A Formalization of Fast Polynomial Multiplication in Coq

Mathias Yap

***Abstract*—Formal verification is an essential process for ensuring the correctness and reliability of systems, particularly in safety-critical engineering systems such as cybersecurity, healthcare, and transport. This thesis presents the formalization and verification of a fast polynomial multiplication algorithm using the Coq proof assistant. Building upon the Verified Calculus library, we address the inefficiencies of naive polynomial multiplication by implementing and proving the correctness of an algorithm based on the Fast Fourier Transform (FFT).**

Our contributions include the formalization of dense polynomial representations, the development of functions for polynomial operations, and the rigorous verification of the FFT and its application in polynomial multiplication. We assume one property of the complex roots of unity without a complete proof. This work demonstrates the feasibility and benefits of using formal methods to verify classical algorithms. Further research might look to formalize the roots of unity, extend the findings of this thesis to floating point arithmetic or formalize the algorithm for different polynomial function bases.

I. Introduction

Formal verification is the process of confirming the correctness of systems and code by using rigorous computational techniques. As systems get increasingly safety-critical, rigorously verifying them has become more important than ever. Ensuring the reliability and correctness of these systems can prevent errors and enhance safety in critical applications such as cybersecurity, healthcare and transport.

One common method of formal verification is through use of model checking. Model checking verifies system properties by means of an exhaustive search [5]. A notable tool for model checking is Ariadne, which is a library for formal verification of nonlinear hybrid systems previously developed at CWI, Amsterdam [2] and currently developed at the universities of Maastricht and Verona. As the evolution of such systems cannot be calculated exactly, Ariadne makes use of numerical analysis to compute results in an approximate way. It uses conservative rounding to guarantee correctness of its result. Model checking is vulnerable to state explosion.

Another solution to formal verification is deductive verification. Deductive verification consists of generating *proof obligations*, the truth of which implies conformance of a system to its specifications. Deductive verification can be done with the help of proof assistants such as Coq. Proof assistants are software tools that help with the development of formal proofs. Beyond just model checking, Coq is capable of formalizing mathematics.

This work builds upon the Verified Calculus library. The Verified Calculus library is an effort to verify the algorithms that are used in Ariadne using Coq. Verification in Coq will help to validate the analysis that Ariadne provides.

One of the basic operations implemented in the Verified Calculus library is polynomial multiplication, which is implemented naively. However, as the complexity of the polynomials increases, the naive approach becomes inefficient.

The problem statement of this work is as follows: Develop and formally prove the correctness of an algorithm for fast polynomial multiplication with rigorous error bounds in floating point arithmetic, using the Coq proof assistant.

The main research questions addressed in this thesis are:

RQ1 Is it possible to implement and prove the correctness of an algorithm for fast polynomial multiplication in floating point arithmetic using Coq?

RQ2 How do different approaches of polynomial multiplication differ in terms of both algorithmic complexity and the difficulty of proving correctness?

Although the proof presented in this thesis is done for reals, this work is also relevant for floats. As polynomial multiplication is exact, it is possible to reason about how it effects the error bounds of the float data type that the Verified Calculus library operates on.

II. Polynomial Multiplication

Naive polynomial multiplication, which we will define later, has a computational complexity of $O(n^2)$, where n is the length of the two polynomials. This becomes computationally expensive for polynomials with large degrees. To address this inefficiency, fast multiplication algorithms have been developed. The first asymptotically faster algorithm obtains a time complexity of $O(n^{1.58})$ and was published by Anatoly Karatsuba in 1962 [10]. In 1965, James Cooley and John Tukey rediscover the FFT [6]. The FFT and its inverse offer a way to efficiently transform polynomials from their coefficient form to their point-value form and back. A fast multiplication algorithm for polynomials in coefficient form that utilizes the FFT achieves a time complexity of $O(n \log n)$ [8].

In this section, we will cover the concepts of naive polynomial multiplication, polynomial representation, the Discrete Fourier Transform (DFT) and the FFT. Finally, we will outline the Fast Multiplication Algorithm that we shall formalize in Coq.

A. Naive Multiplication

Let $p = \sum_{i=0}^{m} a_i x^i$ and $q = \sum_{i=0}^{n} b_i x^i$ be polynomials over $\mathbf{R}$ with indeterminate x and degree m and n respectively. The degree of a polynomial is the highest of the degrees of the

polynomial's individual terms. The product of these two polynomials is another polynomial r such that $r(x) = p(x)q(x)$ for all x. A natural way to compute the product of the two polynomials is by multiplying each term of f with each term of g. This yields:

$$p = q \cdot r = \sum_{i=0}^{m+n} c_i x^i$$

where coefficients c_i are given by

$$c_i = \sum_{k=0}^{i} a_k b_{i-k} \quad \text{for } 0 \leq i \leq m+n$$

The time complexity of this method is $O(mn)$.

B. Polynomial Representation

The fast multiplication algorithm we will explore in this work relies on effective transition between two polynomial representations.

1) *coefficient representation*, where a polynomial is given by a list of its coefficients. Consider the polynomial $p = a_0 + a_1 x + a_2 x^2$. Here, it is represented simply by the list $[a_0, a_1, a_2]$.
2) *point-value representation*, where a degree n polynomial is given by its value at $n+1$ distinct points. This forms a set of pairs $\{(x_1, p(x_1)), ..., (x_{n+1}, p(x_{n+1}))\}$.

In both representations, the length of a polynomial is equal to the size of the list or set that represents them. In coefficient representation, a degree d polynomial is represented by a polynomial of length at least $d+1$. Trailing zeroes can affect the length of a polynomial without affecting its degree. In the coefficient representation, evaluating a polynomial has time complexity $O(n)$, while directly multiplying two polynomials has time complexity $O(n^2)$. Conversely, in the point-value representation, multiplication is straightforward. It is done pointwise over $m+n$ distinct points, resulting in a time complexity of $O(m+n)$. However, to evaluate a polynomial in point-value representation at a new point it is necessary to interpolate and obtain the polynomial coefficients. The Fourier Transform and its inverse provide a way to commute between these two representations.

C. Discrete Fourier Transform

The Discrete Fourier Transform (DFT) maps a sequence $\{x_n\}$ of length n to another sequence $\{X_k\}$, also of length n. Let $\omega_n = e^{2\pi i/N}$ be the principal n-th complex root of unity. The DFT is then defined by

$$X_k = \sum_{i=0}^{n-1} x_n \cdot w_i^{ik}$$

where w_n^{ik} are powers of the principal n-th root of unity. When applied to the coefficient representation of a polynomial, the DFT is equivalent to evaluating the polynomial at n points $w_n^0, w_n^1 ..., w_n^{n-1}$, transforming the polynomial from its coefficient representation into its point-value representation at distinct roots of unity. The inverse Discrete Fourier Transform (iDFT) remaps the sequence $\{X_k\}$ back to the sequence $\{x_n\}$. This converts back from the point-value representation into the coefficient representation. It is defined by

$$x_n = \frac{1}{n} \cdot \sum_{k=0}^{n-1} X_k \cdot w_n^{-nk}$$

Correctness of the iDFT is given by the following transformations to the sums:

$$\begin{aligned} \frac{1}{n} \sum_{k=0}^{n-1} (\mathrm{DFT}(p))_j \cdot \omega_n^{-j} &= \frac{1}{n} \sum_{k=0}^{n-1} \sum_{c=0}^{n-1} p_c \cdot \omega_n^{ck} \omega_n^{-jk} \\ &= \frac{1}{n} \sum_{c=0}^{n-1} p_c \sum_{k=0}^{n-1} \omega_n^{(c-j)k} \end{aligned}$$

Having factored out the coefficient of our polynomial from the nested sum, we are left with an inner sum of powers of the roots of unity. By the property

$$\sum_{j=0}^{n-1} \omega_n^{jk} = 0 \quad \text{for} \quad 1 \leq k < n \tag{1}$$

Of the unit roots, which stems from their even distribution on the unit circle, we can reduce the inner sum to n when $c = j$ and to 0 otherwise. This is the Kronecker delta $\delta_{c,j}$:

$$\delta(c,j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

Then we can infer correctness of the iDFT by showing that applying it after the DFT results in the original polynomial:

$$\frac{1}{n} \sum_{c=0}^{n-1} p_c \sum_{k=0}^{n-1} \omega_n^{(c-j)k} = \frac{1}{n} \sum_{c=0}^{n-1} p_c \delta_{c,j} = \frac{1}{n} n \cdot p = p \tag{2}$$

D. Fast Fourier Transform

The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm capable of computing the DFT with a time complexity of $O(n \log n)$. Many FFT algorithms exist. In this work we will formalize fast multiplication using the Cooley-Tukey FFT, its most common version.

For the purposes of this section, we assume p is a polynomial of length $2n$. Padding a polynomial with zero coefficients is a standard approach to satisfy this condition without altering the polynomial's behavior.

The FFT exploits two key properties of the n-th roots of unity:

$$(w_{2n}^j)^2 = w_n^j \tag{3}$$

$$w_{2n}^{j+n} = -w_{2n}^j \tag{4}$$

The recursive step of the FFT decomposes a polynomial p into two polynomials p_e and p_o such that p_e contains the even terms of p and p_o contains the odd terms after factoring out x. Any evaluation of $p(x)$ can then be rewritten by splitting the sum defining $p(x)$ into two sums over p_e and p_o:

$$p(x) = p_e(x^2) + x \cdot p_o(x^2) \tag{5}$$

This decomposition transforms computing the DFT of p from computing $p(\omega_{2n}^0), ..., p(w_{2n}^{2n-1})$ to computing the following lists and combining them using (5):

$$p_e((w_{2n}^0)^2), ..., p_e((w_{2n}^{2n-1})^2)$$

$$p_o((w_{2n}^0)^2), ..., p_o((w_{2n}^{2n-1})^2)$$

By (3), evaluating at $(w_{2n}^0)^2, ..., (w_{2n}^{2n-1})^2$ is equivalent to evaluating at $w_n^0, ..., w_n^{2n-1}$. Additionally, by the periodicity of the unit roots, the lists $w_n^0, ..., w_n^{n-1}$ and $w_n^n, ..., w_n^{2n-1}$ are exactly equal. Then, we can obtain the necessary evaluations by only evaluating p_e and p_o at $w_n^0, ..., w_n^{n-1}$. p_e and p_o are both of length n given that p is of length $2n$. As a result, computing this sequence is the same as computing the DFT for p_e and p_o.

If we choose p to have a length that is a power of 2 this decomposition can be applied recursively until we reach the base case, the constant polynomial of length 1.

The FFT gains speed over the direct DFT by reusing computations from the DFT of its decompositions. The symmetry in (4) can be exploited to efficiently combine the DFT of p_e and p_o. An evaluation at w_n^k of both p_e and p_o can be used to obtain evaluations at the original polynomial at both w_{2n}^k and w_{2n}^{k+n}. This allows $2n$ evaluations of the original polynomial to be calculated from only n evaluations of its decomposition. Let X be the DFT of p:

$$X_k = p_e(w_n^k) + w_n^k \cdot p_o(w_n^k) \tag{6}$$

$$X_{k+n} = p_e(w_n^k) - w_n^k \cdot p_o(w_n^k) \tag{7}$$

This portion of the FFT, combining the smaller DFT's of p_e and p_o into a larger DFT is commonly called the *butterfly* computation. Its name comes from the shape of the data-flow diagram in the radix-2 case. By recursively applying the decomposition to p_e and p_o, the problem size is divided by 2 at each step of the recursion. Combining the polynomials has time complexity $O(n)$, and there are $O(\log n)$ levels of recursion, resulting in a time complexity of $O(n \log n)$.

E. Fast Multiplication Algorithm

In this section, we introduce the Fast Multiplication Algorithm. It leverages the Fast Fourier Transform to convert between polynomial representations. Its input is two polynomials in coefficient representation. Its output is the product of the two polynomials, also in coefficient representation. Let p and q be polynomials of lengths m and n respectively. The steps of the algorithm are:

- Pad the two polynomials with zeroes to a length of $2^k \geq m + n$.
- Compute the FFT of the two polynomials to obtain the point-value representations.
- Perform pointwise multiplication of the transformed polynomials.
- Compute the inverse FFT of the product.

Let n be the maximum of the lengths of the two polynomials. The multiplication in the point-value representation has a time complexity of $O(n)$, while both the FFT and its inverse have a time complexity of $O(n \log n)$. Consequently, the overall time complexity of the Fast Multiplication Algorithm is $O(n \log n)$, representing a significant improvement over the quadratic time complexity $O(n^2)$ of the naive multiplication algorithm.

III. Coq

Coq is a proof assistant designed to develop mathematical proofs, to write formal specifications and to verify that programs are correct with respect to their specifications. It combines Gallina, a functional programming language, with the Calculus of Inductive Constructions (CIC), a type theory created by Thierry Coquand and Gérard Huet [7]. An extension of the Curry-Howard isomorphism [9], the CIC offers a common language to define programs, properties and proofs in. This allows users to define algorithms and their properties, then use Coq's proof development environment to construct a valid proof. Coq's compiler can then reduce programs and their proofs into a core language, where programs and their correctness are verified. Coq has been used often for the formalization of theorems and algorithms. Formalizations of the FFT in Coq have previously been done by Capretta in 2001, who used a powerlist structure for polynomial representation and by Théry in 2022, who used the SSReflect extension [11] [4].

A. Basic Features of Coq

Before discussing details about recursion and the main datatypes relevant to our proof in Coq, it is important to introduce some fundamental features of the language.

`Terms` are the basic expressions in Coq. They can represent mathematical expressions, propositions and proofs, but also executable programs and program types. The **`Definition`** command associates a name with a term. In our work, we will use it to name data types, such as polynomials, and functions, such as the DFT.

Assertions can be made with the **`Lemma`** command. It lets users state a proposition for which the proof is then built interactively using Coq's proof environment. Assumptions can be made with the **`Axiom`** command. Like the **`Lemma`** command, **`Axiom`** lets users state a proposition. This proposition is assumed to be correct and Coq will not require users to provide a proof of its correctness. Proven lemmas and axioms can be used during the development of proofs for other lemmas.

Although we do not use it directly, it is useful to discuss the **`Inductive`** command. This command defines inductive types and their constructors. Coq can generate *induction principles* for these types, which allows for structural recursion and induction over types defined using the command. The `list` and `nat` types that this work uses are defined in this way.

The **`Fixpoint`** lets users define recursive functions by pattern matching over inductive objects.

B. Tactics in Coq

In our formalization, we leveraged several fundamental Coq tactics to structure our proofs. In this section, we will highlight some important Coq tactics used in the formalization. This will help to understand the proof in IV-E. The `intros` tactic is used to introduce assumptions or variables into the proof context, allowing us to work with them explicitly.`induction` is crucial for performing proofs by induction, enabling us to handle both base and inductive cases systematically. The `simpl` tactic simplifies expressions by reducing them to their more basic forms, making the proof easier to manage. `apply` is employed to utilize existing theorems or hypotheses to prove the current goal, while `rewrite` allows us to substitute equals for equals within a goal, based on a given equation or lemma. `replace` provides a way to substitute one term with another equivalent term, facilitating a clearer or more manageable proof direction. `assert` is used to introduce intermediate lemmas or statements within a proof, breaking down complex goals into simpler sub-goals. Finally, the `auto` tactic is a powerful tool that automates the proof of straightforward goals by applying a combination of known lemmas and hypotheses, thus reducing manual effort. These tactics together form a robust toolkit that enabled the rigorous development and verification of our formal proofs in Coq.

C. Data Types

In this work, we will operate on the natural, list and real data types from the Coq standard library, as well as the implementation of complex numbers from the Coquelicot real analysis library [3].

Natural numbers are represented by the `nat` type. As a simple inductive type, The type nat is defined as the least `Set` containing `O`, which is 0, and closed by the `S` constructor, which is the natural number that follows. As an example, `S(O)` is the natural number 1.

Similarly, the `list` type is also inductive. The type is defined as the least `Set` containing `nil`, the empty list and closed by the "`::`" constructor, which appends an element to the start of the list. The list type can only contain elements that are of the same type. As an example, `O::nil` consists of the list [0], where 0 is a natural number.

The `R` and `C` types represent real and complex numbers respectively. `R` exists in the Coq standard library. Complex numbers are implemented in the Coquelicot library. The type consists of a set of two numbers: an imaginary part and a real part. The numbers for the two parts are of the `R` datatype.

D. Recursion in Coq

In the Calculus of Inductive Constructions (CIC), which underlies Coq, recursive functions are required to terminate. When defining a structurally recursive function through Coq's **`Fixpoint`** command, it pattern matches over an inductively defined object. Internally, Coq's uses guard predicates, which ensure recursive calls are made on arguments that are structurally smaller at each step.

However, when one wants to define recursion on elements that are not inductive or by an operation like halfing natural numbers, Coq's termination checker cannot ensure that the function is applied on a structurally smaller element at each step.

While it is possible to define well-founded recursion in Coq by augmenting a nonstructurally recursive function with a well-founded ordering between inputs and a proof that the input is decreasing in accordance with this ordering, it is more involved than defining recursion through Coq's **`Fixpoint`** functionality. Furthermore, when the code is extracted, the proof of well-foundedness cannot always be erased, potentially resulting in asymptotically slow functions [1].

IV. Formalization of the Multiplication Algorithm in Coq

This section presents the formalization of the polynomial multiplication algorithm. We first define basic operations and conversions between sparse and dense polynomial representations, then extend to complex polynomials. Following this, we define and prove properties of the roots of unity. Then we define and prove correctness of the (i)DFT and the (i)FFT. Finally, we define and prove the fast multiplication algorithm. For brevity, in this section we will give definitions of types and functions as well as statements of important results, but generally omit proofs as an accepted proof in Coq is guaranteed to be correct.

A. Polynomial Representation

The Verified Calculus library represents the polynomials underlying its polynomial models as tuples of coefficients paired with their degree. These tuples may omit zero coefficients for efficiency. However, this representation complicates reasoning about the degree of polynomials, as the degree is not directly tied to the length of the coefficient list. As a result, inductive arguments over the list structure can provide no guarantee of their effect on the polynomial's degree. Furthermore, when we make an inductive argument over the degree of a polynomial in this representation, it is not guaranteed that there exists a coefficient for all natural numbers between 0 and the degree. To address this issue, it is advantageous to represent polynomials as dense lists of coefficients. In this dense representation, the length of the coefficient list is directly related to the polynomial's degree plus one. This correlation simplifies reasoning about polynomial degrees and the effects of polynomial operations on their degree.

We define a dense polynomial type over reals:

```
Definition dense_R_poly := list R.
```

We denote the evaluation of a dense polynomial with real coefficients with:

```
Definition dense_eval(p:dense_R_poly) (x:R).
```

Besides evaluation, we will define basic operations on dense polynomials. We define adding dense real polynomials and multiplying dense polynomials by a scalar. Their signatures are:

```
Fixpoint add_dense_R_poly (p1 p2: dense_R_poly): dense_R_poly
Fixpoint scalar_mult_R (a : R) (p : dense_R_poly) : dense_R_poly
```

As we shall prove correctness of fast polynomial multiplication by its equivalence to naive polynomial multiplication, we highlight the definition of naive multiplication and its proof of correctness. Naive multiplication is defined recursively over the structure of polynomial `p1`.

```
Fixpoint pmul_R (p1 p2 : dense_R_poly) :
                         dense_R_poly :=
  match p1 with
  | nil => nil
  | h1 :: t1 => add_dense_R_poly
                (scalar_mult_R h1 p2)
                (0 :: pmul_R t1  p2)
  end.
```

The correctness of this definition is proven by the following lemma:

```
Lemma pmul_correct_R: forall p1 p2 x,
    (Pdense_eval p1 x) * (Pdense_eval p2 x) =
    Pdense_eval (pmul_R p1 p2) x.
```

We can reason about specific coefficients in this dense representation by using the standard Coq library's `nth` function.:

```
Fixpoint nth (n:nat) (l:list A) (default:A)
{struct l} : A :=
    match n, l with
      | O, x :: l' => x
      | O, other => default
      | S m, nil => default
      | S m, x :: t => nth m t default
    end.
```

This function takes a natural number and returns the list element at that index. `nth` returns a default element whenever the natural number is greater than or equal to the length of the list. In the context of this thesis, we will only require this default element to be complex or real zero. To abstract this default element and increase legibility we introduce a notation:

```
Notation "p `_ j" := (nth j p 0).
```

For lists of equal length, we can infer equality through each element being equivalent when this function is applied:

```
Lemma nth_eq: forall l1 l2,
    length l1 = length l2 ->
      (forall (a : nat), l1`_a = l2`_a) ->
      l1 = l2.
```

As the roots of unity are complex numbers, calculating the DFT requires us to evaluate polynomials on a complex data type. Furthermore, the point-value representation that results from the DFT will then have complex coefficients as well. The main data type we will operate on will be complex polynomials for this reason. We define complex polynomials as a list of the datatype C:

```
Definition dense_poly := list C.
```

Evaluation of complex polynomials at a complex value is defined recursively. In this function, `(d:nat)` is a variable required to track the degree of the current coefficient at each step of the recursion. When evaluating a polynomial, it should always start at `0`. We define the helper function:

```
Fixpoint complex_eval'
(d:nat)(pd: dense_poly)(c:C): C :=
 :match pd with
  | nil => 0
  | cd::pSd => cd * (c^d) +
       complex_eval' (S d) pSd c
  end.
```

Then, we can define evaluation without the `d` variable as a call of this helper function:

```
Definition complex_eval (p:dense_poly)(c:C) :
                                        C :=
  complex_eval' 0 p c.
```

To increase legibility and abstract this implementation detail, we introduce a notation for evaluating a complex polynomial:

```
Notation "p [ x ]" := (complex_eval p x)
```

The following function allows us to transform between real and complex coefficient representations by casting each real coefficient into its complex equivalent:

```
Fixpoint dense_poly_RtoC(p:dense_poly):
                          dense_poly:=
    match p with
    | nil => nil
    | fn:p' => RtoC(fn) :: p_RtoC(p').
```

Although the intermediate results of the fast multiplication of two polynomials with real coefficients might be complex, the resulting polynomial is not. To prove this, we need to reason about the equivalence of real and complex polynomials. We define equivalence between them as evaluating to the same value for every real number x:

```
Definition poly_eqv_R_C(p1:dense_R_poly)
                (p2:dense_poly) :=
    forall x:R,
    RtoC(Pdense_eval p1 x) =
    p2(complex_eval p1 x).
```

Using this definition we prove the correctness of `p_RtoC`:

```
Lemma dense_poly_RtoC_eqv: forall p x,
    RtoC (Pdense_eval p x) =
    complex_eval (dense_poly_RtoC p) (RtoC x).
```

Having defined a notion of equality between complex and real polynomials, we can now reason about what a correct fast multiplication algorithm must satisfy. In this work, we will formalize a fast multiplication algorithm for dense complex polynomials, and prove its equivalence using the `poly_eqv_R_C` definition.

B. Roots of Unity

Before we proceed to define the DFT and FFT, we require a definition of the principal n-th roots of unity and their properties. We do so using De Moivre's formula, where `PI` is the constant from the Reals library and `Ci` is the imaginary component of the `C` data type.

```
Definition root_of_unity(n:nat): C :=
  cos((2*PI)/n) + Ci * sin ((2*PI)/n).
```

In this thesis, we leverage certain properties of unit roots that are fundamental to the proof of the FFT and its inverse. Specifically, we prove the following lemmas regarding unit roots:

```
Lemma unit_root_squares: forall n k,
    n <> O ->
```

```
((root_of_unity (2*n))^k)^2 =
(root_of_unity n)^k.
```

This lemma states property (3) in II-D. This property follows from the periodicity and symmetry of unit roots in the complex plane. The $2n$-th unit roots are simply the n-th unit roots with additional intermediate points. Squaring these roots effectively skips the intermediate points, resulting in the n-th unit roots.

```
Lemma pow_n_to_1: forall n,
    root_of_unity(2^n)^(2^n) = 1.
```

Mathematically, this lemma follows from the definition of the roots of unity and holds for any `n`, not just powers of 2. However, for the purposes of this work we will only need the unit root properties to hold for powers of 2 and restricting the lemma to powers of 2 simplifies the proof to a simple inductive argument over `n`. Next, we prove a lemma concerning a symmetric property that powers of the unit roots possess:

```
Lemma unit_root_symmetry: forall n k,
    root_of_unity (2^S n)^(k+(2^n)) =
    - (root_of_unity (2^ S n))^k.
```

This lemma expresses property (4) in II-D. It states that shifting any power of a 2^{Sn}-th root of unity by n results in its negation. It follows from the symmetric distribution of the unit roots around the unit circle. This lemma holds more generally for any even-numbered unit root, but similarly to `pow_n_to_1`, we only need it to hold for powers of 2.

As the implementation of powers in the `R` datatype as well as the `C` data type only takes natural numbers as the power, we define the following lemma to rewrite the inverse unit root as a positive power:

```
Lemma inverse_unit_root: forall n,
    root_of_unity (2 ^ n) ^ (2 ^ n - 1) =
    / root_of_unity (2 ^ n).
```

This lemma follows from the periodicity of the unit roots.

These four properties of the unit roots are all proven in Coq. One assumption has been made regarding summations over the roots of unity. The following assumption expresses property (1). It makes use of the `sum_n` function, which we shall discuss in IV-F :

```
Axiom unit_roots_sum_to_zero: forall n k,
    Nat.le 1 n ->
    sum_n (
    fun i : nat =>
        root_of_unity (2 ^ n) ^ (k * i) )
        (2 ^ n - 1)      =
    RtoC 0.
```

This assumption is relevant to the proof of the iDFT.

Finally, we introduce a notation for the roots of unity that we will use from here on:

```
Notation "\w_ n " := (root_of_unity n).
```

C. Discrete Fourier Transform

We proceed to define the DFT and prove its correctness. To facilitate this, we introduce the `eval\_powers` function, which takes a complex number `x`, a natural number `n` and a dense complex polynomial p. The function returns a polynomial whose coefficients are evaluations of polynomial p at the points $x^0, x^1, ..., x^{n-1}$.

```
Fixpoint eval_powers (w:C)(n:nat)(p:dense_poly):
                                    dense_poly :=
  match n with
  | O => nil
  | S(n') => eval_powers w n' p ++
             p[w^n'] :: nil
  end.
```

The reason for use of this `eval_powers` function is that it allows us to define and leverage a series of auxiliary lemmas for both the DFT and iDFT, which we shall define later, in the proof of our main results. Since both DFT and iDFT involve evaluating a polynomial at various points, the `evals` function enables us to avoid redundant work. By defining the DFT and iDFT in terms of `evals`, we can use the same auxiliary lemmas for both transforms. The correctness of `eval_powers` is established for each element by the following lemma:

```
Lemma eval_powers_correct: forall x n a p,
    Nat.lt a n ->
    (eval_powers x n p)`_a = p[x^a].
```

Its proof is a straightforward inductive argument over the polynomial p. Finally, we define the DFT as calling the `eval_powers` at the n-th root of unity and its length.

```
Definition DFT(p: dense_poly): list C:=
    let n := length p in
    eval_powers(\w_n) (n) p.
```

Its correctness for each element follows directly from the correctness of `eval_powers`.

D. Defining the Fast Fourier Transform

In this section, we cover the definition and proof of the FFT. As a divide-and-conquer algorithm, FFT consists of two main parts: breaking down the problem into smaller, more manageable subproblems, and then combining these subproblems to find a solution to the original problem. We will discuss the design choices made in defining the algorithm for both of these phases and then present our definition of the FFT.

In the FFT, recursive calls are made on even and odd parts of our polynomials. To facilitate this, we need to construct these parts. This can be done recursively over the list of coefficients, incrementing a natural number `d` from 0 with each call to track the degree of each element. Elements are added to the list only when `d` is even or odd, respectively. We define the helper functions:

```
Fixpoint even_poly'(d:nat)(p:dense_poly):
                                dense_poly :=
 match p with
  | nil => nil
  | a1::p' => if Nat.even(d) then
              a1::even_poly' (S d) p'
              else even_poly'(S d) p'
  end.

Fixpoint odd_poly'(d:nat)(p:dense_poly):
```

```
                                dense_poly :=
  match p with
  | nil => nil
  | a1::p' => if Nat.odd(d) then
              a1::odd_poly'(S d) p'
              else odd_poly'(S d) p'
end.
```

We can then define the even and odd decompositions as calls of these helper functions:

```
Definition even_poly (p:dense_poly) :=
    even_poly' O p.
Definition odd_poly (p:dense_poly)  :=
    odd_poly' O p.
```

As before, we introduce notations for legibility:

```
Notation "\even_ p " := (even_poly p).
Notation "\odd_ p "  := (odd_poly  p).
```

To circumvent having to prove the termination of recursion through this decomposition, we can write the FFT as a structural recursive function. We define the recursive FFT as an argument of the length (its degree+1) of the polynomial and its decompositions. The length of the decompositions is half that of the original polynomial, given that the original polynomial is of an even length. By defining the length of the original polynomial p to be 2^n and then writing the FFT through structural recursion on n, we can guarantee both termination and that at each recursive step until the base case of $n = 0$, the length of the polynomial is even. This allows us to define the FFT as a Fixpoint. The following lemmas let us equate 2^n to the length of the current polynomial at each recursive step:

```
Lemma even_half_len: forall p n,
Nat.le 1 (n)%nat ->
length p = (2*n)%nat ->
length(\even_p) = n.
Lemma odd_half_len: forall p n,
Nat.le 1 (n)%nat ->
length p = (2*n)%nat ->
 length(\odd_p) = n.
```

Having proven termination of the decomposition, we proceed to prove correctness of the butterfly operation, which combines the DFT's of the even and odd decompositions into the DFT of the original polynomial. Let p be the original polynomial, let X be its DFT, let y_e be the DFT of the even decomposition and let y_o be the DFT of the odd decomposition. The butterfly operation computes the DFT of p as follows:

Algorithm 1 FFT Butterfly computation

1: **for** $j \in \text{range}(2^{n-1})$ **do**
2: $\quad y[j] \leftarrow y_e[j] + \omega^j y_o[j]$
3: $\quad y[j + 2^{n-1}] \leftarrow y_e[j] - \omega^j y_o[j]$
4: **end for**

In Coq, accessing indices of a list like this requires us to provide guarantees of the length of the list at each assignment and filling the list with dummy coefficients before assigning them to the actual coefficients. Instead, we split the loop into two functions that construct separate lists. The function `make_left` constructs the list of DFT elements whose indices are in the range 0 and $n - 1$. The function `make_right` constructs the list of DFT elements whose indices range between n and $2n-1$. Both functions are defined over the degree of the decompositions $n - 1$ recursively.

```
Fixpoint make_right (y_e y_o: list C)
(n:nat)(w: C): list C :=
  match n with
  |O => nil
  |S(n') => make_right y_e y_o n' w  ++
        (((y_e`_n' - w^n' * y_o`_n')) :: nil)
end.

Fixpoint make_left (y_e y_o: list C)
                   (n:nat)(w: C): list C :=
  match n with
  |O => nil
  |S(n') => make_left y_e y_o n' w  ++
         ((y_e`_n' + w^n' * y_o`_n') :: nil)
end.
```

The final DFT is constructed by merging these two lists. Since the last index of the first list is exactly one less than starting index of the second list, concatenation suffices:

```
Definition butterfly (y_e y_o: dense_poly)
            (w: C): dense_poly    :=
 let n       := length(y_e) in
 let l1      := make_left y_e y_o (n) w in
 let l2      := make_right y_e y_o (n) w in
      l1 ++ l2.
```

With the decomposition and combination steps defined, we can now present the FFT function. The recursive calls are made on the even and odd decompositions, and the result of their FFT is combined using the butterfly function:

```
Fixpoint fft(m:nat)(p:list C)(w:C):list C :=
  match n with
  | O => p
  | S(n')      =>
    let y_e := fft(m')(\even_p)(w^2%nat) in
     et y_o := fft(m')(\odd_p)(w^2%nat) in
                 butterfly y_e y_o w
  end.
```

E. Proving the Correctness of the FFT

We prove correctness of the FFT by proving its equivalence to the DFT. The equivalence is demonstrated through an inductive proof that relies on several auxiliary lemmas. This section outlines the key steps and lemmas involved in the proof, providing an overview of the logical structure and main ideas.

The core of the proof involves demonstrating that the FFT correctly computes the DFT by combining results from its decomposition. The following lemma establishes the relationship between the evaluations of the original polynomial and its even and odd decompositions.

```
Lemma even_and_odd: forall p x,
    \even_p[x^2] + x * \odd_p[x^2] = p[x].
```

The proof for this lemma is a straightforward inductive argument over p after proving an auxiliary lemma that shows incrementing the starting degree swaps the even and odd polynomial lists.

The following lemmas prove the relations

$$\forall p, j, n : p(w^j_{2*n}) = p_e(w^j_n) + w^j_{2*n} \cdot p_o(w^j_n) \tag{8}$$

$$\forall p, j, n : p(w_{2*n}^{j+n}) = p_e(w_n^{j+n}) + w_{2*n}^k \cdot p_o(w_n^{j+n}) \quad (9)$$

Which are the basis of the inductive step.

```
Lemma FFT_inductive_step_even: forall
(p: dense_cpoly)(j n: nat),
    Nat.le 1 n -> p[(\w_(2*n)^j)] =
    \even_p [(\w_n^j)] +
    \w_(2*n)^j * \odd_p [\w_n ^j].

Lemma FFT_inductive_step_odd: forall
p: dense_cpoly)(j n: nat),
    p [\w_(2^ S n)^(j+ (2^n))] =
    \even_p [\w_(2^n)^j] -
    (\w_(2^ S n) ^j) * (\odd_p)[\w_(2^n)^j].
```

They follow from the `even_and_odd`, `unit_root_squares` and `unit_root_symmetry` lemmas.

To prove correctness of the butterfly function, we first prove that the lists it concatenates correlate to their respective parts of the DFT. The following lemmas states that when the `(n:nat)` parameter is equal to the length of the decompositions and the polynomial parameters are the DFT of the even and odd decompositions of p respectively, both `make_left` and `make_right` correctly compute their corresponding element of the DFT of p. Note that while the correctness lemma of `make_left` is more general, the correctness lemma of `make_right` is restricted to powers of 2 as we restricted the symmetry property to powers of 2 as well. For the purposes of this thesis this is sufficient.

```
Lemma make_left_correct: forall m a y_e y_o p,
    Nat.le 1 m -> Nat.lt a (m) ->
    length p = (2*m)%nat ->
    y_e = DFT (\even_p) ->
    y_o = DFT (\odd_p) ->
    (make_left y_e y_o (m) (\w_(2*m)))`_a  =
    (DFT p)`_a.
Lemma make_right_correct: forall m a y_e y_o p,
    Nat.lt a (2^m) ->
    length p = (2^S m)%nat ->
    y_e = DFT(\even_p) ->
    y_o = DFT(\odd_p) ->
    (make_right
        y_e y_o (2^m)%nat (\w_(2^S m)))`_a  =
    (DFT p)`_(a+(2^m)).
```

To prove these lemmas, we define an auxiliary lemma that lets us rewrite any element that results from `make_left` and `make_right` as a function of elements from input polynomials `y_e` and `y_o`. This allows us to rewrite the left-hand side of the equality. The `evals_correct` lemma lets us rewrite the right hand side of the equality as an evaluation of polynomial p. These two rewrites result in (8) for `make_left` and (9) for `make_right`. Using these results, we can verify the correctness of the `Butterfly` function:

```
Lemma butterfly_correct: forall n y_e y_o p,
    length p = (2^S n)%nat ->
    y_e = DFT(\even_p) ->
    y_o = DFT(\odd_p) ->
    butterfly y_e y_o (\w_(2^S n)) = DFT p.
```

To prove this lemma, we show that each index of the two lists is equal. We can do so by splitting the goal into indices less than `n` and indices greater than or equal to it. Both cases simplify into the `make_left_correct` and `make_right_correct` lemmas respectively. Applying `nth_eq` lets us reach our goal.

Finally, we establish the overall correctness of the FFT algorithm by showing that, for any polynomial it is equivalent to the DFT:

```
Lemma fft_correct: forall
    (m:nat) (p: dense_cpoly),
    length p = (2^m)%nat ->
    fft m p (\w_(2^m%nat)) = DFT p.
```

Its proof is inductive over m. The base case is a polynomial of length 1, which is a constant polynomial. The result of evaluating a constant polynomial will always be its single coefficient. The FFT function that we defined returns polynomial p in this case, which is easily proven correct.

In the inductive step, as we have already proven the correctness of the `Butterfly` function, all that is left is to prove that it is provided the right parameters. The inductive hypothesis lets us rewrite the recursive calls as correct DFTs. After providing some guarantees on the length of the decompositions, the proof is complete. To give an idea of the structure of proofs in Coq, the proof of correctness of the FFT in Coq is given here:

```
Lemma fft_correct: forall n p,
length p = (2^m)%nat ->
fft m p (root_of_unity(2^m%nat)) = DFT p.
Proof.
    induction m as [IHm |].
  - intros p H.
    rewrite -> DFT_constant
        by (simpl in *; auto).
    simpl in *. reflexivity.
  - intros p H.
    simpl.
(* even poly degree *)
    assert
    (length (\even_p) = (2^m)%nat) as
                          length_even.
      apply even_half_len in H.
      apply H.
      apply pow_le_1.
(* odd poly degree *)
    assert
    (length (\odd_p) = (2^m)%nat) as
                           length_odd.
      apply odd_half_len.
      apply pow_le_1.
      apply H.

    assert(root_of_unity (2 ^ m + (2 ^ m + 0))
    *(root_of_unity (2 ^ m + (2 ^ m + 0)) * 1%R)
                       = root_of_unity (2 ^ m))
      as unit_root_rewrite
        by (apply unit_root_squares).
      rewrite -> unit_root_rewrite.
      repeat rewrite -> IHm by auto.
      replace(\w_ (2 ^ m + (2 ^ m + 0)))
      with (\w_(2^S m)) by auto.
      rewrite -> butterfly_correct
      with (p:= p) by auto.
      reflexivity.
Qed.
```

F. Inverse DFT and iFFT

In this section, we define and prove the correctness of the iDFT and the iFFT. The proof of correctness for the iDFT requires us manipulate summation. We shall redefine polynomial evaluation as a summation and prove the correctness of some manipulations of nested sums to reach our goal.
Following this, we will extend the proof of correctness of the FFT to include odd powers of the primitive n-th roots of unity. Finally, we will leverage the extended proof of correctness of the FFT to prove that the iFFT and iDFT are equivalent.

Similarly to the DFT, the iDFT can be defined as a call of `eval_powers` with the multiplicative inverse of the n-th root of unity and the length of the polynomial as its parameters. This yields the list of evaluations of polynomial p at the points:

$$w^0, w^{-1}, ..., x^{-(n-1)}$$

This list of evaluations correctly inverses the DFT's transformation up to a scaling term of $1/N$ where N is the length of the polynomial. The scaling term is implemented through the standard Coq library's `map` function, which applies any function to each element of a list. In this case, the function is $f(x) = n^{-1} * x$:

```
Definition iDFT(p: dense_poly) :=
 let n := length p in
  map(fun x => /n * x)
    ((evals(/ \w_n)) (n)%nat p).
```

Our alternative definition for polynomial evaluation makes use of Coquelicot's `sum_n` function. The `sum_n` function applies the function in its first parameter to all values from 0 until its second parameter and sums them. We can express polynomial evaluation with it as follows:

```
Definition sum_eval(x:C)(p:dense_cpoly) :=
    sum_n(fun c => p`_c*x^c)(length p-1).
```

An inductive proof over p can prove its equivalence to our original evaluation function:

```
Lemma sum_ev_eq: forall x p,
    sum_eval x p = p[x].
```

This alternate evaluation function lets us reason more easily about evaluating the result of another evaluation. Specifically, it helps us prove the following lemma that is necessary for proving the correctness of the iDFT. It states that for each element in our list of coefficients, applying the DFT and then applying the `evals` function at the multiplicative inverse of the n-th unit root to the DFT's result yields our original polynomial up to a scaling term of its length:

```
Lemma evals_inversed: forall j n p,
 (n)%nat = length p -> Nat.lt j (n)%nat ->
 (evals (/ (\w_(n)%nat))(n)%nat
        (DFT p))`_j
    = n * p`_j.
```

By correctness of the `evals` function, the equality can be rewritten to

```
(DFT p)[(/ \w_n) ^ j] = n * p`_j
```

By rewriting evaluation as a summation and proving some auxiliary lemmas about transforming nested summations we can perform the transformations of (**??**) to factor out the coefficients.
By axiom `unit_roots_sum_to_zero` we can finish the proof of this lemma. Finally, we can prove the correctness of the iDFT:

```
Lemma iDFT_correct: forall p,
    p <> nil ->
    iDFT (DFT p) = p.
```

The proof leverages the `evals_inversed` lemma and auxiliary lemmas about mappings and summation manipulation.
Next, we define the iFFT:

```
Definition ifft(n:nat)(p:list C):list C :=
  let w:= /(\w_(2^n)) in
  map(fun x => /(2^n)%nat * x) (fft n p w).
```

The proof of equivalence between the iDFT and iFFT closely follows the FFT proof. We extend correctness of the `fft` to any odd power of the `roots_of_unity`:

```
Lemma fft_correct_odd_pows: forall n p k e,
    length p = (2^n)%nat -> k = S(2*e) ->
    fft n p (root_of_unity(2^n%nat)^k) =
    eval_powers(\w_(2^n)^(k))(2^n-1)%nat p.
```

Leveraging this extension, we can formulate and prove the equivalence of the iFFT and iDFT:

```
Lemma ifft_iDFT: forall p n,
    length p = (2^n)%nat ->
    ifft n p = iDFT p.
```

G. Fast Multiplication Algorithm

In this section, we define the fast multiplication algorithm using FFT, explain its components and provide a proof of its correctness. Before we are able to define our multiplication algorithm, we need to define pointwise multiplication. This is done recursively over the shared length of the two polynomials to multiply. We define the helper function:

```
Fixpoint pointwise_mul'(p1 p2:list C)
                                (n:nat):=
  match n with
  | O      => (p1`_0)*(p2`_0)::nil
  | S (n') => pointwise_mul' p1 p2 n' ++
           (p1`_n * p2`_n ::nil)
end.
```

Next, we verify that performing pointwise multiplication of a point-value representation of two polynomials p_1 and p_2 is equal to the point-value representation of the naive multiplication of p_1 and p_2.

```
Lemma mul_evals: forall p1 p2 w n,
    pointwise_mul'(evals w n p1)
                  (evals w n p2) n =
    evals w n (pmul p1 p2).
```

Similar to how the `Butterfly_correct` proof was structured, to prove this lemma, we prove equivalence at each possible index first by an inductive argument, then generalize to equality of the lists.
As the fast multiplication algorithm requires padding of the input polynomials, we define a padding function which pads a polynomial p with n zeroes at the end:

```
Definition pad(p:dense_cpoly) (n:nat) :=
    p ++ repeat (RtoC 0) n.
```

The following lemma shows that padding does not effect the polynomial's evaluation at any point:

```
Lemma eval_pad: forall p n x,
    p[x] = (pad p n)[x].
```

Now that we have defined the FFT, iFFT and pointwise multiplication, we can define the fast multiplication algorithm:

```
Definition fast_pmul
(p1 p2: dense_cpoly)(n:nat):=
    ifft (S n) (
    pointwise_mul'
        (fft (S n)
           (pad p1 (2^n)%nat)
           (\w_(2^(S n))%nat) )
        (fft (S n)
           (pad p2 (2^n)%nat)
           (\w_(2^(S n))%nat) )
      (2^(S n)-1)%nat
        ).
```

The algorithm takes as input two polynomials, both of length 2^n. It pads these polynomials with zeroes until they are of length 2^{n+1}. It performs the FFT on the two padded polynomials and multiplies them pointwise, resulting in a single polynomial (the point-value representation of their product). Finally, it performs the iFFT to return to a coefficient representation. This is its output.

To prove the correctness of the fast multiplication algorithm, we show that for any polynomials $p1$ and $p2$, the algorithm's output is equivalent to the output of the naive multiplication algorithm:

```
Lemma fast_pmul_correct: forall p1 p2 n x,
  length p1 = (2^n)%nat ->
  length p1 = length p2 ->
  (fast_pmul p1 p2 n)[x] =
  (pmul p1 p2)          [x].
```

Having already proven the correctness of most of its components, the proof of this lemma is straightforward and consists of rewriting the left-hand side of the equality into the right using correctness of the FFT, iFFT and the `mul_evals` lemma. After reasoning that padding the polynomial with zeroes does not affect the evaluation of it, or its naive multiple, we are done.

V. Discussion

In this work, we have formalized and verified a fast polynomial multiplication algorithm using the Coq proof assistant. This formalization includes the definition of different polynomial representations, their evaluation and conversion between their representations. We defined and proved properties of the complex roots of unity, building the proof on the assumption that the n-th roots of unity sum to zero over the range $0...n-1$. As a result, we rigorously defined and proved correctness of the DFT and FFT. We also defined the iDFT and iFFT. While the correctness of the iDFT is contingent on the assumption that the n-th roots of unity sum to zero, the equivalence between the iFFT and iDFT was proven independently of this assumption. We implemented the fast polynomial multiplication algorithm which leverages the FFT and iFFT to achieve a significant reduction in computational complexity compared to classical polynomial multiplication methods.

VI. Conclusion

In this work, we have worked towards developing a fast multiplication algorithm for polynomials with rigorous error bounds by formalizing a fast multiplication algorithm for polynomials with real coefficients. We set out to set out to address three primary research questions.

RQ1: Is it possible to implement and prove the correctness of an algorithm for fast polynomial multiplication in floating point arithmetic using Coq?: Yes, it is possible to implement and prove the correctness of an algorithm for fast polynomial multiplication using Coq. We have successfully formalized the FFT and DFT. Leveraging the assumption of one property of the roots of unity, we also defined and formalized the iDFT, iFFT and the fast multiplication algorithm. Our implementation and proofs focus on polynomials with real coefficients, laying the groundwork for future extensions to floating-point arithmetic. The correctness of the fast multiplication algorithm was demonstrated through verification using the Coq proof assistant.

RQ2: How do different approaches of polynomial multiplication differ in terms of both algorithmic complexity and the difficulty of proving correctness?: We explored naive and fast polynomial multiplication approaches, highlighting the significant difference in time complexity. Proving the correctness of a fast multiplication algorithm was more complex due to the need to formally verify the FFT which required defining and reasoning about roots of unity and complex numbers.

A. Future Work

Future work should focus on defining and verifying the final property of the complex roots of unity. This would finalize the proof of correctness of the iDFT and therefore the entire fast multiplication algorithm. Additionally, extending our verification efforts to floating-point arithmetic will enhance the practical applicability of our work. Another area worth researching is formalization of time complexity in type theory, which would allow for proofs on the time complexity of the fast multiplication algorithm developed in this work.

References

[1] Pedro Abreu et al. "A Type-Based Approach to Divide-and-Conquer Recursion in Coq". In: *Proc. ACM Program. Lang.* 7 (2023), pp. 5–6. DOI: 10.1145/3571196.

[2] Luca Benvenuti et al. "Reachability computation for hybrid systems with Ariadne". In: *Proceedings of the 17th World Congress, The International Federation of Automatic Control (IFAC)*. IFAC. Seoul, Korea, July 6-11, 2008.

[3] S. Boldo, C. Lelay, and G. Melquiond. “Coquelicot: A User-Friendly Library of Real Analysis for Coq”. In: *Mathematics in Computer Science* 9 (2015), pp. 41–62. DOI: 10.1007/s11786-014-0181-1.

[4] Venanzio Capretta. “Certifying the Fast Fourier Transform with Coq”. In: *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*. 2001. DOI: 10.1007/3-540-44755-5_12.

[5] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Cham, Switzerland: Springer International Publishing, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[6] James W. Cooley and John W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. DOI: 10.2307/2003354.

[7] Thierry Coquand and Gérard Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2-3 (1986), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3.

[8] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009, pp. 898–905. ISBN: 978-0262033848.

[9] W. A. Howard. “The formulae-as-types notion of construction”. In: (1980).

[10] A. Karatsuba and Yu. Ofman. “Multiplication of many-digital numbers by automatic computers”. In: *Dokl. Akad. Nauk SSSR* 145.2 (1962), pp. 293–294.

[11] Laurent Théry. *A Formalisation of a Fast Fourier Transform*. Tech. rep. Sophia-Antipolis, France: INRIA.