succinctness as vectors.

Head and Tail Head and tail operations can be defined to allow for the deconstruction of any tensor of shape ι (suc n). head_1 returns the first element of the tensor, while tail_1 returns all following elements in a tensor of shape ι n. These operations allow for recursion over vectors to be defined.

```
head₁ : Ar (ι (suc n)) X → X
head₁ ar = ar (ι fzero)

tail₁ : Ar (ι (suc n)) X → Ar (ι n) X
tail₁ ar (ι x) = ar (ι (fsuc x))
```

One feature of Agda used regularly is seen here, pattern matching. This is a feature found in most functional languages that use algebraic types, such as Haskell, and allows for the breaking down of some types of input fields to the types they are built on. In the above example ι x is of type Position (suc n), which is deconstructed to expose x of type Fin (suc n).

Sum From Equation 1, it can be seen that an operation to sum all elements in a given array is required. By defining this operation generally, over any binary operation and neutral element, we are able to represent any fold-like operations including the sum operation we require. This definition is can be instantiated for any commutative monoid (X, _·_, ϵ) where

- X is a set
- _·_ is some operation X → X → X, such that
  - x · y ≡ y · x
  - (x · y) · z ≡ x · (y · z)
- ϵ is an identity element in X such that ϵ · x ≡ x

With the above definition, sum can be defined as below.

```
module Sum
  {A : Set}
  (_·_ : Op₂ A)
  (ϵ : A)
  (isCommutativeMonoid : IsCommutativeMonoid {A = A} _≡_ _·_ ϵ)
  where
  sum : (xs : Ar (ι n) A) → A
  sum {zero}  xs = ϵ
  sum {suc n} xs = (head₁ xs) · (sum ∘ tail₁) xs
```

For the DFT and FFT in this paper, this is instantiated over complex addition, described as the monoid (ℂ, _+_, 0ℂ). However, this definition allows for any fold-like operation to be defined for any instance of (X, _·_, ϵ) meaning operations such as Π can be instantiated with the same definition and general rules. This is similar to how the DFT and FFT can be instantiated for any definition of Cplx.

Index's in a single dimension . As defined above, Position encodes the bounds on a given index, as well as the index itself. When calculating the DFT some arithmetic on this index is required, this arithmetic would be overly complex if performed while the index is wrapped in a position, and so helper functions are required to convert a given position to its index value. This helper function for the single dimensional case is shown below.

```
iota : Ar (ι N) ℕ
iota (ι i) = toℕ i
```

3.3 DFT

Given the above definition of the complex numbers, tensors, and methods on one dimensional tensors, the formation of the DFT is now trivial. This is of the same shape as Equation 1, requiring through use of Ar^+ that the length of any input vector is non zero, as to satisfy this same condition on the divisor of -ω as defined in 3.1.

```
DFT : Ar⁺ (ι N) ℂ → Ar⁺ (ι N) ℂ
DFT {N} xs j = sum λ k → xs k * -ω N (iota k *ₙ iota j)
```

3.4 Reshape

When working with tensors, it is often necessary for elements to be rearranged, through operations such as transpose or flatten, without any additions or removals. The naïve approach to this, would be to define each rearrange as a function of type Ar s X → Ar p X. This approach however, would operate on too large a space, meaning reasoning upon such functions would be difficult and could not be generalised. An alternate approach is to define a small language of reshapes. This language captures a small set of rearrangements, as well as methods to allow for their composition. Generalised properties, such as how each reshape is applied to a position, can then be defined on this language or reshapes.

For this language, each reshape operations can be considered as a bijective function from shape s to shape p. As this ensures that no tensor can loose or gain data, creating a strict reshape language will strengthen any reasoning in future proofs. This also means that any reshape operation is reversible which will allow for the formation of rules which are general to all operations in the reshape language.

The reshape language is defined as a set of operations from shape to shape as follows.

```
data Reshape : Shape → Shape → Set where
  eq    : Reshape s s                    -- Identity
```

```
_·_   : Reshape p q                          -- Composition of Reshapes
      → Reshape s p
      → Reshape s q
_⊕_   : Reshape s p                          -- Left/ Right application
      → Reshape q r
      → Reshape (s ⊗ q) (p ⊗ r)
split : Reshape (ι (m * n)) (ι m ⊗ ι n) -- "Vector" → 2D Tensor
flat  : Reshape (ι m ⊗ ι n) (ι (m * n)) -- 2D Tensor → "Vector"
swap  : Reshape (s ⊗ p) (p ⊗ s)              -- Transposition
```

Using this definition of reshape and some standard library methods on Fin, it is then possible do define the application of reshape to positions and tensors.

```
_⟨_⟩ : Position p → Reshape s p → Position s
i           ⟨ eq      ⟩ = i
i           ⟨ r · r₁  ⟩ = i ⟨ r ⟩ ⟨ r₁ ⟩
(i ⊗ j)     ⟨ r ⊕ r₁  ⟩ = (i ⟨ r ⟩) ⊗ (j ⟨ r₁ ⟩)
(ι i ⊗ ι j) ⟨ split   ⟩ = ι (combine i j)
ι i         ⟨ flat    ⟩ = let a , b = remQuot _ i in ι a ⊗ ι b
(i ⊗ j)     ⟨ swap    ⟩ = j ⊗ i

reshape : Reshape s p → Ar s X → Ar p X
reshape r a ix = a (ix ⟨ r ⟩)
```

3.4.1 Reverse

As each reshape operation is a bijective function, it is trivial to define a reverse method.

```
rev : Reshape s p → Reshape p s
rev eq = eq
rev (r ⊕ r₁) = rev r ⊕ rev r₁
rev (r · r₁) = rev r₁ · rev r
rev split = flat
rev flat = split
rev swap = swap
```

From this operation, rules on reshape can be defined, allow for formation of relations between reshape operations. This allows for the reduction of the reshape language when operations such as split · flat occur.

```
rev-eq :
  ∀ (r : Reshape s p)
    (i : Position p )
  ---------------------
  → i ⟨ r · rev r ⟩ ≡ i

rev-rev :
  ∀ (r : Reshape s p)
```

```
  (i : Position p )
----------------------------
→ i ⟨ rev (rev r) ⟩ ≡ i ⟨ r ⟩
```

3.4.2 Recursive Reshaping

While the above operations of reshape can be applied to tensors of a fixed shape this language of reshapes can be improved with the creation of recursive reshape operations.

Flatten and Unflatten enable the recursive application of flat and split respectively. This allows for an N-dimensional tensor to be flattened, and for any single dimensional tensor of size length s to be unflattened into a tensor of shape s.

```
♭ : Reshape s (ι (length s))
♭ {ι x } = eq
♭ {s ⊗ s₁} = flat • ♭ ⊕ ♭

-- Unflatten is free from flatten
♯ : Reshape (ι (length s)) s
♯ = rev ♭
```

Transpose flips a tensor over its diagonal by swapping the left and right sub-shape at each level. Transpose applies swap to any non leaf nodes, allowing for any given function designed to operate on multi dimensional tenors, such as the FFT, to do the same swap at each level. It can be seen below that transpose is defined through use of the postfix operator, meaning the input shape goes before ^{t}

```
_ᵗ : Shape → Shape
_ᵗ (ι x ) = ι x
_ᵗ (s ⊗ s₁) = (s₁ ᵗ) ⊗ (s ᵗ)
```

3.5 Operations on tensors of Arbitrary rank

In addition to the above reshape operations, some methods which can operate directly on multi dimensional tensors are required.

Zip With performs point-wise application of a given function f over two tensors of the same shape.

```
zipWith : (X → Y → Z) → Ar s X → Ar s Y → Ar s Z
zipWith f arr₁ arr₂ pos = f (arr₁ pos) (arr₂ pos)
```

This has many uses, below is shown one example where zipWith is used over tensors x and y, of shape (ι n ⊗ ι m), to add the values at each position. This two dimensional

shape is defined arbitrarily for ease of readability, however, zipWith is not restricted on the shape meaning a tensor of any shape can be used.

$$\text{zipWith } \_+\_ \begin{bmatrix} x_{1,1} & \dots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,n} \end{bmatrix} \begin{bmatrix} y_{1,1} & \dots & y_{1,n} \\ \vdots & \ddots & \vdots \\ y_{m,1} & \dots & y_{m,n} \end{bmatrix} \equiv \begin{bmatrix} x_{1,1} + y_{1,1} & \dots & x_{1,n} + y_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} + y_{m,1} & \dots & x_{m,n} + y_{m,n} \end{bmatrix}$$

Map is similar to zipWith, but operates over only one tensor, applying a function f to each element.

$$\begin{aligned} &\text{map} : (f : X \to Y) \to \text{Ar } s\ X \to \text{Ar } s\ Y \\ &\text{map } f\ arr\ i = f\ (arr\ i) \end{aligned}$$

The functions nest and unnest can then be defined to create an isomorphism between tensors of the form Ar (s ⊗ p) X and nested tensors of the form A s (Ar p X). This allows for the definition of a new function mapLeft which can apply a given function to each p shaped sub tensor.

$$\begin{aligned} &\text{nest} : \text{Ar } (s \otimes p)\ X \to \text{Ar } s\ (\text{Ar } p\ X) \\ &\text{nest } arr\ i\ j = arr\ (i \otimes j) \end{aligned}$$

$$\begin{aligned} &\text{unnest} : \text{Ar } s\ (\text{Ar } p\ X) \to \text{Ar } (s \otimes p)\ X \\ &\text{unnest } arr\ (i \otimes j) = arr\ i\ j \end{aligned}$$

$$\begin{aligned} &\text{mapLeft} : \forall\ \{s\ p\ t : \text{Shape}\} \to (\text{Ar } p\ X \to \text{Ar } t\ Y) \to \text{Ar } (s \otimes p)\ X \to \text{Ar } (s \otimes t)\ Y \\ &\text{mapLeft } f\ arr = \text{unnest } (\text{map } f\ (\text{nest } arr)) \end{aligned}$$

3.6 FFT

Given the above operations, it is now possible to begin forming a definition for the FFT.

Looking at the basic derivation of the Cooley Tukey FFT over an input vector defined in Equation 4, three distinct sections can be observed.

$$X_{j_1 r_1 + j_0} = \underbrace{\sum_{k_0=0}^{r_2-1} \underbrace{\left[\underbrace{\left(\sum_{k_1=0}^{r_1-1} x_{k_1 r_2 + k_0} \omega_{r_1}^{k_1 j_0} \right)}_{SectionA} \omega_{r_1 r_2}^{k_0 j_1} \right]}_{SectionB} \omega_{r_2}^{k_0 j_1}}_{SectionC} \tag{5}$$

Section A takes the form of a DFT of length r_1. In vector form, the first element of the input for this DFT is located at index k_0, each subsequent input is then found taken by making a step of r_2, r_1 times. In vector form this is a relatively complex input to reason upon, when we can instead consider our input in tensor form, initially, as a tensor of shape $\iota\ r_1 \otimes \iota\ r_2$. In this form, Section A can be considered to apply the DFT to each column of the input tensor. Similar to Section A, Section C then takes the form of a DFT of length r_2. In our $\iota\ r_1 \otimes \iota\ r_2$ tensor form, this is equivalent to the application of the DFT over the rows of the result of section B.

Section B differs to section A and C, and applies what are generally referred to as, the twiddle factors. In tensor form this section is equivalent to a point wise multiplication over each element from Section A. This step can be represented in Agda as zipWith _*_, on a tensor containing these "twiddle factors".

```
2D-twiddles : Ar⁺ (ι r₂ ⊗ ι r₁) ℂ
2D-twiddles {r₁} {r₂} (k₀ ⊗ j₁) = -ω (r₂ *ₙ r₁) (iota k₀ *ₙ iota j₁ )
```

It can be seen here that when computing these twiddle factors, the number of elements in the input vector is used as the base value. It is defined previously that this base value cannot be zero, and so this step imposes the requirement that the FFT can only operate upon tensors with one or more elements.

Using this twiddle tensor, the definition for the two dimensional FFT is generated by forming each section into its own step. Of note in the definition below are the three uses of swap. The first swap allows DFT' to map over the columns of the input array, while the next inverts this and allows map to be performed over the rows. The final swap is performed because, given an input in row major order, the result of the FFT is produced in column major order. For this to be represented correctly when flatten, ♭, is applied the output tensor must be transposed, this is performed over two dimensions with swap. Because of this third swap, the shape of the output tensor is transposed, as indicated to in the function type by ^{t}.

```
2D-FFT : Ar⁺ (ι r₁ ⊗ ι r₂) ℂ → Ar⁺ ((ι r₁ ⊗ ι r₂) ᵗ) ℂ
2D-FFT {r₁} {r₂} arr =
```

```
let
  innerDFTapplied      = mapLeft (DFT {r₁}) (reshape swap arr)
  twiddleFactorsApplied = zipWith _*_ innerDFTapplied 2D-twiddles
  outerDFTapplied      = mapLeft (DFT {r₂}) (reshape swap twiddleFactorsApplied)
in reshape swap outerDFTapplied
```

Given knowledge that the DFT should be equivalent to the FFT, the two dimensional definition can then be improved by instead applying the FFT at each step. This requires the slight modification of the 2D-FFT implementation such that it accepts a tensor of any shape s as input.

The definition for the twiddle factors must also be redefined, such that twiddles can be computed for any shape with more than two dimensions. It is easy to see, that the previous base of the roots of unity, $r_1 \times r_2$, maps directly to the length of any given tensor. To calculate the power of the root of unity, we can define offset-prod. This flattens the values of k and j, before multiplying them together to calculate the power.

```
offset-prod : Position (s ⊗ p) → ℕ
offset-prod (k ⊗ j) = iota (k ⟨ ♯ ⟩) *ₙ iota (j ⟨ ♯ ⟩)

twiddles : Arⁿ (s ⊗ p) ℂ
twiddles {s} {p} i = -ω (length (s ⊗ p)) (offset-prod i)
```

The definition of this general twiddle tensor now allows for FFT to be defined for an input of any shape. The same problem of the output shape must then be dealt with again. As the result of the FFT is in column major order, the result must be transposed for flatten to represent it correctly. This can be achieved through the application of swap to outerDFTapplied before returning, as each sub tensor is the result of the application of the FFT and will be transposed.

```
FFT : Arⁿ s ℂ → Arⁿ (s ᵗ) ℂ
FFT {ι N} arr = DFT arr
FFT {r₁ ⊗ r₂} arr =
  let
    innerDFTapplied      = mapLeft FFT (reshape swap arr)
    twiddleFactorsApplied = zipWith _*_ innerDFTapplied twiddles
    outerDFTapplied      = mapLeft FFT (reshape swap twiddleFactorsApplied)
  in reshape swap outerDFTapplied
```

As time was invested at the start of the project into a the creation of a language on tensors and reshaping, every case of the Cooley Tukey algorithm can be represented within the three lines shown above. Given a proof of correctness, this generality makes way for further experiments into different radix sizes, and combination of radix sizes, to be easily undertaken.

If this was instead written in `C`, or a `C` style language, this level of generality would be almost impossible. Any such general, `C` style implementation would require many, low level, index manipulations. Without structures such as those defined for here for position, these index manipulations become increasingly complex as the radix sizes, and levels of nesting, increase. This complexity makes it difficult to reason upon any such implementation meaning garuntees are more challenging to achive.

4 Proof of correctness

Given the above defintion of the FFT, and our previous definition of the DFT, a proof of equality between the two can be formed.

To define the relation between DFT and FFT, pointwise equality _≅_ can be used used. This defines equality between two tensors of shape s to hold when ∀ (i : Position s) → xs i ≡ ys i. This allows for proofs to be defined for a general position i.

As the DFT operates on the vector form, reshape operations must be used to flatten the input tensor and unflatten the output for comparison. Not mentioned previously, is the reindex reshape operation. Reindex allows for poitions to be cast from ι n to ι m when n ≡ m without any change in indices. As the output of the FFT must be read in column major order, it is of the form s ^{t}. When flattened this gives a tensor of shape ι (# s ^{t}). Meanwhile, without the use of reindex, the output of the DFT is of shape ι (# s). Reindexing allows this to be modeled as ι (# s ^{t}) without reordering the results in this tensor. This allows for the use of pointwise equality.

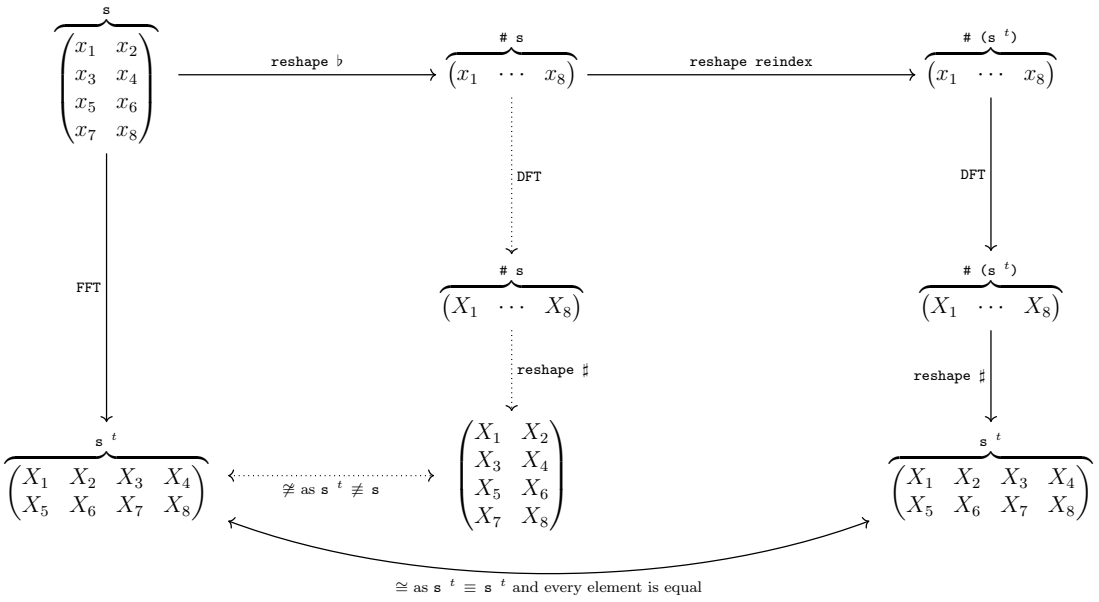

Figure 1: Diagram showing the effects of reshaping

Using what we now know from the above relation, the proposition for this proof describing the relationship between the FFT and DFT can be formed.

```
fft≅dft :
  ∀ (arr : Ar⁺ s ℂ)
→ FFT arr
  ≅
  ( (reshape ♯)
  ∘ DFT
  ∘ (reshape (reindex (|s|≡|s^t| {s}) · ♭))) arr
```

4.1 Chain of Reasoning

While the proposition defines what we wish to prove, the chain of reasoning is used to justify that the proof holds. The full proof can be found in the attached files, while the most important sections are discussed here. It is important to note that as proofs must hold every invariant, at every step a large amount of complexity is held within this chain of reasoning. As done previously to hide NonZero, as much complexity as possible is hidden in the below extracts from the main chain of reasoning as to improve readability. This complexity remains important, however, as it what allows the strict guarantees provided by Agda to hold. The full proof can be found in the attached files.

Before the chain of reasoning can be defined, some specific syntax must be described which is used to define this chain of reasoning.

Underscores are used throught the proof to hide complexity when Agda is able to automatically infer a value. This allows for the low level complexity to be hidden while working on high level aspects of the proof.

_≡⟨_⟩_ represents one step in a chain of reasoning. Therefore a ≡⟨ p ⟩ b can be read as "a is equivilant to b, using the evidence provided in p", where p should be of type a ≡ b.

_⊡_ represents the trasnitivity of proofs. Say we have $p_1 : a \equiv b$ and $p_2 : b \equiv c$. p_1 ⊡ p_2 would provide evidence that a ≡ c.

4.1.1 Inductive Step

The main proof is built inductively on the case of a one dimensional tensor, and a multi multi dimensional tensor. This allows my proof to match the shape of the FFT definition.

```
fft≅dft {ι N} {{ ι nz-N }} arr i = refl                    (1)
fft≅dft {r₁ ⊗ r₂} {{ nz-r₁ ⊗ nz-r₂ }} arr (j₁ ⊗ j₀) =      (2)
```

These first two lines of this chain of reasoning split the proof on the shape of the input tensor. 1 pattern matches the case where the shape is one dimensional, as FFT on such a shape is equal by definition to the DFT, no chain of reasoning is required to prove this case. This is the base case of the induction. 2 pattern matches on the alternate case, and precedes the remainder of the proof, where r_1 and r_2 represent the left and right sub shapes.

```
  begin
    FFT {r₂} (λ k₀ →
          FFT {r₁} (λ k₁ → _) j₀ * _                       (3)
    ) j₁
  ≡⟨ fft≅dft _ j₁ ⟩
    DFT {# r₂ ᵗ} (λ k₀ →
          FFT {r₁} (λ k₁ → _) j₀ * _                       (4)
    ) (j₁ ⟨ ♯ ⟩)
```

```
    ≡⟨ DFT-cong (λ x → cong₂ _*_ (fft≅dft _ j₀) refl) (j₁ ⟨ ♯ ⟩ ) ⟩
      DFT {# r₂ ᵗ} (λ k₀ →
            DFT {# r₁ ᵗ} (λ k₁ → _) (j₀ ⟨ ♯ ⟩) * _                    (5)
      ) (j₁ ⟨ ♯ ⟩)
    -- ...
```

Splitting upon the shape allows the left hand side to take the form shown in step 3. Step 4 and 5 are then able to apply the proposition currently being proven to the outer and inner instances of FFT. This allows both instances to be represented as DFT, which in turn allows for the representation of the left hand side in sum form.

```
    ≡⟨⟩
      sum {# r₂ ᵗ} (λ k₀ →
            sum {# r₁ ᵗ} (λ k₁ →
                  arr _
              *
                  -ω _ _ -- Inner DFT -ω
            )
        *
            -ω _ _       -- Twiddle Factor -ω
        *
            -ω _ _       -- Outer DFT -ω
      )
```

4.1.2 Cooley Tukey Derivation

Using the rule that $c \times \sum_{i=0}^{n} x_i \equiv \sum_{i=0}^{n} cx_i$, the two instances of -$\omega$ in the outer sum, can be moved into the inner sum. With all instances of -ω gathered, the rules of the roots of unity can be used, following the inverse of the initial Cooley Tukey derivation, to represent all roots of unity as one.

As the main logic of this step considers positions as natural numbers, we can define the body of the proof as a lemma on six natural numbers. By providing concrete values for each of these numbers, this lemma can then be applied in the main proof.

```
cooley-tukey-derivation :
  ∀ (r₁ r₂ k₀ k₁ j₀ j₁ : ℕ)
  → {{ nonZero-r₁ : NonZero r₁ }}
  → {{ nonZero-r₂ : NonZero r₂ }}
  →
    -ω
      (r₂ *ₙ r₁)
      {{ m*n≢0 r₂ r₁ }}
      (
        (r₂ *ₙ k₁ +ₙ k₀)
        *ₙ
        (r₁ *ₙ j₁ +ₙ j₀)
      )
    ≡
      -ω (r₁) (k₁ *ₙ j₀)
    * -ω (r₂ *ₙ r₁) {{ m*n≢0 r₂ r₁ }} (k₀ *ₙ j₀)
```

```
    * -ω (r₂) (k₀ *ₙ j₁)
cooley-tukey-derivation r₁ r₂ k₀ k₁ j₀ j₁ {{ nonZero-r₁ }} {{ nonZero-r₂ }}
  = begin
      -ω
        (r₂ *ₙ r₁)
        ( (r₂ *ₙ k₁ +ₙ k₀) *ₙ (r₁ *ₙ j₁ +ₙ j₀) )
    ≡⟨ rearrange-ω-power ⟩
      -ω
        (r₂ *ₙ r₁)
        ( r₂ *ₙ (k₁ *ₙ j₀)
          +ₙ k₀ *ₙ j₀
          +ₙ r₁ *ₙ (k₀ *ₙ j₁)
          +ₙ r₂ *ₙ (r₁ *ₙ (j₁ *ₙ k₁))
        )
    ≡⟨ split-ω ⟩
          -ω (r₂ *ₙ r₁) (r₂ *ₙ (k₁ *ₙ j₀))
        * -ω (r₂ *ₙ r₁) (k₀ *ₙ j₀)
        * -ω (r₂ *ₙ r₁) (r₁ *ₙ (k₀ *ₙ j₁))
        * -ω (r₂ *ₙ r₁) (r₂ *ₙ r₁ *ₙ (j₁ *ₙ k₁))
    ≡⟨ remove-constant-term ⟩
          -ω (r₂ *ₙ r₁) (r₂ *ₙ (k₁ *ₙ j₀))
        * -ω (r₂ *ₙ r₁) (k₀ *ₙ j₀)
        * -ω (r₂ *ₙ r₁) (r₁ *ₙ (k₀ *ₙ j₁))
    ≡⟨ simplify-bases ⟩
          -ω r₁ (k₁ *ₙ j₀)
        * -ω (r₂ *ₙ r₁) (k₀ *ₙ j₀)
        * -ω r₂ (k₀ *ₙ j₁)
    ■
```

It can clearly be seen that the above lemma has four distinct steps.

Rearrange Power rearrange-ω-power expands the second term of -ω, and rearanges the result such that r_2 then r_1 are the rightmost elements. With standard Agda methods, this would require a large chain of reasoning, where each set could apply one property on the natural numbers. Instead, the `solver` for natural numbers can be used. Natural numbers, addition, and multiplication form an algebraic structure called a Commutative Ring. To form this structure a set of properties on the number type must hold, this set of properties includes commutativity, associativity and the distributive property of multiplication over addition. The `solver` method is able to utilise these properties to form chains of reasoning automatically. This simplifies what would otherwise be a long, strict proof while maintaining all correctness properties.

```
rearrange-ω-power =
  -ω-cong₂
    refl
    (solve
      6
      (λ r₁ℕ r₂ℕ k₀ℕ k₁ℕ j₀ℕ j₁ℕ →
```

```
                (r₂ℕ :* k₁ℕ :+ k₀ℕ)
            :*
                (r₁ℕ :* j₁ℕ :+ j₀ℕ)
        :=
                r₂ℕ :* (k₁ℕ :* j₀ℕ)
            :+  k₀ℕ :* j₀ℕ
            :+  r₁ℕ :* (k₀ℕ :* j₁ℕ)
            :+  r₂ℕ :* (r₁ℕ :* (j₁ℕ :* k₁ℕ)))
      refl
      r₁ r₂ k₀ k₁ j₀ j₁
    )
```

As ring is defined generally, a special syntax must be used to describe the lemma to solve.

Split ω split-ω applies ω-N-k$_0$+k$_1$, which defines that $-\omega_N^{k_0+k_1} \equiv -\omega_N^{k_0} + -\omega_N^{k_1}$. This breaks down the current large power of the root of unity, into four smaller roots of utranspose nity.

Remove Constant Term remove-constant-term applies ω-N-mN, which states that $-\omega_N^{Nm} \equiv 1$, to remove the last root of unity $-\omega_{r_2 r_1}^{(r_2 r_1)(j_1 k_1)}$.

Simplify Bases simplify-bases applies ω-r$_1$x-r$_1$y, which states that $-\omega_{r_1 x}^{r_1 y} \equiv -\omega_x^y$, to eliminate r$_2$ or r$_1$ when they appear in both the power and the base.

4.1.3 Nesting of Sums

With the inverse of the above derivation applied to the current chain of reasoning, the left hand side (from the FFT) takes the form of a nested sum with only one root of unity.

```
sum {# r₂ ᵗ}
  (λ k₀ →
    sum {# r₁ ᵗ} (λ k₁ →
        arr
          ((k₁ ⊗ k₀) ⟨ ((reindex (|s|≡|sᵗ| {r₁})) ⊕ reindex (|s|≡|sᵗ| {r₂}))
                     · split
                     · flat
                     · (♭ ⊕ ♭) ⟩)
      *
        -ω (# r₂ ᵗ *ₙ # r₁ ᵗ) --
    )
  )
```

As the DFT to compare against takes the following form it is clear to see that the next step requires the manipulation these nested sums.

```
sum {# (r₁ ⊗ r₂) ᵗ} (λ k →
    arr (k ⟨ reindex (|s|≡|sᵗ| {r₁ ⊗ r₂}) · ♭ ⟩)
```

```
    *
      -ω (# r₂ ᵗ *ₙ # r₁ ᵗ) --
  )
```

This manipulation can be done in four steps.

sum-reindex can be used (once reversed) to remove the transposition on the length of the sum as the length of a shape, and the length of its transpostion are equal. This also removes all instances of reindex from the right hand side.

```
sum-reindex :
  ∀ {m n : ℕ}
    {xs : Ar (ι m) A}
  → (prf : m ≡ n)
  ----------------------------------------
  → sum xs ≡ sum (reshape (reindex prf) xs)
sum-reindex refl = refl
```

sum-swap can then be used to transition from sum {# r₂} λ k₀ → sum {# r₁} λ k₁ → _ to sum {# r₁} λ k₁ → sum {# r₂} λ k₀ → _. The proof which accompanies sum-swap is somewhat complicated, and an observant reader may ask if this step could be replaced by using the commutativity of multiplication after merge-sum, but this is not the case. For this papers definition of sum, we state that sum xs ≡ sum ys when xs ≅ ys. If the commutativity of multiplication was used after merge-sum in place of sum-swap this equality would not hold, as the elements of the tensors on each side would be ordered incorrectly.

merge-sum can then be used to go from two nested sums sum {# r₁} λ k₁ → sum {# r₂} λ k₀ → _ to a singlular sum, sum {# (r₁ ⊗ r₂)} λ k → _, this removes the combination of poisitions, (k₁ ⊗ k₀) ⟨ split ⟩ from the left hand side, replacing it with k.

sum-reindex can then be used to re-apply reindex. This makes the left hand side equal to the right hand side completing the chain of reasoning.

4.2 What this means

The proposition above declares that the result of the DFT is equal to the result of the FFT. The chain of reasoning which then follows the proposition proves it to be correct. This provides the guarantee that this definition of the FFT produces the same value as the DFT. This guarantee is provided on two generics, the definition of complex, and the shape meaning it is applicable to any, implementation of complex, or shaped tensor.

As complex is defined generically, any implementation which holds the required properties can be provided. This allows for the use of this FFT with its attached

guarantees on any finite field with complex roots of unity which can hold the required properties. This allows other formally verified systems operating on such as field to utilise the FFT without losing correctness guarantees, invariant of whether such a system operates on machine floats, or an abstract implementation of complex.

As this implementation is defined generically on the shape of the input tensor, no restriction is placed on the radix choice. This allows for the computation of the FFT on any input with a non prime length without the need for padding. If the implementation was instead defined for a fixed radix r, any input would need to be padded to length r^n. This zero padding is required for some implementations of the FFT, but can increase the amount of computation required.[30]

These guarantees also allow for this implementation of the FFT to be utilised in future Agda projects without the loss of correctness guarantees. This opens the door for any future research projects in Agda which require the FFT, allowing for proofs in such projects to be performed on the trivial DFT, while methods are implemented on the FFT, simplifying reasoning.

5 Compilation

Given the definition of the FFT, and proof of its correctness, it is is now possible for us to generate runnable code with the same guarantees of correctness attached. A runnable instance of this FFT implementation can be generated through use of Agda's Builtin IO library, and the GHC Haskell backend. The IO library allows for the definition of output and an entry point into the program. The GHC Haskell backend allows for translation into runnable Haskell, allowing us to confirm that the FFT and DFT implementations run and produce expected results.

5.1 Complex Numbers

To generate a runnable instance of the FFT as defined, an implementation of the Complex numbers must be defined. For the example shown here, I have defined complex numbers as pairs of Agda's builtin Float.

Unlike other number systems in Agda, builtin Float is not defined inductively, and is instead a wrapper around IEEE 754 [31]. This means that no properties, such as commutativity or associativity, are provided and that any such property must be assumed through postulations. This weakens implementations built on top of Float, as they become prone to minor floating point errors. Such errors are generally unavoidable, however, within floating point mathematics, and the resultant effects are generally well studied.[32] This makes the use of builtin Float acceptable for most implementations.

Given this implementation of floating point numbers as $\mathbb{R}$, the carrier, $\mathbb{C}$, for complex numbers can be defined as a pair of floating point numbers, representing the real and imaginary components.

```
record ℂ : Set where
  constructor _+_i
  field
    real-component : ℝ
    imaginary-component : ℝ
```

A constructor for $\mathbb{C}$ is defined at the same time, allowing for composition and decomposition into component parts. Builtin Float then provides wrappers around some primitive operations which can be used to define the required methods on this carrier set, such as addition.

```
_+_ : ℂ → ℂ → ℂ
_+_ (xRe + xIm i) (yRe + yIm i)
  = (primFloatPlus xRe yRe) + (primFloatPlus xIm yIm) i
```

Once all of these method are defined, their properties can be proven or postulated. In an ideal implementation, all postulations would be made on the methods of builtin Float, and the properties of complex would be paired with proofs. However, for this example I have chosen to postulate the properties of complex directly as to improve the simplicity of this example.

```
complexImplementation : Cplx real
complexImplementation = record {
```

```
ℂ = ℂ
; _+_ = _+_
-- ...
```

This implementation of complex numbers then allows for the FFT to be instantiated.

5.2 Runnable FFT

Given a concrete definition of the complex numbers, it is now possible to run my implementation of the FFT. The methods are not intended to be efficiant or practical for most uses. The input is hard coded before compilation, and the Haskell back-end is used to execute, however, it does allow for the results to be shown and analysed.

The `main` method is the entry point to the program and defines the input to run on which is parsed to show-full-stack. show-full-stack is then able to print out the original tensor, the tensor in vector form, the flat result of running the FFT, and the flat result of running the DFT.

```
show-arr arr = putStrLn $ "Tensor:      "
    ++ (showTensor showℂ $ arr)
show-flat-arr arr = putStrLn $ "Flat Tensor:"
    ++ (showTensor showℂ $ reshape flatten-reindex arr)
show-flat-FFT-result arr = putStrLn $ "FFT Result: "
    ++ (showTensor showℂ $ reshape (rev ♯) (FFT arr))
show-flat-DFT-result arr = putStrLn $ "DFT Result: "
    ++ (showTensor showℂ $ (DFT (reshape flatten-reindex arr)))

show-full-stack : Ar s ℂ → IO {a} ⊤
show-full-stack arr = do
  show-arr arr
  show-flat-arr arr
  show-flat-FFT-result arr
  show-flat-DFT-result arr
```

This allows for my implementation of the DFT and FFT to be ran for any input. The result of this execution can then be compared against the result of Numpy's FFT method, allowing for an average floating point error to be established against a well known implementation. We can expect this average error to be low, but not zero, with an expected worst case growth bounded by $(\log N)$[32]. This is because each operation on floating point numbers introduces minor rounding errors, and these minor errors then grow with each operation. As each of the three implementations use different operations in different orders, can be expected that these rounding errors will grow to different sizes.

I conducted a comparison of results for multiple input tensors. An example of one of these comparisons is shown below. All of these comparisons showed similar results, with the results of my implementation of the FFT and DFT matching those of the Numpy FFT, with some expected rounding errors. This shows my implementation to work as expected.

5.2.1 Example test

For this example, I used the following two dimensional tensor of shape $(\iota\ 4 \otimes \iota\ 4)$, constructed from randomly generated numbers between -100 and 100, as input.

$$\begin{pmatrix} 87 & 13 & 72 & -44 \\ 99 & 8 & -63 & 25 \\ 90 & -31 & 56 & 19 \\ -100 & 37 & 4 & 61 \end{pmatrix}$$

This is flattened for input to the DFT into the vector.

$$\begin{pmatrix} 87 & 13 & 72 & -44 & 99 & \dots & 61 \end{pmatrix}$$

With this tensor used as input, I am able to compare the results of the DFT, FFT and Python FFT. The full results can be found in the attached files, however, they show that the Agda FFT and DFT produce the same results as the Python FFT, with some expectedly minor rounding errors.

The average difference between each value of the Agda DFT, and each value of the Python FFT is 2.23×10^{-13}, while the maximum difference is 9.38×10^{-13}. The same average difference between the Agda FFT, and Python FFT is lower, at 4.05×10^{-14}, while the maximum difference is 1.42×10^{-13}. Being so low, it is easy to attribute these differences to the issues caused by floating point arithmetic as discussed above.

5.3 Future implementations

In the paper "Measuring the Haskell Gap", Petersen et al analyse the difference in performance between the "best performing `c` implementation and the best performing Haskell implementation".[33] In their initial benchmarks, Petersen et al describe implementations compiled with GHC as between 1.72× and 82.9× slower than `c` counterparts.[33] As translation into a GHC Haskell program is a required step to run my above implementation, this same negative performance is parsed on.

This is not, however, the only method with which an Agda definition can be made runnable. My formally proven FFT implementation can instead be ported into a high performace array language such as SaC [34] or Futhark[35]. Such a port would then allow for the introduction of parallelism, as well as allowing for the generation of efficient `c` code, avoiding performance issues caused by use of the Haskell compiler. Such a port would ideally allow for the correctness properties of my FFT definition to be parsed on to the definition built from it in the chosen array language. In [36], Šinkarovs and Cockx define such a method for Agda to Sac translation. As the number of operations I have used is very limited, and the small number of operations I have defined on tensors are very similar to those defined in SaC, this would be a easy next step. This would then allow for future investigation into the generation of code with FFTW like performance.

6 Project Review

This project has faced some significant delays. As discussed in my progress report, in the first half of this project, I had significant difficulty's with the implementation of the FFT. This difficulty was resultant of my initial lack of knowledge about Agda, a language I have not worked with previously, but was overcome after attempting four different implementation techniques.

After the progress report, however, I faced more difficulties. I initial believed that the proof equating the DFT and FFT would be somewhat trivial, I was very much mistaken. The final proof shown previously, and in the attached files was the result of a large number of iterations which changed both the preposition, and the chain of reasoning.

One major issue I faced was that my initial preposition, as was shown in my progress report, did not perform reindexing over the input vector for the DFT, as was shown in 4. I spent a large amount of time attempting to complete the proof without noticing this issue, and only discovered this issue when I formed a contradiction. This time was still helpful in the long run. This is because it greatly helped my understanding of proofs in Agda, allowing me to experiment with a large number of proof tactics, some of which I made use of in the final proof. So much time was lost to this issue, and the burnout it caused, that I was required to apply for an extension which allowed for the proof to be completed, and report written.

As seen previously, my final proof forms a strong relation between the DFT and FFT, providing a guarantee of the correctness of the FFT. This was the main goal of my project brief. However, because so much time was lost to a plethora of issues, I was unable to properly investigate `c` code generation. This was an additional, but non guaranteed, goal in my project brief which I had hoped would be achievable.

With this in mind, my original and my updated Gantt chart can be found in figure 2 and 3.

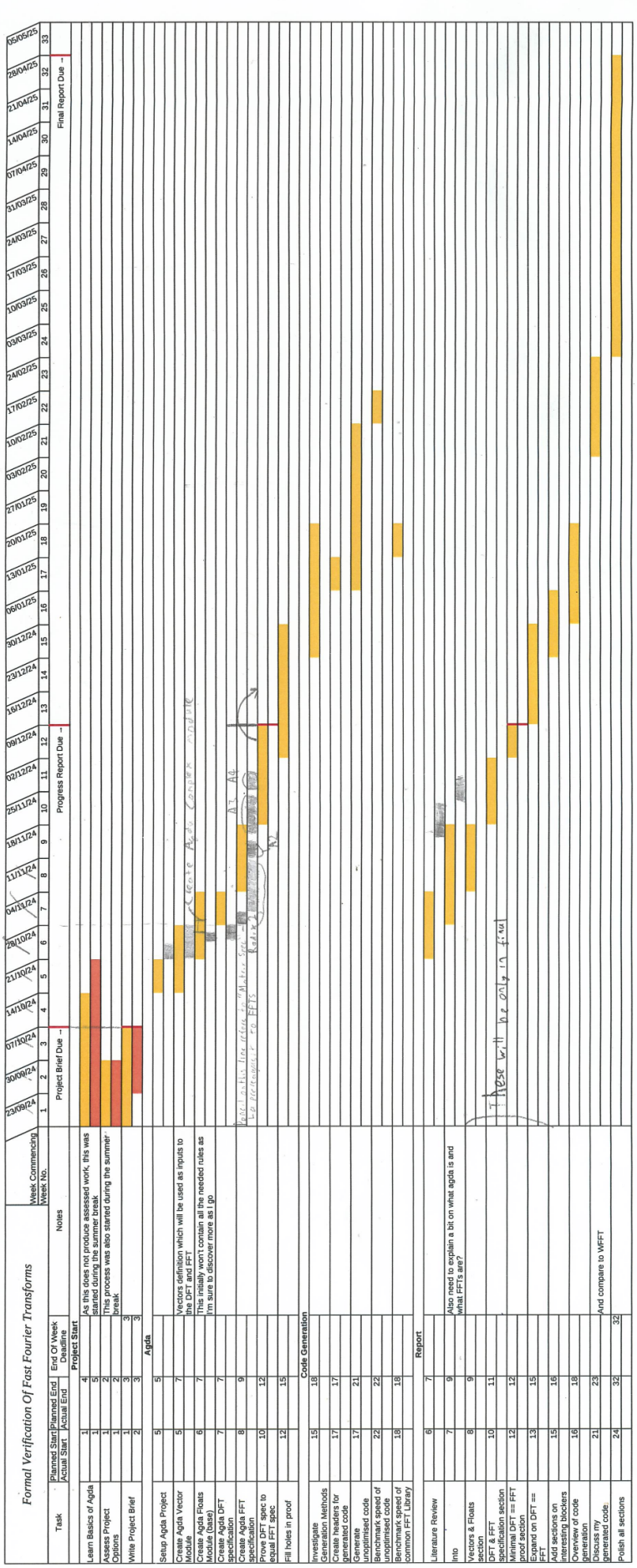

Figure 2: Gantt chart for the first half of the project

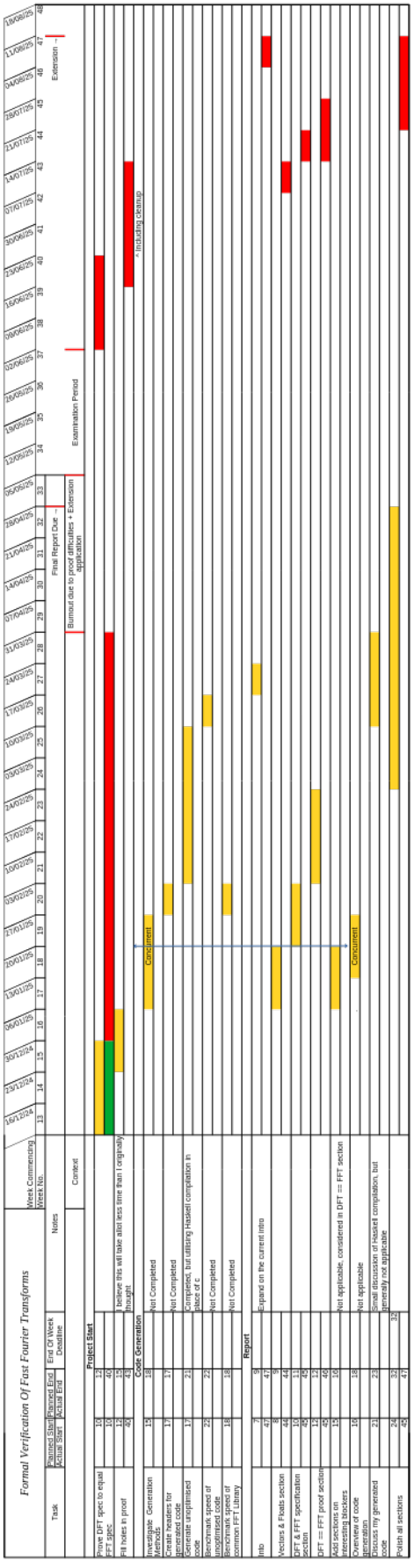

Figure 3: Final Gantt chart for the second half of the project

7 Conclusion

This paper provides a formally verified implementation of the Cooley Tukey Fast Fourier Transform. This implementation is built inductively, first defining representations of tensors and complex numbers. A trivial implementation of the Discrete Fourier Transform is then provided. This is then used to define an implementation of the Fast Fourier Transform.

Unlike most verifiable implementations of the FFT, this implementations of the FFT is radix generic and is defined on an abstract definition of Complex. Defining the FFT generically allows for any non prime input to be split optimally, and for the structure of any future parallelism to be defined at run time.

Given this general implementation of the FFT, I have then provided a proof that it is equal to the DFT for all possible inputs. With a basic implementation of the complex numbers, and a basic compiler, this allowed for the generation of a runnable, verified, instance of the FFT.

In future research I wish to investigate the generation of optimised code, through translation into an intermediate language such as SaC. The speed and floating point accuracy of such code would ideally be comparable to, or an improvement upon, the speed and accuracy of most similar kernels in FFTW. Should such comparison show significant improvement over the results of FFTW, investigation into the generation of a kernel for FFTW would be possible.

Additionally, within the Agda community, this work allows future research projects to make use of the FFT interchangeably with the DFT. This allows for future research into the verification of many signal processing algorithms, such as fast convolution or correlation, or for more general methods such as fast polynomial multiplication.