THESIS

TYPED SYNTHESIS OF FAST MULTIPLICATION ALGORITHMS FOR POST-QUANTUM CRYPTOGRAPHY

Submitted by

William Scarbro

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2024

Master's Committee:

Advisor: Sanjay Rajopadhye

Jedidiah McClerg
Jeffrey Achter

ABSTRACT

TYPED SYNTHESIS OF FAST MULTIPLICATION ALGORITHMS FOR POST-QUANTUM CRYPTOGRAPHY

Multiplication over polynomial rings is a time consuming operation in many post-quantum cryptosystems. State-of-the-art implementations of multiplication for these cryptosystems have been developed by hand using an algebraic framework. A similar class of algorithms, based on the Discrete Fourier Transform, have been optimized across a variety of platforms using program synthesis. We demonstrate how the algebraic framework used to describe fast multiplication algorithms can be used in program synthesis. Specifically, we extend and then abstract this framework for use in program synthesis, allowing AI search techniques to find novel, high performance implementations of polynomial ring multiplication across platforms.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Post-quantum cryptosystems which are variants of the Ring Learning With Errors (RLWE) [22] problem rely heavily on operations in polynomial rings, the most expensive of which is multiplication. Any cryptographic standard will need an efficient implementation across a variety of platforms. The goal of this work is to develop program synthesis tools which will automatically optimize polynomial ring multiplication in RLWE cryptosystems for a variety of computational contexts.

While there are several algorithms for performing fast polynomial multiplication, we focus our attention on an approach which uses the Number Theoretic Transformation (NTT) because it has the best asymptotic computational complexity.

When developing algorithms using program synthesis, the choice of a Domain Specific Language (DSL) determines the space of possible implementations as well as the reasoning used to search for equivalent programs. We judge that a DSL which is most similar to the algebraic framework used to study polynomial ring structures is the most effective. This idea is supported by a state-of-the-art implementation of the NTT [25] which was manually developed using an algebraic framework for describing fast multiplication algorithms [3]. We refine this algebraic framework for program synthesis to create a language which is specialized for polynomial ring multiplication. That is to say, each program within this language corresponds to a valid implementation of polynomial ring multiplication.

Our choice of DSL varies significantly from those used to synthesize similar algorithms. One such algorithm, the Discrete Fourier Transformation (DFT) on complex numbers, is a close cousin of the NTT. In the digital signal processing library Spiral [11], the authors describe rewrites of the complex valued DFT using a language based on interpreting the DFT as a matrix vector product. While the DSL we use looks very different from the one used by Spiral, our goal was to capture the same recursive structure in our DSL as is exposed in Spiral's language. Spiral has clearly

demonstrated that program synthesis as an effective approach for optimizing the DFT. We show that a similar program synthesis method is effective for optimizing fast multiplication.

The most significant contribution of this work is incorporating the recursive structure used by Spiral for DFT synthesis into the algebraic framework describing fast multiplication. This produces a system which can describe a variety of implementations for fast-multiplication algorithms used in RLWE cryptosystems. In addition, we show how manual optimizations made to the NTT for use in RLWE cryptosystems can be automatically incorporated into this system. For example, our analysis identifies specific steps within NTT algorithms which produce unnecessary permutations in the codomain and may be trivially omitted.

Finally, we combine the program synthesis engine with AI search techniques to facilitate the discovery of efficient implementations. To expose this system to automated optimization, we develop a suggestion based search algorithm for expanding one implementation of polynomial multiplication into a neighborhood of similar algorithms.

We evaluate the efficacy of the program synthesis approach by modeling how algorithm design influences performance. To this end, several compilation pathways are used to isolate the effect of individual algorithmic and compiler optimizations. A model of execution time is then used to study the interaction of compiler optimizations and algorithmic specification with the final execution time.

Chapter 2

Background

2.1 Definitions

We restrict the definition of a ring to commutative rings with unity. Examples of such rings are $\mathbb{Z}$ and $\mathbb{Z}/n\mathbb{Z}$. A finite field further restricts this class by requiring the multiplication operator to be invertible. An example of a finite field is $\mathbb{F}_P \cong \mathbb{Z}/P\mathbb{Z}$ where P is prime. Within a finite field, an element which is a primitive Nth root of unity will be called ω_N. The modulus operator ($\%$) is used to map an element of an equivalence class in $\mathbb{Z}/n\mathbb{Z}$ to its canonical representative in $[0, n-1]$, e.g., $7\%5 = 2$.

Polynomial rings are formed by extending a ring R by a variable X to form the set $R[X]$. This set consists of elements of the form $a_0 + a_1X + a_2X^2 + ...$ with $a_i \in R$. A quotient ring can be constructed by partitioning the elements of the set $R[X]$ into equivalence classes based on their residues modulo a characteristic polynomial C, producing the set $R[X]/C$.

A ring isomorphism f, is a structure preserving (homomorphic) map between rings that is also a bijection. Together, these two properties allow ring isomorphisms to rewrite the multiplication operator. Specifically, $a * b$ is equivalent to $f^{-1}(f(a) * f(b))$.

2.1.1 Notation

A type symbol, such as $\mathbb{F}_P$, can be extended by a superscript to describe the structure of an array formed from elements of the type. Examples are $\mathbb{F}_P^n$ and $\mathbb{F}_P^{n\times n}$. When a superscript is attached to a scalar value, it should be interpreted as exponentiation using the relevant multiplication operator. An array structure is indexed to define a scalar value using a list of indices as a subscript, the length of which corresponds to the number of dimensions of the array. Examples are a_j, $\mathcal{F}_n(a)_i$, and $(\mathcal{T}_n)_{i,j}$. The use of superscripts and subscripts in other cases are implicit to a particular definition;

for example, they describe the structure of a transformation's domain, the size of a finite field, or the order of a root of unity.

2.2 Multiplication in RLWE

Many RLWE [22] cryptosystems use rings of the form $\mathbb{F}_P[X]/(X^n+1)$ where n is a power of two, P is prime, and $2n$ divides $P-1$. This ring structure is selected to be amenable to fast multiplication. Specifically, the characteristic polynomial X^n+1 has factors of the form $X-\omega_{2n}^{1+2i}$. Factors in the characteristic polynomial of a quotient ring support fast multiplication because these factors determine internal direct products which can be exploited to reduce the number of operations required to perform multiplication.

Our framework generalizes the structure of the characteristic polynomial by allowing polynomial rings of the form $\mathbb{F}_P[X]/(X^n-\omega_N^d)$. In the case of RLWE multiplication, there are many choices of d and N which could represent (X^n+1) e.g., $(X^n-\omega_{2z}^z)$. However, $2z$ must divide $P-1$ for ω_{2z} to exist in $\mathbb{F}_P$ [1]. To insure the existence of roots of unity, and avoid modification of the variable N, we require the following divisibility constraints on n, d, N and P.

$$n|d$$

$$d|N$$

$$N|P-1$$

The values of n and P as well as the ratio between d and N are determined by the relevant cryptosystem. The choice of d and N with the constrained set is inconsequential. This is because the value of z disappears when ω_{kz}^z is reduced to ω_k when evaluating.

[1] This is a consequence of the Primitive Element Theorem (PET), Fermat's Little Theorem (FLT), and Lagrange's Theorem. PET states that $\mathbb{F}_P^*$ must contain a multiplicative generator (a), and FLT states that a must have multiplicative order $P-1$. If $2z|P-1$, we may find an example of ω_{2z} in $\mathbb{F}_P^*$ as $a^{(P-1)/(2z)}$. If $2z \nmid P-1$, then by Lagrange's Theorem there is no example of ω_{2z} in $\mathbb{F}_P^*$, because ω_{2z} would generate a subgroup of $(\mathbb{Z}_P^*, *)$ of size $2z$.

2.3 Discrete Fourier Transformation

The Discrete Fourier Transformation (DFT) is a family of operations used in a variety of domains including digital signal processing [11], financial models [1], and quantum computing [26].

2.3.1 Number Theoretic Transform

One way to classify Discrete Fourier Transformations describes the algebraic structure of the inputs. The Number Theoretic Transform (NTT) is one such class, and refers to Fourier Transformations performed on vectors over finite fields. The standard definition of the Number Theoretic Transformation $\mathcal{F}_n$ is given below.

$$\mathcal{F}_n : \mathbb{F}_P^n \to \mathbb{F}_P^n \tag{2.1}$$

$$\mathcal{F}_n(a)_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \tag{2.2}$$

A direct evaluation of the above equation leads to a naive algorithm with computational complexity $\mathcal{O}(n^2)$.

2.3.2 Fast Fourier Transform

Another way to classify the Fourier Transformation describes the structure and complexity of the algorithm. The Fast Fourier Transformation (FFT) refers to algorithms with $\mathcal{O}(n \log(n))$ complexity [9]. One such algorithm is described by the equation below, which may be derived from the substitution $i \to i' + i''n/2$, with $i' = i\%(n/2)$ and $i'' = \lfloor i/(n/2) \rfloor$.

$$\mathcal{F}_n(a)_{i'+i''n/2} = \mathcal{F}_{n/2}(a_{2j})_{i'} + \omega_2^{i''} \omega_n^{i'} \mathcal{F}_{n/2}(a_{2j+1})_i \tag{2.3}$$

The above structure can be further described as the radix-2, decimation-in-frequency FFT. Equation (2.3) is radix-2 because it divides the domain into two subsets ($\{a_{2j} | 0 \leq j < n/2\}$ and $\{a_{2j+1} | 0 \leq j < n/2\}$). Equation (2.3) uses decimation-in-frequency because the subsets are

selected by striding along the domain using the radix as the step size (the alternative, decimation-in-time, uses contiguous subsets).

2.3.3 DFT as a Linear Operator

One can interpret the Discrete Fourier Transform as a matrix vector product. This interpretation leads to an alternative definition of equation (2.2) using a transformation matrix $\mathcal{T}$.

$$\mathcal{T}_n \in \mathbb{F}_P^{n \times n} \tag{2.4}$$

$$(\mathcal{T}_n)_{i,j} = \omega_n^{ij} \tag{2.5}$$

$$\mathcal{F}_n(a) = \mathcal{T}_n a \tag{2.6}$$

Using this interpretation, fast Fourier algorithms can be described as sparse decompositions of the transformation matrix. This is the view taken by Spiral's authors [11] and will be further explained in Section 2.5.2.

2.3.4 DFT as Polynomial Evaluation

Another interpretation of the Fourier transformation looks at evaluating a polynomial of degree (at most) $n-1$ over the set $\{\omega_n^i | 0 \leq i < n\}$. This interpretation arises from a notion of equivalence between a vector of length n and a polynomial of degree (at most) $n-1$ created by equating the ith vector element to the polynomial coefficient corresponding to the X^i term.

$$a(X) \in \mathbb{F}_P[X],\ \vec{a} \in \mathbb{F}_P^n \tag{2.7}$$

$$a(X) \equiv \vec{a} \tag{2.8}$$

$$\mathcal{F}_n(\vec{a})_i = a(\omega_n^i) \tag{2.9}$$

Using this interpretation, a fast Fourier algorithm can be described as a chain of ring isomorphisms starting from the input domain $\mathbb{F}_P[X]/(X^n-1)$ and ending with the codomain $\prod_{i=0}^{n-1} \mathbb{F}_P[X]/(X-\omega_n^i)$. This interpretation was proposed by Fiduccia [12] and was further explored in the context of fast multiplication by Bernstein [3].

2.4 Fast Multiplication

Fast multiplication algorithms rely on finding a function $f : \mathcal{R}_0 \to \mathcal{R}_1$ which defines an isomorphism between rings $(\mathcal{R}_0, *, +)$ and $(\mathcal{R}_1, \times, +)$. Using the isomorphic property of f, one may replace multiplication in one ring with multiplication in another, as follows.

$$a * b = f^{-1}(f(a) \times f(b)) \tag{2.10}$$

In equation (2.10) the multiplication operator $*$ (on $\mathcal{R}_0$) has been replaced by the multiplication operator $\times$ (on $\mathcal{R}_1$).

2.4.1 FFT Multiplication

The polynomial evaluation interpretation of the Discrete Fourier Transformation leads to a family of fast multiplication algorithms. Specifically, a fast Fourier algorithm can be used to map the inputs to the codomain $\prod_{i=0}^{n-1} \mathbb{F}_P[X]/(X-\omega_n^i)$, where multiplication can be performed pointwise. Then, the inverse Fourier transformation can be used to recover the product in the input domain as shown in equation (2.11).

$$a * b = \mathcal{F}_n^{-1}(\mathcal{F}_n(a) * \mathcal{F}_n(b)) \tag{2.11}$$

This method has complexity $\mathcal{O}(n \log(n))$ and is usually attributed to Schönhage and Strassen [24].

2.5 Related Work

2.5.1 State-of-the-art Polynomial Ring Multiplication

The utility of ring isomorphisms for developing fast multiplication algorithms in cryptography can be shown by a review of the state of the art. Seiler [25] developed a novel NTT implementation for ring multiplication using the ring isomorphism framework developed by Bernstein [3]. Coincidental to the inclusion of Seiler's NTT implementation in the Kyber cryptosystem [5] several other projects have studied Seiler's implementation in both software and hardware. Nguyen et al. [19], compare fast multiplication using the Toom-Cook algorithm to Seiler's NTT method. Seiler's implementation was also used by Huang et al. [13] to develop an FPGA implementation of the Kyber cryptosystem.

Seiler builds upon previous optimizations of NTT for use in RLWE. For example, when using the Fast Fourier Transformation for multiplication in the ring $\mathbb{F}_P[X]/(X^n + 1)$ special adjustment is necessary to ensure that a negative-wrapped convolution is performed. This is because the standard definition of the FFT performs multiplication in the ring $\mathbb{F}_P[X]/(X^n - 1)$. One strategy applies scaling factors of ω_{2n}^i to each a_i and b_i input, and then applies scaling factors ω_{2n}^{-i} to each c_i result. This can be made more efficient by including these factors in the precomputed constants of the algorithm to produce a negative-wrapped Fourier transformation. This strategy was developed by Roy et al. [23] and Pöppelmann et al. [20] using analysis of the equational view of the Fourier Transformation. Seiler instead derives this strategy by viewing the FFT as a series of ring isomorphisms. Our framework includes this optimization by taking the same view as Seiler.

Another optimization made to the FFT when used for fast multiplication removes needless permutations of the codomain. This is possible because the missing permutations will be accounted for in the inverse transformation. Seiler implements this optimization through a manual derivation which leads to a particular permutation of elements in the domain, specifically in bit reversal order. We extend this optimization by allowing a family of permutations in the codomain, derived automatically through our synthesis tool.

Finally, efficient modular reductions based on Montgomery's trick [18] are a common optimization made to NTT algorithms. In state-of-the-art designs of NTT, the implementation of Montgomery's trick, and the frequency of modular reduction, are highly dependent on the hardware architecture [25] [23]. We do not improve the state of the art with regards to this method because our goal is not to improve performance on a particular platform but to design optimizations which can be applied automatically on any platform.

2.5.2 FFT in Program Synthesis

Program synthesis has proved to be a feasible and effective method for deriving fast implementations of the Discrete Fourier Transformation. For example, Spiral [11] is a program synthesis engine that uses rewrite rules to describe a space of possible FFT implementations and a combination of experimentation and AI tools to find and analyze elements within this space. While program synthesis methods are not generally feasible when scaled to large problem sizes, the use of rewrite rules which eventually terminate in base cases makes program synthesis feasible within the FFT problem domain.

The efficacy of program synthesis for the FFT relies on a unique property of the Fourier Transformation. In particular, the Fourier Transformation allows two recursive structures to be exploited at the same time. These structures are demonstrated by a rewrite rule used by Spiral [11]. This rule can be represented as a decomposition of the Fourier transformation matrix (below), where $n = km$.

$$\mathcal{T}_n = (\mathcal{T}_k \otimes I_m) T_m^n (I_k \otimes \mathcal{T}_m) L_k^n \tag{2.12}$$

The notation for the matrices T_m^n and L_k^n is borrowed from Spiral. T_m^n represents a diagonal matrix containing twiddle factors of the form ω_n^{ij}. The matrix L_k^n represents a strided permutation. When using this decomposition, one may recurse on both $(\mathcal{T}_k \otimes I_m)$ and $(I_k \otimes \mathcal{T}_m)$. This decomposition may be derived using the following formula.

$$\mathcal{F}_n(a)_{i+mi'} = \sum_{j=0}^{k-1} \omega_k^{i'j} \omega_n^{ij} \sum_{j'=0}^{m-1} a_{j+kj'} \omega_m^{ij'} \tag{2.13}$$

To derive the decomposition in eqn. (2.12) one can replace the appropriate expressions in eqn. (2.13) with the definition of the DFT ($\mathcal{F}$) from eqn. (2.2) to produce the following doubly-recursive algorithm.

$$\mathcal{F}_n(a)_{i+mi'} = \mathcal{F}_k(\omega_n^{ij} \mathcal{F}_m(a_{j+kj'})_i)_{i'} \tag{2.14}$$

While many divide and conquer algorithms pay a constant factor penalty when the domain is divided into more than two pieces, this doubly-recursive structure allows the number of divisions to be increased without increasing the asymptotic computational complexity. This leads to many implementations of the FFT with different recursive structures but similar computational complexity. However, these implementations may differ on several important characteristics, such as memory access patterns or arithmetic intensity.

Chapter 3

Approach

3.1 Contributions

1. **Abstract Synthesis** We raise the level of abstraction from previous FFT synthesis methods by viewing each step of an FFT as a ring isomorphism rather than a linear operator. This produces a DSL which more closely represents the underlying polynomial ring structures.

2. **Generalize Ring Structure** Raising the level of abstraction facilitates synthesizing multiplication algorithms for a more general class of ring structures. This class includes the ring structures used in the cryptosystems Crystals Kyber [5] and NewHope [2].

3. **Remove Unnecessary Permutations** Our analysis identifies the ring isomorphism (*SwapJoinProd*) which produces unnecessary permutations in the codomain. Replacing this isomorphism avoids applying these unnecessary permutations (Section 5.3.3).

4. **Automated Search** Abstracting each ring isomorphism as a morphism over types allows automated search of the algorithm design space. Extracting performance features from a compiled algorithm allows AI algorithms to effectively search this space.

3.2 Framework

We first give a framework describing ring isomorphisms which can be used as fundamental operations to automatically synthesize fast multiplication algorithms. An imperative definition of each fundamental operation (and its inverse) is given, allowing programs expressed as ring isomorphisms to be compiled to imperative languages.

There are several goals motivating the design of this framework. First, the fundamental operations are sufficiently expressive to allow synthesis of programs exhibiting the doubly-recursive structure from Section 2.5.2. Second, these fundamental operations adhere to closure properties,

which ensure the application of any sequence of fundamental operations will eventually terminate. This property is formally proven in Appendix C.

3.3 Implementation

We use a simplified representation of the ring isomorphism framework to build a program synthesis engine for deriving fast multiplication algorithms. This is achieved by translating the algebraic structures described by our framework to a type system in Haskell [17]. In this representation the fundamental operations of the framework correspond to morphisms over these types. We borrow the idea of a functor from category theory to allow manipulation of this type system through a set of morphisms and lifting functions (functions over morphisms).

In the end, this produces a method for describing fast multiplication algorithms as an initial ring structure and series of morphisms. Each of these algorithms can be compiled to an imperative language (C) to produce an implementation of polynomial ring multiplication.

3.4 Design Space Exploration

A simulated annealing algorithm is used to explore a set of multiplication algorithms and find elements which correspond to fast implementations. Algorithms in this space are expanded into a set of neighbors which share similar characteristics using a novel suggestion search algorithm. This method guides brute force search using a set of precomputed, symbolic, rewrite suggestions.

Chapter 4

Framework

The framework presented by Bernstein [3] compares and contrasts the algebraic structure of various fast multiplication algorithms. To use this framework in program synthesis it is necessary to make some refinements. To this end, we require that the fundamental operations must be computationally precise to allow translation to imperative languages. We define a series of Fundamental Ring Isomorphisms (FRIs), provide imperative implementations for each isomorphism (and its inverse), and finally describe how FRIs can be used to synthesize a variety of imperative programs. We follow the convention that a capital letter represents the imperative implementation of the ring isomorphism represented by the corresponding lowercase letter.

4.1 Factor

$$\phi_k : \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} \to \prod_{z=0}^{k-1} \frac{\mathbb{F}_P[X]}{X^{n/k} - \omega_N^{(d+zN)/k}} \tag{4.1}$$
$$\phi_k(X^{n/k})_z = \omega_k^{(d+zN)/k}$$

The *Factor* ring isomorphism (ϕ_k) exposes internal direct products in the underlying ring structure. Viewed another way, the *Factor* isomorphism is a generalization of the Discrete Fourier Transformation. Specifically, the DFT decomposes the ideal $(X^n - 1)$ into a set of substitution maps which send $X \rightsquigarrow \omega_N^z$ for $z \in [0, n-1]$. This precisely matches the *Factor* isomorphism when $k = n$ and $d = 0$. The *Factor* isomorphism generalizes the DFT by parameterizing the domain through the variable d, and parameterizing the codomain through the variable k.

An imperative definition of the *Factor* ring isomorphism is given below, where $n = mk$.

$$\Phi_{n,k,d} : \mathbb{F}_P^n \to \mathbb{F}_P^{k \times m} \tag{4.2}$$
$$\Phi_{n,k,d}(a)_{z,j} = \sum_{i=0}^{k-1} a_{im+j} \omega_N^{(d+zN)i/k}$$

This algorithm may be derived by inspecting the operations performed by applying the *Factor* isomorphism to a polynomial in the canonical representation of $R[X]/(X^n - \omega_N^d)$. According to the definition of ϕ_k, each X^m term is replaced by the root of unity: $\omega_k^{(d+zN)/k}$. To expose X^m terms, one rewrites each X^l power as $(X^m)^i X^j$ using the variable substitution: $l \to im + j$ for $i \in [0, k-1]$ and $j \in [0, m-1]$. This substitution is also applied to the coefficients, producing the term a_{im+j}. In the canonical representation of $\mathbb{F}_P/(X^{n/k} - \omega_N^{(d+zN)/k})$, each X^j term appears exactly once. To adhere to this form, coefficients are accumulated along i, producing the summation in equation (4.2). The scaling factor attached to each coefficient, $\omega_N^{(d+zN)i/k}$, represents the replaced $(X^m)^i$ term.

The inverse transformation is more difficult to derive. A proof that equation (4.3) correctly inverts the function defined in equation (4.2) is given in Appendix A.

$$\Phi_{n,k,d}^{-1} : \mathbb{F}_P^{k \times m} \to \mathbb{F}_P^n \tag{4.3}$$
$$\Phi_{n,k,d}^{-1}(A)_{im+j} = k^{-1} \sum_{z=0}^{k-1} A_{z,j} \omega_N^{-(d+zN)i/k}$$

4.2 Label

$$\xi_k : \frac{\mathbb{F}_P[X]}{X^{km} - \omega_N^d} \to \left(\frac{\mathbb{F}_P[Y]}{Y^m - \omega_N^d} \right) \frac{[X]}{X^k - Y} \tag{4.4}$$
$$\xi_k(X^k) = Y$$

The isomorphism *Label* (ξ_k) replaces the expression X^k with the symbol Y. On its own, this transformation only allows reordering of the coefficients, shown below.

$$\begin{aligned} a =& a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 \\ & \in \mathbb{F}_P[X]/(X^8 - 1) \\ \xi_2(a) =& (a_0y^0 + a_4y^1)x^0 + (a_1y^0 + a_5y^1)x + (a_2y^0 + a_6y^1)x^2 + (a_3y^0 + a_7y^1)x^3 \\ & \in (\mathbb{F}_P[Y]/(Y^2 - \omega_N^d))[X]/(X^4 - Y) \end{aligned}$$

The *Label* isomorphism is necessary to describe decimation-in-time FFT algorithms, as discussed by Bernstein [3], who calls this operation "Striding". An imperative definition of ξ_k is given below. To ensure termination and to avoid producing identity operations, we require that $m, k > 1$.

$$\begin{aligned} \Xi_{k,m} : \mathbb{F}_P^{m\times k} &\to \mathbb{F}_P^{k\times m} \\ \Xi_{k,m}(a)_{i,j} &= a_{j,i} \\ \Xi_{k,m}^{-1}(A)_{j,i} &= A_{i,j} \end{aligned} \tag{4.5}$$

While this transformation does not involve computation, it represents a derangement of the data in memory. Specifically, the arrays in (4.5) are stored in row major order. This corresponds to the L_m^n permutation in Spiral [11].

4.3 Normalize

$$\begin{aligned} \psi : \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} &\to \left(\frac{\mathbb{F}_P[Y]}{Y^n - 1}\right) \frac{[X]}{X - \omega_N^{d/n}Y} \\ \psi(X) &= \omega_N^{d/n}Y \end{aligned} \tag{4.6}$$

The *Normalize* (ψ) isomorphism returns a characteristic polynomial to the canonical form $Y^n - 1$. To ensure termination and to avoid producing identity operations, we require $d \neq 0$ and $n \neq 1$. *Normalize* is a linear time operation which applies scaling factors to each element of the input, as shown below.

$$\Psi_{n,d} : \mathbb{F}_P^n \to \mathbb{F}_P^n \tag{4.7}$$
$$\Psi_{n,d}(a)_i = \omega_N^{id/n} a_i$$
$$\Psi_{n,d}^{-1}(a)_i = \omega_N^{-id/n} a_i$$

This transformation appears in several algorithms. The most well known is the use of twiddle factors in the radix-2 algorithm (ω_N^i in eqn. 2.3). Another example is the scaling factors used to allow a standard FFT algorithm to compute a negative-wrapped convolution, as discussed in Section 2.5.1. In addition, the twiddle matrix T_m^n used by Spiral [11] (discussed in Section 2.5.2) is another example of *Normalize*. Finally, this transformation is a generalization of the "Twisting" map from Bernstein [3].

4.4 Define

The two versions of *Define* given below represent two cases for replacing a variable with its definition.

$$\delta^E : \left(\frac{\mathbb{F}_P[Y]}{Y - \omega_N^d}\right) \frac{[X]}{X^m - Y} \to \frac{\mathbb{F}_P[X]}{X^m - \omega_N^d} \tag{4.8}$$

The *DefineExp* (δ^E) isomorphism provides an indirect inverse to the *Label* isomorphism by defining the innermost variable, in this case Y, with an identity produced by *Label*.

$$\delta^R : \left(\frac{\mathbb{F}_P[Y]}{Y - \omega_N^d}\right) \frac{[X]}{X - \omega_N^z Y} \rightarrow \frac{\mathbb{F}_P[X]}{X - \omega_N^{d+z}} \tag{4.9}$$

The *DefineRot* (δ^R) isomorphism provides an indirect inverse to the *Normalize* isomorphism by defining the innermost variable, in this case Y, with an identity produced by *Normalize*.

When viewed imperatively, the $Define$ isomorphisms are identity operations. This is because the variable Y does not appear in the canonical representation of $\mathbb{F}_P[Y]/(Y - \omega_N^d)$.

4.5 Swap

The four versions of *Swap* given below represent three cases for reversing the order of two surrounding ring structures. Refer to eqns. (5.3) through (5.5) for definitions of surrounding ring structures.

$$\zeta^{EP} : \left(\prod_{i=0}^{n_2-1} S[X](i)\right) \frac{[Y]}{Y^{n_1} - X} \rightarrow \prod_{i=0}^{n_2-1} \left(S[X](i) \frac{[Y]}{Y^{n_1} - X}\right) \tag{4.10}$$

SwapEP (ζ^{EP}) reverses the order of an *Expansion* surrounding a *Product*. The innermost term, $S[X](i)$, is a polynomial ring structure whose outermost variable extension is X and is parameterized by the variable i.

SwapEP can be implemented as a permutation (Z); shown below. This permutation is similar to the implementation of *Label* (eqn. (4.5)), except it moves chunks of memory at a time. The length of each chunk, m, corresponds to the length of the data used to represent the innermost term in eqn. (4.10).

$$Z_{n_1,n_2,m} : \mathbb{F}_P^{n_1 \times n_2 \times m} \to \mathbb{F}_P^{n_2 \times n_1 \times m} \tag{4.11}$$
$$Z_{n_1,n_2,m}(a)_{i,j,k} = a_{j,i,k}$$
$$Z^{-1}_{n_1,n_2,m}(a)_{j,i,k} = A_{i,j,k}$$

SwapRP (η^{RP}) reverses the order of a *Rotation* surrounding a *Product*. When viewed imperatively, *SwapRP* is an identity operation.

$$\eta^{RP} : \left(\prod_{i=0}^{n-1} S[X](i)\right) \frac{[Y]}{Y - \omega_N^d X} \to \prod_{i=0}^{n-1} \left(S[X](i) \frac{[Y]}{Y - \omega_N^d X}\right) \tag{4.12}$$

SwapER (θ^{ER}) reverses the order of an *Expansion* surrounding a *Rotation*. The innermost term, $S[W]$ represents a polynomial ring structure whose outermost variable extension is W.

$$\theta^{ER} : \left(S[W] \frac{[Y]}{Y - \omega_N^d W}\right) \frac{[X]}{X^n - Y} \to \left(S[W] \frac{[V]}{V^n - W}\right) \frac{[X]}{X - \omega_N^{d/n} V} \tag{4.13}$$
$$\theta^{ER}(X) = \omega_N^{d/n} V$$

The derivation of the *SwapER* isomorphism is given in Appendix B. Similar to *Normalize*, the implementation of *SwapER* only applies scaling factors to elements of the input. The precise method is described by the function Θ.

$$\Theta_{n,m,d} : \mathbb{F}_P^{n \times m} \to \mathbb{F}_P^{n \times m} \tag{4.14}$$
$$\Theta_{n,m,d}(a)_{i,j} = \omega_N^{id/n} a_{i,j}$$
$$\Theta^{-1}_{n,m,d}(a)_{i,j} = \omega_N^{-id/n} a_{i,j}$$

As before, m corresponds to the length of the data used to represent the innermost term. Notice Θ extends Ψ in the same way Z extends Ξ.

4.6 Join

The three versions of $Join$ given below represent three cases for joining identical surrounding ring structures.

$$\rho :: \prod_{i=0}^{n_1-1} \left(\prod_{j=0}^{n_2-1} S(i,j) \right) \to \prod_{i=0}^{n_1 n_2-1} S(\lfloor i/n_2 \rfloor, i\%n_2) \tag{4.15}$$

$$\mu :: \left(S[Z] \frac{[X]}{X^n - Z} \right) \frac{[Y]}{Y^{n_2} - X} \to S[Z] \frac{[Y]}{Y^{n_1 n_2} - Z} \tag{4.16}$$

$$\sigma :: \left(S[Z] \frac{[X]}{X - \omega_N^{d_1} Z} \right) \frac{[Y]}{Y - \omega_N^{d_2} X} \to S[Z] \frac{[Y]}{Y - \omega_N^{d_1+d_2} Z} \tag{4.17}$$

JoinProd (ρ), *JoinExp* (μ), and *JoinRot* (σ) can all be used to reduce the number of surrounding expressions in a ring structure. When viewed imperatively each of these functions is an identity operation.

4.7 SwapJoin

$$\tau :: \prod_{i=0}^{n_1-1} \left(\prod_{j=0}^{n_2-1} S(i,j) \right) \to \prod_{i=0}^{n_1 n_2-1} S(i\%n_1, \lfloor i/n_1 \rfloor) \tag{4.18}$$

SwapJoinProd (τ) reverses the order of two $Products$ and then combines the two. *SwapJoinProd* is necessary to describe many fast Fourier algorithms (see), where the order of elements in

the codomain ($\prod_{i=0}^{n-1} \mathbb{F}_P[X]/(X - \omega_N^i)$) is important. However, this isomorphism tends to obstruct fast polynomial multiplication, where the order of elements within a *Product* does not matter, and the permutation will have to be undone by the inverse operation. For this reason, *SwapJoinProd* is replaced with *SwapProd* before compilation (Section 5.3.3).

The imperative implementation of *SwapJoinProd* matches the implementation Z, the imperative definition of *SwapEP*, presented by eqn. (4.11).

Chapter 5

Implementation

5.1 Simplified Representation

5.1.1 Type System

The polynomial ring structures described in the previous section can be simplified by recognizing common structures in surrounding expressions. This simplified representation produces a type system which can be used to describe the domain and codomain of each fundamental ring isomorphism. The structure of this system is given below.

$$Ring := Base\ \mathbb{N}\ \mathbb{N} \mid Product\ \mathbb{N}\ (\mathbb{N} \to Ring) \mid Expansion\ \mathbb{N}\ Ring \mid Rotation\ \mathbb{N}\ Ring \quad (5.1)$$

The correspondence of this system with the surrounding ring structures manipulated by the previous section is given by eqns. (5.2) through (5.5). The variables P and N are omitted from the simplified representation because they are not modified by any fundamental ring isomorphism.

$$Base\ n\ d := \frac{\mathbb{F}_P[X]}{X^n - \omega_N^d} \quad (5.2)$$

$$Product\ n\ f := \prod_{i=0}^{n-1} f(i) \quad (5.3)$$

$$Expansion\ n\ (S[Y]) := (S[Y])\frac{[X]}{X^n - Y} \quad (5.4)$$

$$Rotation\ d\ (S[Y]) := (S[Y])\frac{[X]}{X - \omega_N^d Y} \quad (5.5)$$

For brevity we will use the abbreviations *Prod*, *Exp*, and *Rot* for *Product*, *Expansion* and *Rotation* respectively.

5.1.2 Morphisms

We can now translate the type signature of each fundamental ring isomorphism into the simplified representation as a morphism. In doing so, we change the level of abstraction so what was once a type signature is now a definition. At this abstraction level, eqns. (5.6) through (5.17) represent functions with signature $(Ring \rightarrow Ring)$ [2]. In these equations, Church's lambda notation [7] is used to describe anonymous functions.

$$Factor\ k : \phi_k(Base\ n\ d) = Prod\ k\ (\lambda z.\ Base\ (n/k)\ ((d + zN)/k)) \tag{5.6}$$

$$Label\ k : \xi_k(Base\ n\ d) = Exp\ k\ (Base\ (n/k)\ d) \tag{5.7}$$

$$Normalize : \psi(Base\ n\ d) = Rot\ (d/n)\ (Base\ n\ 0) \tag{5.8}$$

$$DefineExp : \delta^E(Exp\ n\ (Base\ 1\ d)) = Base\ n\ d \tag{5.9}$$

$$DefineRot : \delta^R(Rot\ d_1\ (Base\ 1\ d_2)) = Base\ n\ (d_1 + d_2) \tag{5.10}$$

$$SwapEP : \zeta^{EP}(Exp\ n_1\ (Prod\ n_2\ f) = Prod\ n_2\ (\lambda i.\ Exp\ n_1\ f(i))) \tag{5.11}$$

$$SwapRP : \eta^{RP}(Rot\ d\ (Prod\ n\ f)) = Prod\ n\ (\lambda i.\ Rot\ d\ f(i)) \tag{5.12}$$

$$SwapER : \theta^{ER}(Exp\ n\ (Rot\ d\ S)) = Rot\ (d/n)\ (Exp\ n\ S) \tag{5.13}$$

$$JoinExp : \mu(Exp\ n_1\ (Exp\ n_2\ S)) = Exp\ (n_1 n_2)\ S \tag{5.14}$$

$$JoinRot : \sigma(Rot\ d_1\ (Rot\ d_2\ S)) = Rot\ (d_1 + d_2)\ S \tag{5.15}$$

$$\begin{aligned} JoinProd : \rho(&Prod\ n_1\ (\lambda i.\ Prod\ n_2\ (\lambda j.\ f(i,j)))) \\ &= Prod\ (n_1 n_2)\ (\lambda z.\ f(\lfloor z/n_2 \rfloor, z\%n_2)) \end{aligned} \tag{5.16}$$

$$\begin{aligned} SwapJoinProd : \tau(&Prod\ n_1\ (\lambda i.\ Prod\ n_2\ (\lambda j.\ f(i,j)))) \\ &= Prod\ (n_1 n_2)\ (\lambda z.\ f(z\%n_1, \lfloor z/n_1 \rfloor)) \end{aligned} \tag{5.17}$$

[2] The careful reader will notice the definitions of these functions are not legal within Haskell's type system. Specifically, each of these *Morphisms* defines a partial function on the type *Ring*. Here, and in later definitions, a *Maybe* monad is used to represent partial functions (producing the type $Ring \rightarrow Maybe\ Ring$), but is omitted from our discussion to avoid clutter.

It is advantageous to represent morphisms symbolically as a type *Morphism* defined as follows.

$$\begin{aligned} Morphism \ := & \ Factor \ Int \mid Label \ Int \mid Normalize \mid \\ & DefineExp \mid DefineRot \mid \\ & SwapEP \mid SwapRP \mid SwapER \mid \\ & JoinExp \mid JoinRot \mid JoinProd \mid SwapJoinProd \end{aligned} \tag{5.18}$$

To recover the *Ring* map from the symbolic *Morphism* we implement an *applyMorphism* function which unsymbolizes a *Morphism* by returning the corresponding *Ring* map.

$$applyMorphism :: Morphism \rightarrow Ring \rightarrow Ring \tag{5.19}$$

5.1.3 Algorithms in the Simplified Representation

A fast multiplication algorithm can be described as an initial *Ring* structure, a series of *Morphisms*, and a final *Ring* structure. This is summarized by the type *Path*.

$$Path := Path \ \{start :: Ring, \ series :: [Morphism], \ end :: Ring\} \tag{5.20}$$

A *Path* is legal when when the composition of functions formed by the unsymbolized *series* maps *start* to *end*.

5.1.4 Compiling the Simplified Representation

The *Path* representation of a multiplication algorithm can be translated to an imperative equation by first applying a *compile* function to each *Morphism* in *series*. This step simply substitutes the current *Morphism* with the corresponding forward imperative definition found in Chapter 4. However, in the simplified representation, each *Morphism* on its own does not contain enough

information to define the imperative definition, so one must also know the *Ring* structure the *Morphism* is applied to. This results in a compile function with the following signature.

$$compile :: Morphism \to Ring \to LinearOperator \tag{5.21}$$

To simplify notation, many of the imperative definitions in Section 4 use multidimensional vectors (e.g. $\mathbb{F}_P^{k\times m}$). For compilation, we implicitly convert to one dimensional vectors (e.g. $\mathbb{F}_P^{km}$). The type *LinearOperator* represents symbolized functions of type ($\mathbb{F}_P^n \to \mathbb{F}_P^n$).

Second, one must also define the inverse operation of a *Path*. This is achieved by applying a corresponding *compile_inverse* function to each *Morphism* in *series*. The *compile_inverse* function recovers the inverse, imperative definition found in Chapter 4. The *compile_inverse* function has the same signature as *compile*. Notice that in Chapter 4 the inverse, imperative definitions are described using variables defined in the type of the domain of the forward function. As a result, and somewhat counter intuitively, the inverse of a *Morphism* is defined using the *Ring* of its codomain rather than its domain.

Lastly, one must determine how to perform multiplication at the *end* structure of the *Path*. We simplify this step by requiring that all *end Ring* structures may only contain the *Product* surrounding structure. This ensures that multiplication may always be performed point-wise using the same implementation.

These three steps determine the functions f and f^{-1} as well as the operator $\times$ from equation (2.10).

5.2 Functors

Fundamental ring isomorphisms alone are not enough to synthesize fast Fourier algorithms. This is because the domain of many isomorphisms cannot be matched against their codomain. For example, a single application of the *Factor* isomorphism (ϕ_k eqn. (4.1)) produces a polynomial ring structure of the form $Prod\ n\ (\lambda z.\ Base\ m\ f)$ which cannot be matched against the domain

of *Factor*; simply $(Base\ n\ d)$. Therefore, using only the ring isomorphisms presented, even the most simple algorithms remain out of reach.

To solve this problem, we borrow and abuse the idea of a functor from category theory [16]. This strategy defines functors for each of *Rotation*, *Expansion*, and *Product*. These functors extend the signature of a function defined over *Ring* by wrapping additional structure around the domain and codomain.

5.2.1 Rotation

The functor for *Rotation* is named $\mathcal{R}$. It's signature and definition are given below.

$$\mathcal{R} : (Ring \to Ring) \to Ring \to Ring \tag{5.22}$$

$$\mathcal{R}(f_L,\ Rotation\ d\ S) = Rotation\ d\ f_L(S) \tag{5.23}$$

Let F_L be the imperative version of the lifted function f_L acquired through the expression $compile(f_L, S)$. The imperative definition of $\mathcal{R}$, which we will call $\hat{\mathcal{R}}$, simply applies the function F_L to the domain.

$$\hat{\mathcal{R}}(F_L) : \mathbb{F}_P^n \to \mathbb{F}_P^n \tag{5.24}$$

$$\hat{\mathcal{R}}(F_L, a)_i = F_L(a)_i \tag{5.25}$$

5.2.2 Expansion

The functor for *Expansion* is named $\mathcal{E}$. It's signature and definition are given below.

$$\mathcal{E} : (Ring \to Ring) \to Ring \to Ring \tag{5.26}$$

$$\mathcal{E}(f_L,\ Expansion\ n\ S) = Expansion\ n\ f_L(S) \tag{5.27}$$

Let F_L be the imperative version of the lifted function f_L. The imperative definition of $\mathcal{E}$, which we will call $\hat{\mathcal{E}}$, repeats the function F_L over n subsets of the domain. This corresponds to the construction $I_n \otimes F_L$, when F_L is viewed as a linear operator. The signature of the function F_L is $\mathbb{F}_P^{n'} \to \mathbb{F}_P^{n'}$ in the definition of $\hat{\mathcal{E}}_n$ below.

$$\hat{\mathcal{E}}_n(F_L) : \mathbb{F}_P^{n \times n'} \to \mathbb{F}_P^{n \times n'} \tag{5.28}$$

$$\hat{\mathcal{E}}_n(F_L, a)_{i,j} = F_L(a_i)_j \tag{5.29}$$

5.2.3 Product

The functor for *Product* is named $\mathcal{P}$, and is described below.

$$\mathcal{P} : (Ring \to Ring) \to Ring \to Ring \tag{5.30}$$

$$\mathcal{P}(f_L,\ Product\ n\ f_P) = Product\ n\ (f_L \circ f_P) \tag{5.31}$$

The imperative definition of $\mathcal{P}$ is more subtle. In the definition above the function f_P is composed with the function f_L to form the function in the result ($f_C : \mathbb{N} \to Ring$). Let F_C be the imperative version of this function with with signature $\mathbb{N} \to \mathbb{F}_P^{n'} \to \mathbb{F}_P^{n'}$. F_C is obtained by composing f_P with the imperative version of f_L through the expression $(compile\ f_L) \circ f_P$. The imperative definition of the *Product* functor, $\hat{\mathcal{P}}_n$, is given below.

$$\hat{\mathcal{P}}_n(F_C) : \mathbb{F}_P^{n \times n'} \to \mathbb{F}_P^{n \times n'} \tag{5.32}$$

$$\hat{\mathcal{P}}_n(F_C, a)_{i,j} = F_C(i, a_i)_j \tag{5.33}$$

5.3 Algorithm Synthesis

The framework developed in the previous sections leads to a novel domain of fast multiplication algorithms. This domain is represented by the type *Path* from equation (5.20). We will first show how to construct a legal *Path* and then how to find similar legal *Path*s.

To find a legal *Path* we use a *findMorphisms* function to identify a set of *Morphism*s which can be applied to a given *Ring* structure. The signature of the *findMorphisms* function is given below.

$$findMorphisms :: Ring \to [Morphism] \tag{5.34}$$

A function *findLegalPath* may be constructed from the functions previously described (*apply-Morphism* and *findMorphisms*) as well as some selection function of type (*[Morphism] -> Morphism*). An implementation of the function *findLegalPath* is given in Listing 5.1. The variables *current_ring* and *series* are initialized to *start* and an empty list respectively.

Listing 5.1: `findLegalPath` function

```
findLegalPath ::
    ([Morphism] -> Morphism) ->
    Ring ->
    Ring ->
    [Morphism] ->
    Path
findLegalPath selection_func start current_ring series =
    let
        possible_morphs = findMorphisms current_ring
        new_morph = selection_func possible_morphs
        new_ring = applyMorphism new_morph current_ring
    in
        if null possible_morphs then
            Path start series current_ring
            else
            findLegalPath
                selection_func
                start
                new_ring
                (new_morph : series)
```

Before applying an optimization algorithm we need a method of expanding one *Path* into a neighborhood of similar *Paths*. In the following Sections (5.3.1, 5.3.2) we present two possible expansion algorithms.

5.3.1 Bounded Search Algorithm

The Bounded Search Algorithm (BSA) expands a single *Path* into a neighborhood of similar *Paths*. Given an initial *Path*, BSA first identifies the sequence of *Ring* structures traversed by this *Path*. After selecting two *Ring* structures from this sequence (*sub_start* and *sub_end*), BSA attempts to find a series of *Morphism* which will lead from *sub_start* to *sub_end*. If such a series exists, it can be spliced into the initial *Path* to form a distinct *Path* with similar characteristics while still traversing from the same *start* to the same *end*. Fig. 5.1 depicts an example.

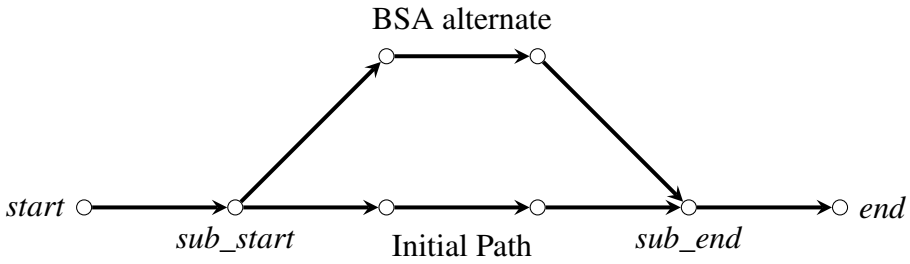

Figure 5.1: A new *Path* found by BSA

BSA is bounded in two ways. First, the distance between *sub_start* and *sub_end* in the initial *Morphism* sequence is bounded. This ensures the new *Path* will share characteristics with the initial *Path*. Second, the length of possible new subseries between *sub_start* and *sub_end* is bounded. This limits the depth of the search and the time it will take BSA to expand a *Path*.

The neighborhood of expanded *Paths* found by BSA is a result of the selection of *sub_start* and *sub_end* pairs as well as the possible replacement subseries between any selected pair.

BSA works well on small problem sizes but does not scale well to larger values of n. This is because, as the value of n increases the number of possible *Morphisms* produced by *findMorphisms* increases, which in turn increases the branching factor of the search algorithm.

5.3.2 Suggestion Search Algorithm

The Suggestion Search Algorithm (SSA) is inspired by the rewrite rules used in Spiral. In Spiral, rewrite rules are used to recursively redefine the decomposition of a linear operator and are

derived by hand and then supplied to the synthesis engine. In our system, these rewrite rules take the form of suggested replacements of *Morphism* series and are derived automatically.

Suggestions are derived from equivalence classes over *Morphism* series. These equivalence classes are discovered by examining a *Ring* with structure (*Base 6 0*). We start by enumerating all reachable *Ring* structures from (*Base 6 0*), and all possible *Morphism* series (of bounded length) between these *Ring* structures. Then, each equivalence class is constructed as a set of *Morphism* series which all lead from the same start *Ring* to the same end *Ring*.

Notice this is not an exact notion of equivalence. We would like to use equivalence to replace one *Morphism* series with another from the same equivalence class. To do this naively, we would need know that for any two *Morphism* series from the same equivalence class, given any starting *Ring* structure, both *Morphism* series would lead to the same end *Ring* structure. This is not the notion of equivalence produced by the above algorithm, which only assures that two "equivalent" *Morphism* series will lead to the same end *Ring* when given a particular start *Ring* structure. Perhaps one could formulate conditions on the initial *Ring* which ensure strict equivalence, but we take a different approach. Rather than producing identities, which are strictly equivalent, we use our relaxed notion of equivalence to produce suggestions, which must be verified. Eqn. (5.35) gives an example of two *Morphism* series which are equivalent for some starting *Rings* but not equivalent for others. In this example, the *buildPath* function takes an *start Ring* and a *Morphism series* and produces a *Path* when one can be legally constructed, and returns *Nothing* otherwise.

$$
\begin{aligned}
&series\_a := [Factor\ 3] \\
&series\_b := [Label\ 2,\ \mathcal{E}\ (Factor\ 3),\ SwapEP,\ \mathcal{P}\ Define] \\
&path\_6a := buildPath\ (Base\ 6\ 0)\ series\_a \\
&path\_6b := buildPath\ (Base\ 6\ 0)\ series\_b \\
&path\_3b := buildPath\ (Base\ 3\ 0)\ series\_b \\
&assert\ \ (end\ path\_6a\ ==\ end\ path\_6b) \\
&assert\ \ (path\_3b\ == Nothing)
\end{aligned}
\tag{5.35}
$$

Suggestions are discovered from the $(Base\ 6\ 0)$ structure. As a result, they are initially limited to trivial sizes. We can extend the equivalence classes discovered from $(Base\ 6\ 0)$ to larger sizes by representing *Morphisms* in these equivalence classes symbolically. We do so by replacing integer factors with symbolic variables. The size 6 is chosen to differentiate between the two factors: 3 and 2. This method produces 27 equivalence classes, some of which correspond to the rewrite rules used by Spiral. For example, the *Morphism* series used in (6.5) and its equivalence with (*Factor mk*), can be discovered by this method.

SSA follows the same design as BSA but uses suggestions to limit the branching factor of the search step. First, SSA considers a slice of *Morphisms* taken from a *Path*. SSA then matches the slice against symbolic representations of *Morphism* series in the library of equivalence classes. This match is defined by a map between symbolic variables and integers. The other symbolic *Morphism* series in a matched equivalence class represent suggested replacements of the initial slice. Importantly, some of the suggested replacements may contain variables which have not been mapped because they do not appear in the original matched symbolic representation. SSA uses the search step of BSA to discover values for these missing variables by representing each match as a *SymbolicMatchContext* (Listing 5.2). Viewed another way, the symbolic match contexts defined by the matched equivalence class are used to filter candidates from the result of the *findMorphisms*

function which we have reason be believe lead to the correct *sub_end Ring* structure. SSA terminates a search sequence when the correct *sub_end* structure is reached or there are no *Morphisms* discovered which correspond to a *SymbolicMatchContext*.

Listing 5.2: A (simplified) type label for *SymbolicMatchContext*. The *Map* records values for each symbolic variable. The *List* of *SymbolicMorphisms* represents the remaining *Morphisms* which may be matched by BSA.

```
type SymbolicMatchContext = (Map String Int, [SymbolicMorphism])
```

5.3.3 *Path* Optimization

To enhance performance before compiling to linear operators, a Path undergoes modification. Specifically, all instances of *SwapJoinProd* are substituted with *JoinProd*. This substitution is legal because the permutation caused by *SwapJoinProd* affects the sequence of elements in subsequent *Product* structures in the *Path*. Hence, replacing *SwapJoinProd* with *SwapJoin* only alters the order of elements in these *Products*. The modified order will be taken into account when the inverse transformation is applied. Seiler [25] implicitly applies this optimization, leading to the bit-order reversal in the codomain. The expansion algorithms SSA and BSA can't detect this optimization since they strictly compare the equality of two *Ring* structures and overlook equivalence modulo *Product* order.

Chapter 6

Familiar Algorithms

Standard

The standard Fourier transformation algorithm may be expressed as a *Path* containing a single *Morphism*. This *Path* is shown below.

$$\begin{array}{lrrr} (N = n) & Ring & Morphism & Linear\ Operator \\ \hline & Base\ n\ 0 & Factor\ n & \mathcal{T}_n \\ Prod\ n\ (\lambda i.Base\ 1\ i) & & - & - \end{array} \tag{6.1}$$

Negative Wrapped

A negative-wrapped Fourier transformation contains the same *Morphism*. However, it starts from a different *Ring* and uses a different value of N. The linear operator $\mathcal{T}_n^{NW}$ represents the modified transformation matrix (refer to eqn. (4.2) for a definition).

$$\begin{array}{lrrr} (N = 2n) & Ring & Morphism & Linear\ Operator \\ \hline & Base\ n\ n & Factor\ n & \mathcal{T}_n^{NW} \\ Prod\ n\ (\lambda i.Base\ 1\ (1 + 2i)) & & - & - \end{array} \tag{6.2}$$

Negative-Wrapped Reduction

This algorithm describes how to use the *Normalize* isomorphism to reduce a negative-wrapped transformation ($\mathcal{T}_n^{NW}$) to the standard transformation ($\mathcal{T}_n$). The first step of this algorithm applies the operator T_n^n, which represents a diagonal twiddle matrix. The T_n^n operator corresponds to the

ω_{2n}^{i} powers applied in Section 2.5.1.

$$\begin{array}{lrll} (N = 2n) & Ring & Morphism & Linear\ Operator \\ \hline & Base\ n\ n & Norm & T_n^n \\ & Rot\ 1\ (Base\ n\ 0) & \mathcal{R}\ (Factor\ n) & \mathcal{T}_n \\ Rot\ 1\ (Prod\ n\ (\lambda i.Base\ 1\ (2i))) & & SwapRP & (I_n) \\ Prod\ n\ (\lambda i.Rot\ 1\ (Base\ 1\ (2i))) & & DefineRot & (I_n) \\ Prod\ n\ (\lambda i.(Base\ 1\ (1+2i))) & & - & - \end{array} \tag{6.3}$$

Decimation in Frequency

The radix-k, decimation-in-frequency algorithm first applies a *Factor k* isomorphism and recurses on the remaining *Factor m* isomorphism. This is a generalized notion of recursion, because the initial *Ring* type of the *Factor m* isomorphism changes at each step. As a result, this algorithm is difficult to represent using identities of linear operators, as it requires defining a large family of Fourier transformations and their decompositions.

$$\begin{array}{lrl} (N = n) & Ring & Morphism \\ \hline & Base\ n\ 0 & Factor\ k \\ & Prod\ k\ (\lambda i.Base\ m\ (mi)) & \mathcal{P}\ (Factor\ m) \\ Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Base\ 1\ (i+kj))) & & SwapJoinProd \\ Prod\ n\ (\lambda z.Base\ 1\ z) & & - \end{array} \tag{6.4}$$

Decimation in Frequency, Recursive

A perfectly recursive implementation of the decimation-in-frequency algorithm is given below. This algorithm uses the *Normalize* isomorphism in the same way as the negative-wrapped reduction to return a Fourier transformation to standard form. Notice eqn. (6.5) is one of the decompositions used by Spiral [11], and the reverse of eqn. (2.12).

$(N = n)$	$Ring$	$Morphism$	$Linear\ Operator$
	$Base\ n\ 0$	$Factor\ k$	$\mathcal{T}_k \otimes I_m$
	$Prod\ k\ (\lambda i.Base\ m\ (mi))$	$\mathcal{P}\ Norm$	T^n_m
	$Prod\ k\ (\lambda i.Rot\ i\ (Base\ m\ 0))$	$\mathcal{P}\ (\mathcal{R}\ (Factor\ m))$	$I_k \otimes \mathcal{T}_m$
	$Prod\ k\ (\lambda i.Rot\ i\ (Prod\ m\ (\lambda j.Base\ 1\ (kj))))$	$\mathcal{P}\ SwapRP$	(I_m)
	$Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Rot\ i\ (Base\ 1\ (kj))))$	$\mathcal{P}\ (\mathcal{P}\ DefineRot)$	(I_m)
	$Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Base\ 1\ (i + kj)))$	$SwapJoinProd$	L^n_k
	$Prod\ n\ (\lambda z.Base\ 1\ z)$	$-$	$-$

(6.5)

Decimation in Frequency, Negative Wrapped

The morphism sequence for a decimation-in-frequency algorithm does not require any modification to describe a negative-wrapped transformation. Changing the initial *Ring* and the value of N from eqn. (6.4) produces the following algorithm.

$(N = 2n)$	$Ring$	$Morphism$
	$Base\ n\ n$	$Factor\ k$
	$Prod\ k\ (\lambda i.Base\ m\ (m + 2mi))$	$\mathcal{P}\ (Factor\ m)$
	$Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Base\ 1\ (1 + 2i + 2jk)))$	$SwapJoinProd$
	$Prod\ n\ (\lambda z.Base\ 1\ (1 + 2z))$	$-$

(6.6)

Decimation in Frequency, Negative Wrapped, Recursive

Combining the previous two modifications to the decimation-in-frequency algorithm produces the following *Path*. The matrix $(T^*)^n_m$ is a modified version of the standard twiddle matrix, and is defined by the imperative implementation of *Normalize*.

$(N = 2n)$	$Ring$	$Morphism$	$Lin.\ Op.$
	$Base\ n\ n$	$Factor\ k$	$\mathcal{T}_k \otimes I_m$
	$Prod\ k\ (\lambda i.Base\ m\ (m + 2mi))$	$\mathcal{P}\ Norm$	$(T^*)^n_m$
	$Prod\ k\ (\lambda i.Rot\ (1 + 2i)\ (Base\ m\ 0))$	$\mathcal{P}\ (\mathcal{R}\ (Factor\ m))$	$I_k \otimes \mathcal{T}^{NW}_m$
	$Prod\ k\ (\lambda i.Rot\ (1 + 2i)\ (Prod\ m\ (\lambda j.Base\ 1\ (2kj))))$	$\mathcal{P}\ SwapRP$	(I_m)
	$Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Rot\ (1 + 2i)\ (Base\ 1\ (2kj))))$	$\mathcal{P}\ (\mathcal{P}\ DefineRot)$	(I_m)
	$Prod\ k\ (\lambda i.Prod\ m\ (\lambda j.Base\ 1\ (1 + 2i + 2kj)))$	$SwapJoinProd$	L^n_k
	$Prod\ n\ (\lambda z.Base\ 1\ (1 + 2z))$	$-$	$-$

(6.7)

Decimation in Time

Using decimation in time requires the addition of the *Label* function. This function is intended to increase the locality of memory accesses. The precise interaction between this function and cache efficiency is difficult to model and prompts the use of experimental optimization.

$(N = n)$	$Ring$	$Morphism$
	$Base\ n\ 0$	$Label\ k$
	$Exp\ k\ (Base\ m\ 0)$	$\mathcal{E}\ (Factor\ m)$
	$Exp\ k\ (Prod\ m\ (\lambda i.Base\ 1\ (ki)))$	$SwapEP$
	$Prod\ m\ (\lambda i.Exp\ k\ (Base\ 1\ (ki)))$	$DefineExp$
	$Prod\ m\ (\lambda i.Base\ k\ (ki))$	$\mathcal{P}\ (Factor\ k)$
	$Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Base\ 1\ (i + mj)))$	$SwapJoinProd$
	$Prod\ n\ (Base\ 1\ z)$	$-$

(6.8)

Decimation in Time, Recursive

As before, we may define a perfectly recursive version of the decimation-in-time algorithm. Notice by comparing the linear operators we have merely applied the identity

$A_m \otimes B_k = L^n_k(B_k \otimes A_m)L^n_m$ to eqn. (6.5). An abstracted representation of this identity through the ***Label*** isomorphism facilitates the generation of unique algorithms. For example, consider how the application of $Label\ k/l$ in the first step of the algorithm below could change the result.

$(N = n)$	$Ring$	$Morphism$	$Linear\ Operator$
	$Base\ n\ 0$	$Label\ k$	L^n_k
	$Exp\ k\ (Base\ m\ 0)$	$\mathcal{E}\ (Factor\ m)$	$I_k \otimes \mathcal{T}_m$
	$Exp\ k\ (Prod\ m\ (\lambda i.Base\ 1\ (ki)))$	$SwapEP$	L^n_m
	$Prod\ m\ (\lambda i.Exp\ k\ (Base\ 1\ (ki)))$	$DefineExp$	(I_n)
	$Prod\ m\ (\lambda i.Base\ k\ (ki))$	$\mathcal{P}\ Norm$	T^n_k
	$Prod\ m\ (\lambda i.Rot\ i\ (Base\ k\ 0))$	$\mathcal{P}\ (\mathcal{R}\ (Factor\ k))$	$I_m \otimes \mathcal{T}_k$
	$Prod\ m\ (\lambda i.Rot\ i\ (Prod\ k\ (\lambda j.Base\ 1\ (mj))))$	$\mathcal{P}\ SwapRP$	(I_n)
	$Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Rot\ i\ (Base\ 1\ (mj))))$	$\mathcal{P}\ (\mathcal{P}\ DefineRot)$	(I_n)
	$Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Base\ 1\ (i + mj)))$	$SwapJoinProd$	L^n_k
	$Prod\ n\ (\lambda i.Base\ 1\ z)$	$-$	$-$

(6.9)

Decimation in Time, Negative Wrapped, Recursive

As with decimation in frequency, the negative-wrapped and perfectly-recursive modifications may be combined into a single, decimation-in-time algorithm. This is left as an exercise for the reader.

Chapter 7

Evaluation

7.1 Algorithm Compilation

Several compilation pathways were used to generate C code from the algorithm description given by a *Path*. The first stage of each pathway applies the *compile* and *compile-inverse* functions which produces a symbolic representation of linear operators from *Morphisms*. These symbolic linear operators correspond to the capital Greek letters used in Chapter 4 and are used as a common intermediate representation for all compilation pathways.

The most direct compilation pathway groups symbolic linear operators into classes based on their sparse structure. These classes are; square, tensor-with-identity, diagonal, and permutation. Predefined function kernels implement each one of these classes, which are then adapted to a particular linear operator using parameters and precomputed constants. We call this pathway *Looped*.

Another set of compilation pathways unrolls all loop structures. This facilitates the application of peephole simplifications such as removing multiplication by one and introducing subtraction. There are two versions of this pathway, one applies peephole simplifications and another does not. These two versions are called *Peep* and *NoPeep* respectively.

All compilation pathways use a platform independent implementation of the Montgomery fast reduction algorithm [18] to perform operations in finite fields. This method is much less sophisticated than those found in Seiler [25] and Nguyen et al. [19].

The goal of the compilation pathways is not to produce state-of-the-art implementations of fast multiplication, but rather to evaluate the efficacy of the proposed program synthesis framework. For this reason, we value generality and diversity of generated code over platform specific optimizations. This is why the three compilation pathways isolate the effects of loop unrolling and peephole simplifications. While we do not aim for the state of the art, we would like our results to generalize to high performance implementations of post-quantum cryptographic algorithms. This

explains the choice of a platform independent implementation of Montgomery fast reduction. This implementation is certainly slower than those which exploit the ISA of their target platform, however, it shares the important characteristic that the reduction is made in constant time. Constant time reduction is a necessary feature of cryptographic implementations, to avoid some side channel attacks [15], and is also an important performance feature which is used to construct the model in eqn. (7.1).

7.2 Algorithm Optimization

The overarching goal of the methods proposed is to expose the process of designing fast multiplication algorithms to automated search through autotuning. As a result, we have focused our efforts on the development of a program synthesis framework and use a relatively simple optimization algorithm, simulated annealing [14], to search for optimal implementations. The ability of such a simple search method to find implementations which outperform human designs demonstrates the efficacy of this program synthesis approach. Applying more sophisticated optimization algorithms is left to future work.

The implementation of simulated annealing is relatively straightforward. We use program execution time as a cost measure, fix the number of epochs at 250, and randomly restart from the best implementation with a probability of one in twenty at each epoch. Like most other simulated annealing algorithms, the likelihood of accepting a new state is proportional to temperature, which decreases over time.

7.3 Experimental Results

Experimental results measuring the performance of synthesized algorithms were collected on an Intel® Xeon® Processor E3-1230 v2 machine. Binary code was produced using gcc version *8.5.0* using the *-O3* optimization flag.

All reported data concerns only the forward transformation. We focus on the results of the forward transformation for two reasons: simplifying the algorithm facilitates the study of compiler

optimizations, and the data presented is more directly comparable to other implementations of the Number Theoretic Transform. All optimized algorithms were produced using simulated annealing with SSA as an expansion function. All execution times are calculated using the mean of values in the interquartile range of a sample of 100 executions.

7.4 Baseline

For a point of comparison, we use baseline algorithms in which the only non identity *Morphism* is *Factor k* (with fixed k). This baseline represents the state of the art for NTT implementations in post-quantum cryptography. An example of such an algorithm is the one used by Seiler [25]. Within the FRI framework, Seiler's FFT algorithm corresponds to a homogeneous series of *(Factor 2)* isomorphisms.

The results of an initial study of baseline algorithms is given by Figures 7.1. The metric *Pseudo Ops* was developed by [28] to compare the performance of Fast Fourier Transformations across kernel sizes. Reporting using *Pseudo Ops* normalizes the execution time of each algorithm by calculating operations per second using the number of operations required by a naive implementation of FFT (rather than the true number of operations performed by the algorithm). This naive baseline is calculated as $5n\log(n)/t$ where n is the kernel size and t is the program execution time.

For baseline algorithms compiled using *Peep* and *NoPeep* the optimal *Factor* size is 2. For baseline algorithms compiled using *Looped* the optimal *Factor* size is 4 for most kernel sizes. This discrepancy is likely a result of a trade off between the number of field operations, which increases with *Factor* size, and the overhead incurred by the control structure of function calls, which decreases with *Factor* size. In the *Unrolled* code, there is little overhead cost incurred by the control structure so the smallest *Factor* size is the most successful. In the *Looped* case a *Factor* size of 4 appears to strike the optimal balance between the number field operations and the complexity of the control structure. The tendency of additional control structure to outweigh changes in other computation costs foreshadows our inability to optimize *Looped* code.

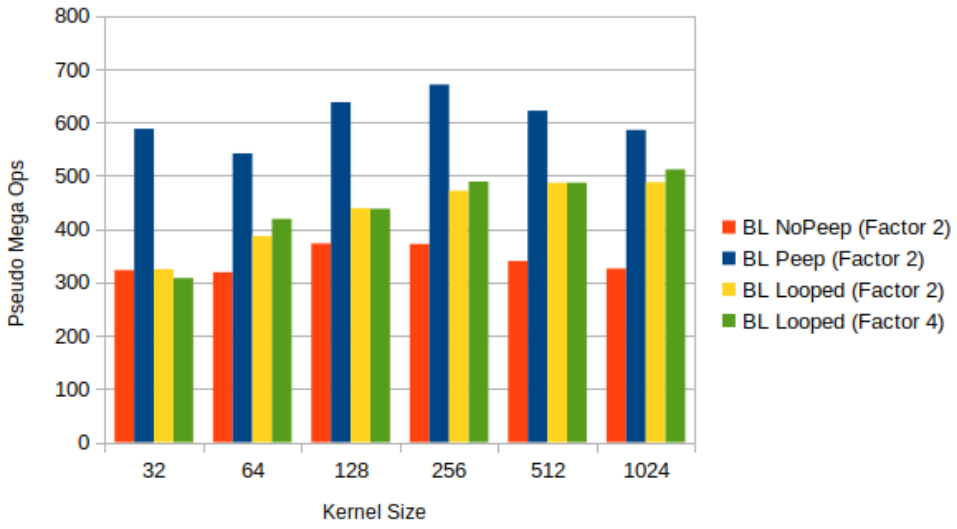

Figure 7.1: Performance of Baseline Algorithms

7.5 Optimization Results

The success of the optimization algorithm varies greatly depending on compilation method and kernel size. One of the more successful examples of the simulated annealing algorithm converging on a local optimum is given by Figure 7.3. Several baseline algorithms were used as initial states of the simulated annealing algorithm. In Figure 7.3 the initial state is a homogeneous *Factor 4* algorithm. For this reason the Speedup reported in Figure 7.3 is larger than the Speedup reported in Figure 7.6 which uses the faster, *Factor 2* algorithm as a baseline.

The results of optimizing *Looped* code is given by Figure 7.4. This figure shows only marginal improvement of the optimized code over the relevant baseline for most kernel sizes. This suggests that program synthesis is not a successful strategy for the *Looped* compilation pathway. Specifically, the *Looped* code cannot have fewer finite field operations than the homogeneous *Factor 2* algorithm because it lacks peephole simplifications. We know from the study of baseline algorithms that the control structure overhead of *Looped* code has the ability to outweigh changes in other computation costs. It is therefore likely that any benefits produced by modifying the homogeneous *Factor 2* algorithm for *Looped* code are outweighed by the cost incurred from the additional control structure associated with these modifications.

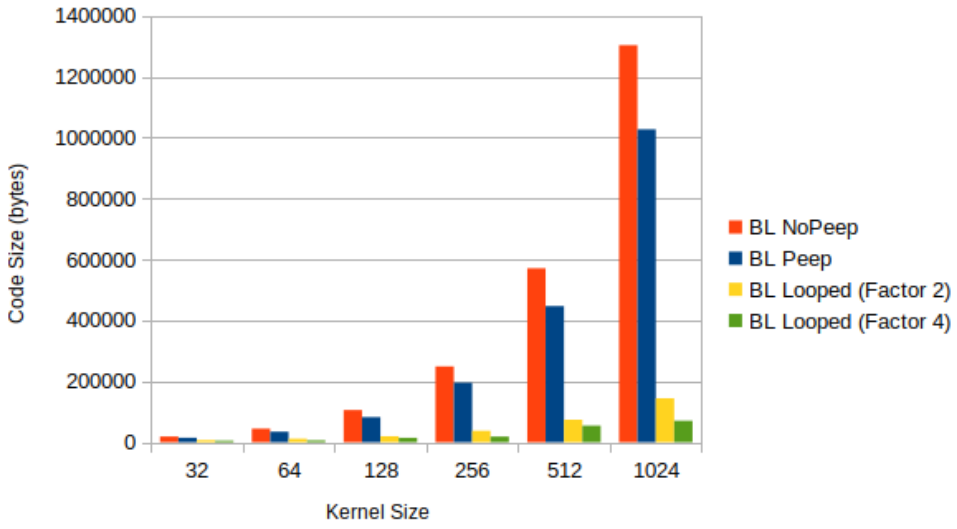

Figure 7.2: Code Size of Baseline Algorithms

Figure 7.5 presents the results of optimizing the unrolled code. The most successful optimizations were found for programs compiled using *Peep* which achieved speedups of up to 20% and consistently above 8% for large kernel sizes. In comparison with the *Looped* version of the code, these performance improvements may be due to a reduction in the number of finite field operations or better memory access patterns. It is therefore useful to consider the optimization improvement produced by the *NoPeep* compilation pathway, to isolate the effect of reducing the number of field multiplications. Optimizations made to *NoPeep* were less successful than the pathway containing peephole simplifications but still consistently above 7% for most kernel sizes. A comparison between the speedup values of the *Peep* and *NoPeep* is given by Figure 7.6.

7.6 Modeling Performance

As we added more compilation optimizations, such as loop unrolling and peephole simplification, the effectiveness of program synthesis grew. This progression is evident in the superior speedup values: *UnrolledNoPeep* shows an improvement compared to *Looped*, and *UnrolledPeep* outperforms *UnrolledNoPeep*. While this trajectory is promising, there is a trade-off. Incorporating these optimizations increases the code size, as depicted in Figure 7.2, limiting our ability to

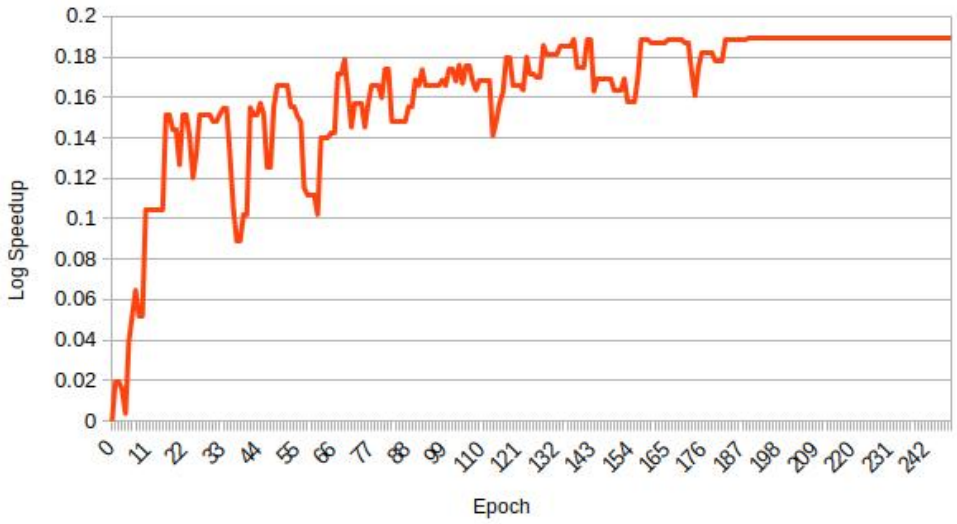

Figure 7.3: Convergence of Simulated Annealing Algorithm
Kernel Size: 1024
Compiler: *Peep*

handle larger kernel sizes. As a result, modeling the sources of performance enhancement becomes crucial to more effectively focus compiler optimizations.

Most NTT algorithms spend the majority of their execution time on finite field multiplication and the resulting modular reduction step. If we assume that each finite field multiplication contributes a constant amount of time (α) to the overall execution of the algorithm, we can determine α and use this value to better understand the performance improvements of each algorithm. This assumption leads to the following model (eqn. (7.1)) of execution time where E is the execution time, M is the number of finite field multiplications, αM is the time spent as a result of multiplication, and G is the remainder of the execution time. Each of these values is a function of n: the kernel size.

$$E = \alpha M + G \tag{7.1}$$

Using this model, we determine the value of α by comparing the values of the baseline *UnrolledNoPeep* and *UnrolledNoPeep* codes. Using α we can determine the fraction of time spent as

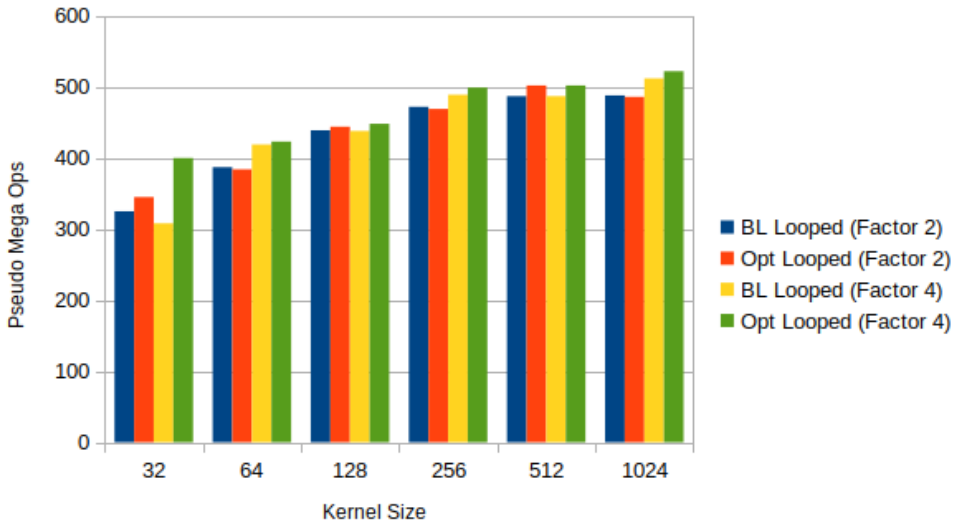

Figure 7.4: Performance of Looped Code

a result of finite field multiplication and the absolute amount of time spent on the rest of the code (G). All of these values are shown in Table 7.1. The equations which produced each cell of this table are given in Appendix D.

Table 7.1: Performance Model

Kernel Size	Fraction Mult. Peep	Fraction Mult. NoPeep	α	G_0
32	82.06%	90.15%	6.97E-09	2.44E-07
64	70.06%	82.39%	6.46E-09	1.06E-06
128	70.94%	83.00%	5.56E-09	2.04E-06
256	80.32%	89.08%	5.98E-09	3.00E-06
512	82.79%	90.58%	6.65E-09	6.37E-06
1024	79.74%	88.73%	6.80E-09	1.77E-05

In Table 7.1, the value of α fluctuates slightly as kernel size increases, but remains relatively constant overall, suggesting this quantity is somewhat universal. Using the model given in eqn. (7.1) and the value α, we can identify the source of performance improvements made to the unrolled code by the program synthesis engine. Specifically, we can separate the effect of reducing the number of multiplications from other improvements, such as better memory access patterns.

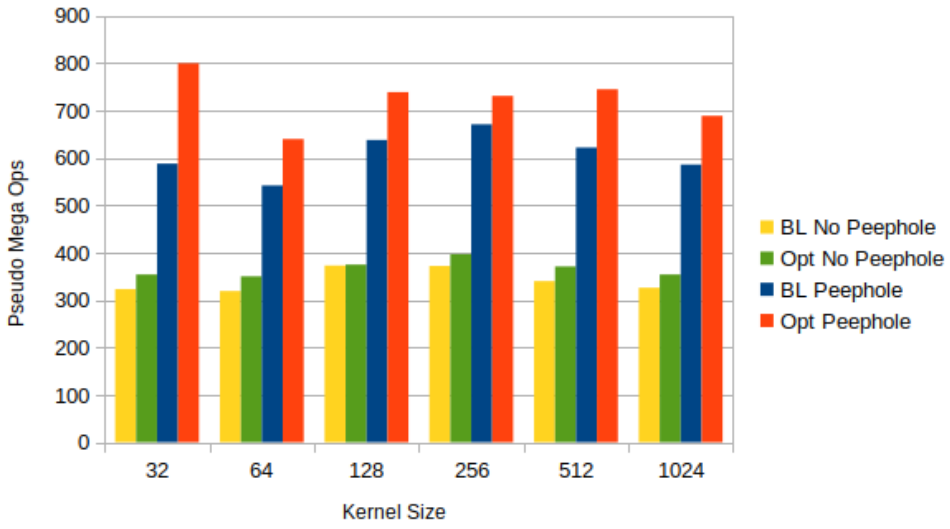

Figure 7.5: Performance of Unrolled Code

Within the model, the function G represents the execution time which is independent of the number of multiplications, so we are concerned with the speedup of G. The baseline for this speedup calculation is the value of G for the *NoPeep* and *Peep* baseline codes; which is assumed to be the same for both code sets and is column G_0 in Table 7.1.

Figure 7.7 displays the values of *G Speedup* for the optimized *Peep* and *NoPeep* codes. This perspective presents the improvements made to the *NoPeep* kernels in a more positive light. When we take into account that over 80% of the execution time is fixed (because *NoPeep* does not reduce the number of multiplications) the program synthesis engine has sped up the remaining time by as much as 6X, and over 2.5X for all kernel sizes over 256. The program synthesis engine increased G for *Peep* for most kernel sizes, likely because G has a relatively small effect when compared with the number of multiplications.

The differences between the optimization of *Peep* and *NoPeep* can be exemplified by the best *Paths* found for each compilation method. Over every optimized *Path* for *Peep* in the data set presented above, the *Norm* isomorphism is used 16 times, compared with zero times for the *NoPeep* compiler. This data suggests that *Normalize* may only improve performance when peephole simplifications are applied, otherwise, *Normalize* increases the number of multiplications. The decrease

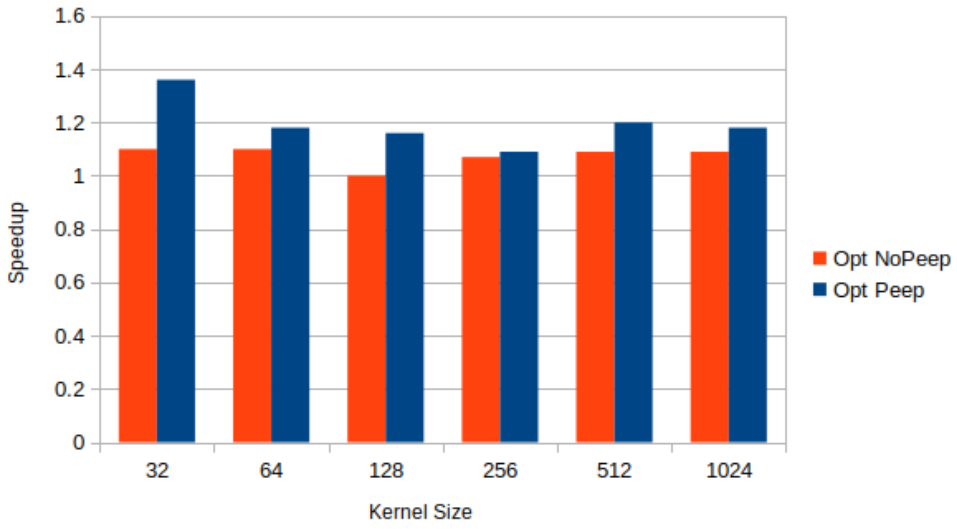

Figure 7.6: Speedup of Unrolled, Optimized Code

in the number of multiplications for *Paths* compiled using *Peep* is presented in Figure 7.8. This figure measures the number of multiplications relative to the baseline *NoPeep* algorithms, which is $2n\log(n)$.

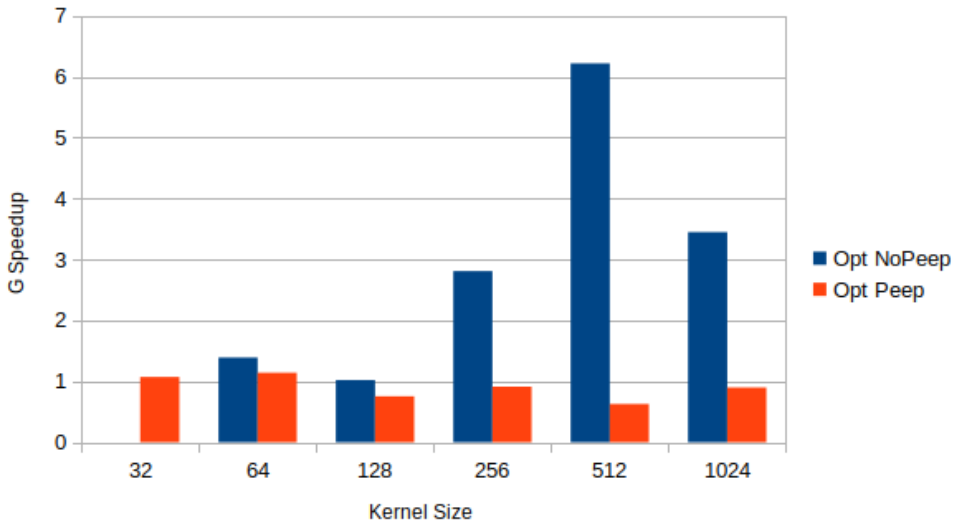

Figure 7.7: Multiplication Independent Improvement (G Speedup) of Unrolled Code

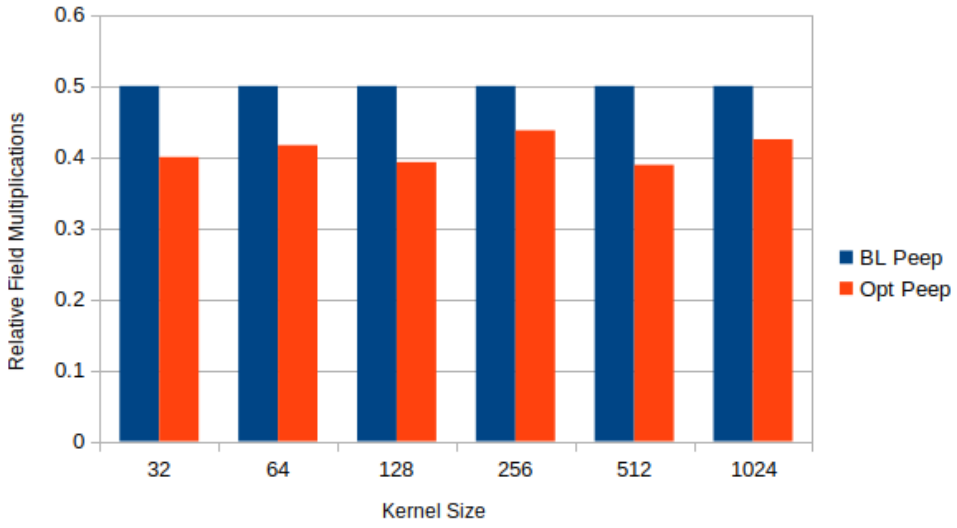

Figure 7.8: Relative Field Multiplications of Unrolled Algorithms

Chapter 8

Future Work

The ring isomorphism framework can be generalized to include other fast Fourier algorithms. These include the split radix FFT [10], multidimensional FFTs, and Bluestein's algorithm [4]. In addition, the equivalence classes over *Morphism* series produced for the suggestion search algorithm deserve better study. It is likely these suggested identities exhibit useful structure which may aid in their discovery and application. Finally, the imperative definitions given by Chapter 4 were developed by hand through a combination of derivation and observation. A method to discover these implementations automatically given only the signature of the ring isomorphism they implement would be extremely useful.

Dependent types in Agda [27] offer the capability to embed and enforce specific properties directly within the type system, such as divisibility constraints in the context of ring isomorphisms. By refining the *Ring* type with dependent types, constraints like ensuring valid *Product* structures or *Rotation* factors can be represented. A significant portion of the current implementation relies on the assumption that an input *Ring* is correctly constructed. As a result, this system will produce undefined behavior when illegal *Ring* structures are manipulated or compiled. Scaling the current system to a more complex types will likely require the addition of dependent types to ensure stability.

The code generation processes in this work are severely limited. They merely aim to evaluate the program synthesis method presented. We believe that combining our program synthesis engine with more sophisticated code generation techniques can produce state-of-the-art software and hardware implementations for fast multiplication and fast Fourier transformations. A more sophisticated code generation process might include; a representation of parallelism, generalization to any scalar structure, hardware specific instructions, and linear operator fusion.

An enhanced search algorithm would likely find optimized implementations more quickly. A study of related work in autotuning [6] suggests the use of machine learning methods combined with learned cost measures might be successful for this application.

There are two important theoretical pieces of this work which are missing. The fist is a proof that the system of FRIs is Church-Rosser [8] when the isomorphism *JoinProd* is removed. We can show this up to *Product* permutation, and conjecture this to be the case from observation, but have not completed this proof. Having such a proof would allow this methods to be directly applied to the synthesis of DFT algorithms. The second is a formal comparison with the Spiral [11] system. Any language which can describe all the rewrite rules used in a Term Rewrite System, can at least describe the same set of objects. We have shown that we can synthesize some of the rewrite rules used by Spiral for the DFT, but not all. We also do not know if our system can describe objects which are not reachable by rewrite rules currently used in Spiral, or through the addition of trivial new rules. Such an understanding would require measuring the size of the span of the objects described by our language.

Chapter 9

Conclusion

We have demonstrated the potential of typed synthesis in optimizing polynomial ring multiplication algorithms, a cornerstone operation in many post-quantum cryptosystems. Integrating program synthesis with an algebraic framework produced a novel approach that allows automated optimization of multiplication algorithms across diverse computational platforms.

The introduction of a refined framework, which considers each step of the Fast Fourier Transform (FFT) as a ring isomorphism rather than a linear operator, marks a significant advancement. This abstraction not only simplifies the synthesis process but also expands the applicability to a broader class of ring structures, relevant to cryptosystems like Crystals Kyber [5] and NewHope [2]. The elimination of unnecessary permutations in the codomain and the incorporation of automated search techniques further underscore the efficacy and versatility of the proposed method.

The implementation, which translates algebraic structures to a Haskell-based type system, exemplifies the practicality of the theoretical framework. This approach highlights the synergy between symbolic mathematics and program synthesis via category theory, the applications of which extend beyond polynomial ring isomorphisms to a more general class of programming abstractions, such as the polyhedral model [21].

Our evaluation underlines the effectiveness of the proposed methods in enhancing algorithmic performance, demonstrating improvements over existing state-of-the-art algorithms. These findings not only validate the proposed approach but also open up new horizons for research in synthesized implementations of post-quantum cryptography.

Bibliography

[1] *Fourier Transform Methods in Finance*. John Wiley & Sons, Ltd, 2012.

[2] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. NewHope. https://pq-crystals.org/dilithium/, 2020.

[3] Daniel J Bernstein. Multidigit multiplication for mathematicians. 2001.

[4] L. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

[5] Joppe Bos, Leo Ducas, Eike Kiltz, T. Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *Proceedings - 3rd IEEE European Symposium on Security and Privacy, EURO S and P 2018*, pages 353–367. IEEE, Jul 2018.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[7] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.

[8] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[9] James W Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series, 1965.

[10] Pierre Duhamel and Henk D. L. Hollmann. 'Split radix' FFT algorithm. *Electronics Letters*, 20:14–16, 1984.

[11] F. Franchetti et al. SPIRAL: Extreme performance portability. In *Proceedings of the IEEE, vol. 106, no. 11, pp. 1935-1968*. IEEE, Nov. 2018.

[12] Charles M Fiduccia. Polynomial evaluation via the division algorithm the fast Fourier transform revisited. *Fourth annual ACM symposium on theory of computing*, 1972.

[13] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, 17(17), Aug 2020.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[15] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO'96*, pages 104–113. Springer, Berlin, Heidelberg, 1996.

[16] Saunders Mac Lane. *Categories for the Working Mathematician: Second Edition*. Springer, 1998.

[17] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[18] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519, 1985.

[19] Duc Tri Nguyen and Kris Gaj. Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing.

[20] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In Kristin Lauter and Francisco

Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365, Cham, 2015. Springer International Publishing.

[21] Sanjay Rajopadhye. The polyhedral model. In *PPoPP'07: Symposium on Principles and Practice of Parallel Programming*, 2007.

[22] O. Regev. On Lattices, Learning With Errors, Random Linear Codes, and Cryptography. *Symposium on Theory of Computing*, pages 84–93, 2005.

[23] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*, page 371–391, Berlin, Heidelberg, 2014. Springer-Verlag.

[24] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing 7*, pages 281–292, 1971.

[25] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.

[26] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[27] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. https://plfa.inf.ed.ac.uk/22.08/.

[28] Jianxin Xiong, Jeremy Johnson, Robert Johnson, David Padua, and Mathstar Inc. SPL: A language and compiler for DSP algorithms. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 36, 05 2001.

Appendix A

Proof of Inverse *Factor* Definition

The following is a proof the implementation of $\Phi^{-1}_{n,k,d}$ given by eqn. (4.3) correctly computes the inverse of $\Phi_{n,k,d}$. We do so by showing $\Phi^{-1}_{n,k,d}(A)_{im+j}$ recovers a_{im+j} where A is attained through the forward application of $\Phi_{n,k,d}$ on a.

$$Let\ A_{z,j} = \Phi_{n,k,d}(a)_{z,j} = \sum_{i=0}^{k-1} a_{im+j}\omega_N^{(d+zN)i/k} \tag{A.1}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = k^{-1}\sum_{z=0}^{k-1} A_{z,j}\omega_N^{-(d/k+zN/k)i} \tag{A.2}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = k^{-1}\sum_{z=0}^{k-1}\left(\sum_{l=0}^{k-1} a_{lm+j}\omega_N^{(d/k+zN/k)l}\right)\omega_N^{-(d/k+zN/k)i} \tag{A.3}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = k^{-1}\sum_{l=0}^{k-1}\sum_{z=0}^{k-1} a_{lm+j}\omega_N^{(d/k+zN/k)l-(d/k+zN/k)i} \tag{A.4}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = k^{-1}\sum_{l=0}^{k-1}\sum_{z=0}^{k-1} a_{lm+j}\omega_N^{(d/k)(l-i)}\omega_N^{(zN/k)(l-i)} \tag{A.5}$$

$$\Phi^{-1}_{n,k,d}(A)_{im+j} = k^{-1}\sum_{l=0}^{k-1}\left(a_{lm+j}\omega_N^{(d/k)(l-i)}\sum_{z=0}^{k-1}\omega_N^{(zN/k)(l-i)}\right) \tag{A.6}$$

Consider the following sub-expression from (A.6) which we will label as the function $\gamma(l,i)$.

$$\gamma(l,i) = \sum_{z=0}^{k-1}\omega_N^{(zN/k)(l-i)}$$

Using the identity $\omega_N^{N/k} = \omega_k$ we can transform this sub-expression.

$$\gamma(l,i) = \sum_{z=0}^{k-1}\omega_k^{z(l-i)}$$

Furthermore, consider the case when $l = i$. Under this condition this sub-expression simplifies to the following summation.

$$\gamma(l, i) = \sum_{z=0}^{k-1} \omega_k^0 = k$$

In the other case, when $l \neq i$, we can use the following identity (eqn. (A.7) proven in Section A.1) by setting $\beta = l - i$. Notice that the constraint $i, l \in [0, k-1]$ allows $l \neq i$ to imply $l - i \not\equiv 0 \mod k$.

$$\forall \beta \not\equiv 0 \mod k : \sum_{z=0}^{k-1} \omega_k^{\beta z} = 0 \tag{A.7}$$

Therefore, when we make the assumption that $l \neq i$ and apply identity (A.7) we find that $\gamma(l, i) = 0$. Using the two cases considered ($l = i$ and $l \neq i$) we can rewrite $\gamma(l, i)$.

$$\gamma(l, i) = k\delta_{i,l}$$

(Where δ is the Kronecker Delta function)

Replacing the rewrite of $\gamma(l, i)$ in (A.6) produces the following.

$$\Phi_{n,k,d}^{-1}(A)_{im+j} = k^{-1} \sum_{l=0}^{k-1} a_{lm+j} \omega_N^{(d/k)(l-i)} k\delta_{i,l} \tag{A.8}$$

Inspecting this summation, we find that when $l \neq i$, the $\delta_{i,l}$ term causes the entire summand to become zero. Therefore, we only consider the summand element where $l = i$.

$$\Phi_{n,k,d}^{-1}(A)_{im+j} = k^{-1} a_{im+j} \omega_N^{(d/k)(i-i)} k$$

$$\Phi_{n,k,d}^{-1}(A)_{im+j} = a_{im+j}$$

□

A.1 Proof of Equation (A.7)

This is a well known property of roots of unity but is proven here for completeness.

$$
\begin{aligned}
S &= \sum_{z=0}^{k-1} \omega_k^{\beta z} \\
\omega_k^{\beta} S &= \sum_{z=0}^{k-1} \omega_k^{\beta(z+1)} \\
\omega_k^{\beta} S &= \omega_k^{\beta k} + \sum_{z=1}^{k-1} \omega_k^{\beta z} \\
\omega_k^{\beta} S &= \omega_k^{0} + \sum_{z=1}^{k-1} \omega_k^{\beta z} \\
\omega_k^{\beta} S &= \sum_{z=0}^{k-1} \omega_k^{\beta z} \\
\omega_k^{\beta} S &= S \\
S(\omega_k^{\beta} - 1) &= 0
\end{aligned}
$$

Assume $\beta \not\equiv 0 \mod k$. Then $\omega_k^{\beta} - 1 \neq 0$. With this assumption, $S = 0$. We may summarize this as follows.

$$\forall \beta \not\equiv 0 \mod k : \sum_{z=0}^{k-1} \omega_k^{\beta z} = 0 \tag{A.9}$$

□

Appendix B

Derivation of *SwapER* (θ^{ER})

The following is a derivation of the function θ^{ER} given by eqn. (4.13). The ring structure describing the domain of θ^{ER} admits the following identities.

$$Y = \omega_N^d W \tag{B.1}$$

$$X^n = Y \tag{B.2}$$

We can combine this system into a single identity which describes X using W.

$$X^n = \omega_N^d W \tag{B.3}$$

We would like to define a variable V with the property $V^n = W$. We can achieve this by rearranging the previous equation into the following form. This operation is legal because $n|d$.

$$(\omega_N^{-d/n} X)^n = W \tag{B.4}$$

By setting $V = \omega_N^{-d/n} X$, we have the following identities.

$$V^n = W \tag{B.5}$$

$$X = \omega_N^{d/n} V \tag{B.6}$$

Eqns. (B.5) and (B.6) are then used to construct the codomain of θ^{ER} in eqn. (4.13).

Appendix C

Proof that a *Path* must be Finite

To show that a legal *Path* must be finite, it suffices to show that the number of occurrences of all FRI classes ($\phi_k,\ \xi_k,\ \psi$, etc.) in the composition sequence is finite. This is accomplished with the help of two metric functions $\mathcal{Q}$ and $\mathcal{M}$, defined over *Ring*.

$$\mathcal{Q} : Ring \rightarrow (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \tag{C.1}$$

$$\mathcal{Q}(S) = (\mathcal{Q}_P(S), \mathcal{Q}_E(S), \mathcal{Q}_R(S)) \tag{C.2}$$

$$\mathcal{M} : Ring \rightarrow (\mathbb{N} \times \mathbb{N}) \tag{C.3}$$

$$\mathcal{M}(S) = (\mathcal{M}_P(S), \mathcal{M}_E(S), \mathcal{M}_B(S)) \tag{C.4}$$

The functions $\mathcal{Q}_P$, $\mathcal{Q}_E$, and $\mathcal{Q}_R$ count the number of occurrences of each surrounding structure in a *Ring*. Specifically; $\mathcal{Q}_P$ counts *Products*, $\mathcal{Q}_E$ counts *Expansions*, and $\mathcal{Q}_R$ counts *Rotations*.

The functions $\mathcal{M}_P$, $\mathcal{M}_E$, and $\mathcal{M}_B$ are defined as follows. The metric function $\mathcal{M}_P$ represents the product over the dimensions of all $Products$ in a *Ring*. The metric function $\mathcal{M}_E$ represents the product over the dimensions of all $Expansions$ in a *Ring*. The function $\mathcal{M}_B$ represents the dimension of the inner $Base$ term in a *Ring*. Examples of $\mathcal{M}_P$, $\mathcal{M}_E$, and $\mathcal{M}_B$ are given below.

$$\mathcal{M}_P(Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Exp\ t\ (Base\ l\ mj)))) = m * k \tag{C.5}$$

$$\mathcal{M}_E(Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Exp\ t\ (Base\ l\ mj)))) = t \tag{C.6}$$

$$\mathcal{M}_B(Prod\ m\ (\lambda i.Prod\ k\ (\lambda j.Exp\ t\ (Base\ l\ mj)))) = l \tag{C.7}$$

Only three of the fundamental ring isomorphisms have an effect on the metric space of $\mathcal{M}$. These isomorphisms are $\phi_k,\ \xi_k$, and δ^E; their effects are described below.

$$\mathcal{M}(\phi_k(S)) = \mathcal{M}(S) * (k, 1, 1/k) \tag{C.8}$$

$$\mathcal{M}(\xi_k(S)) = \mathcal{M}(S) * (1, k, 1/k) \tag{C.9}$$

$$\mathcal{M}(\delta^E(Exp\ k\ S)) = \mathcal{M}(Exp\ k\ S) * (1, 1/k, k) \tag{C.10}$$

Define the total dimension of a ***Ring*** S to be the product of $\mathcal{M}_P(S)$, $\mathcal{M}_E(S)$, and $\mathcal{M}_B(S)$. The total dimension is a universal invariant of all ***Rings*** in an FRI composition sequence.

Consider a ***Ring*** S_0 with total dimension $D \in \mathbb{N}$. With minor loss of generality, we assume the encoding of S_0 is finite. This implies the values of the metric functions $\mathcal{Q}$ and $\mathcal{M}$ are bounded on S_0, as is the value D.

Define the domain G_{S_0} to be the set of ring structures reachable from S_0 through a composition sequence of FRIs. The image $\mathcal{M}(G_{S_0})$ is bounded by the following inequalities.

$$(i, j) \in \mathcal{M}(G_{S_0})$$

$$i * j \leq D \tag{C.11}$$

$$\mathcal{M}_P(S_0) \leq i \leq D \tag{C.12}$$

$$1 \leq j < D/\mathcal{M}_P(S_0) \tag{C.13}$$

Equation (C.11) comes from the fact that the total dimension is constant. Equation (C.12) may be inferred from the following; there is no fundamental ring isomorphism which decreases $\mathcal{M}_P$, equation (C.11), and a naive lower bound on $\mathcal{M}_E(G_{S_0})$. Finally, equation (C.13) follows from eqns. (C.11) and (C.12), as well as the same naive lower bound on $\mathcal{M}_E(G_{S_0})$.

Consider a composition sequence of fundamental ring operations H, which leads from S_0 to S^*. By definition, $\mathcal{M}(S^*) \in \mathcal{M}(G_{S^0})$. The notation $(f)^H$ represents the set of occurrences of the FRI f in the composition H.

We can conclude the value $|(\phi_k)^H|$ is bounded using the following reasoning. There is no fundamental ring isomorphism which decreases $\mathcal{M}_P$ and the value of $\mathcal{M}_P(S^*)$ is bounded, therefore the value of $\mathcal{M}_P$ is increased a finite number of times in the composition H. Every occurrence of the function ϕ_k in H increases the value of $\mathcal{M}_P$, therefore $|(\phi_k)^H|$ is finite.

We can bound $|(\delta^E)^H|$ by considering the domain $\mathbf{B}^1$: the set of ring structures with metric points in the domain $\{(i,j) \in \mathcal{M}(G_{S_0}) \mid i * j = D\}$. All ring structures in $\mathbf{B}^1$ contain a $Base$ of dimension of 1, e.g., $Base\ 1\ d$. Notice that ϕ_k is the only FRI which can map ring structures from $\neg\mathbf{B}^1$ to elements in $\mathbf{B}^1$. In addition, δ^E is the only FRI which can map elements in $\mathbf{B}^1$ to elements in $\neg\mathbf{B}^1$ and this is a universal property of δ^E. Therefore, if both S_0 and S^* are in $\neg\mathbf{B}^1$, then for each occurrence of δ^E in H there is a unique, corresponding occurrence of ϕ_k. Using this correspondence we can conclude $|(\delta^E)^H|$ is bounded. Figure C.1 displays an example sequence adhering to the conditions on S_0 and S^*. The cases when S_0 and S^* are allowed to be in $\mathbf{B}^1$ lead to the same result.

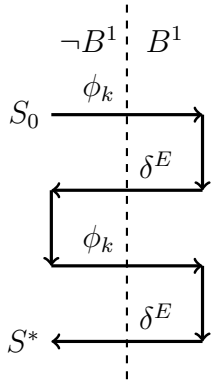

Figure C.1: An example composition sequence demonstrating the bound on $|(\delta^E)^H|$.

We can use the bound on $|(\delta^E)^H|$ to bound $|(\xi_k)^H|$ in the following way. With a bound on $|(\delta^E)^H|$, there are a limited number of steps in H which decrease the value of $\mathcal{M}_E$, and the value of $\mathcal{M}_E(S^*)$ is bounded, therefore the value of $\mathcal{M}_E$ is increased only a finite number of times in H. Every occurrence of ξ_k increases the value of $\mathcal{M}_E$, therefore $|(\xi_k)^H|$ is finite.

The value of $|(\psi)^H|$ may be bounded by considering the domain $\mathbf{d}^0$, the set of ring structures with a $Base$ in normal form: $(Base\ n\ 0)$. Notice that ψ is the only FRI that maps elements from $\neg\mathbf{d}^0$ to elements in $\mathbf{d}^0$, and this is a universal property of ψ. In addition, ψ can only be applied to elements in $\neg\mathbf{B}^1$. For this reason, consider the set of longest possible sub-sequences of H for which the mapped ring structures are in $\neg\mathbf{B}^1$, call this set $H_{\neg B^1}$. There are finitely many such sub-sequences, because $|(\delta_E)^H|$ has been bounded. Let $c_{\neg B^1}$ be an element of $H_{\neg B^1}$. With the condition that the mapped ring structures are in $\neg\mathbf{B}^1$, ϕ_k is the only FRI which can map elements in $\neg\mathbf{d}^0$ to elements in $\mathbf{d}^0$ (without this condition, we would need to consider δ^R, refer to Figure C.2). Therefore, if both the start and end ring structures of $c_{\neg B^1}$ are in $\neg\mathbf{d}^0$, then for every occurrence of ψ in $c_{\neg B^1}$ there is a unique, corresponding occurrence of ϕ_k in $c_{\neg B^1}$. Notice, $|(\phi_k)^{c_{\neg B^1}}|$ is bounded. The previous two properties together bound $|(\psi)^{c_{\neg B^1}}|$. The sequence $c_{\neg B^1}$ represents an arbitrary element of $H_{\neg B^1}$ and the set $H_{\neg B^1}$ contains only finitely many elements, therefore the bound on $|(\psi)^{c_{\neg B^1}}|$ implies a bound on $\sum_{c_{\neg B^1}}^{H_{\neg B^1}} |(\psi)^{c_{\neg B^1}}|$. The morphism ψ may only be applied to elements of $\neg B^1$, therefore the bound on $\sum_{c_{\neg B^1}}^{H_{\neg B^1}} |(\psi)^{c_{\neg B^1}}|$ implies a bound on $|(\psi)^H|$. The cases when the start and end ring structures of $c_{\neg B^1}$ are allowed to be in $\mathbf{d}^0$ lead to the same result.

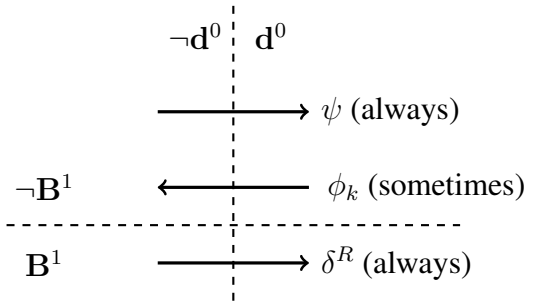

Figure C.2: A summary of the functions ψ, ϕ_k and δ^R with respect to the domains $\mathbf{d}^0$ and $\mathbf{B}^1$.

The value $|(\delta^R)^H|$ is trivially bounded using a constraint on $|(\psi)^H|$. Each occurrence of ψ adds a *Rotation* surrounding ring structure, and ψ is the only FRI to do so. The function δ^R must remove a *Rotation* surrounding ring structure, so $|(\delta^R)^H| \leq |(\psi)^H| + \mathcal{Q}_R(S_0)$. In other words, the total number of $Rotations$ removed in H may not exceed the total number of $Rotations$ produced in

H plus the total number of $Rotations$ which exist at the start of H. We will reuse this reasoning several times in the remainder of the proof to the extent that it is useful to label this quantity.

$$\mathbf{\Phi}\mathcal{Q}_P(H) = |(\phi_k)^H| + \mathcal{Q}_P(S_0) \tag{C.14}$$

$$\mathbf{\Phi}\mathcal{Q}_E(H) = |(\xi_k)^H| + \mathcal{Q}_E(S_0) \tag{C.15}$$

$$\mathbf{\Phi}\mathcal{Q}_R(H) = |(\psi_k)^H| + \mathcal{Q}_R(S_0) \tag{C.16}$$

The notation $\mathbf{\Phi}\mathcal{Q}(H)$ represents the total number of unique surrounding structures which may exist in H, which we will call the *maximum flux* of H. Formally identifying uniqueness would require the addition of labels to the simplified representation. This use of $\mathbf{\Phi}$ has no correspondence with the imperative definition of ϕ from eqn. (4.2).

The *Join* functions (ρ, τ, μ, and σ) all combine two structures of the same type, reducing the number of that type. This allows us to bound the occurrences of these isomorphisms by considering the maximum flux of H.

$$|(\rho)^H| + |(\tau)^H| < \mathbf{\Phi}\mathcal{Q}_P(H) \tag{C.17}$$

$$|(\mu)^H| < \mathbf{\Phi}\mathcal{Q}_E(H) \tag{C.18}$$

$$|(\sigma)^H| < \mathbf{\Phi}\mathcal{Q}_R(H) \tag{C.19}$$

The *Swap* functions (ζ^{EP}, η^{RP}, and θ^{ER}) can be viewed as a strict total order of *Product*, *Rotation*, and *Expansion* (see Fig. C.3). This order ensures that any pair of structures cannot be swapped more than once. Knowing this, we can bound the occurrences of these isomorphisms by considering the maximum flux of H. These bounds are given by eqns. (C.20) through (C.22).

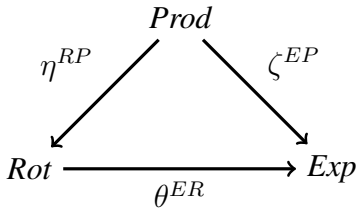

Figure C.3: The strict total order implied by the swap functions ζ^{EP}, η^{RP}, and θ^{ER}

$$|(\zeta^{EP})^H| \leq \min(\ \mathbf{\Phi}\mathcal{Q}_E(H),\ \mathbf{\Phi}\mathcal{Q}_R(H)\ ) \tag{C.20}$$

$$|(\eta^{RP})^H| \leq \min(\ \mathbf{\Phi}\mathcal{Q}_R(H),\ \mathbf{\Phi}\mathcal{Q}_P(H)\ ) \tag{C.21}$$

$$|(\theta^{ER})^H| \leq \min(\ \mathbf{\Phi}\mathcal{Q}_E(H),\ \mathbf{\Phi}\mathcal{Q}_R(H)\ ) \tag{C.22}$$

After bounding the number of occurrences of each FRI type in an unconstrained composition sequence of FRIs, we conclude that a *Path* must be finite.

□

Appendix D

Performance Model Calculations

The data presented in Table 7.1 is based on the performance model in eqn. (7.1). In addition to the assumptions used to construct the performance model, these calculations assume the value of G is the same for the baseline $NoPeep$ and $Peep$ codes.

The calculations are presented eqns. (D.1) through (D.5) and use eqn. (7.1). The subscripts $_P$ and $_{NP}$ refer to the $Peep$ and $NoPeep$ samples respectively.

$$M_P(N) = n\log(n) \tag{D.1}$$

$$M_{NP}(N) = 2n\log(n) \tag{D.2}$$

$$\alpha(n) = \frac{E_{NP}(n) - E_P(n)}{n\log(n)} \tag{D.3}$$

$$G(n) = E_P(n) - \alpha(n)\ n\log(n) \tag{D.4}$$

$$G(n) = E_{NP}(n) - \alpha(n)\ 2n\log(n) \tag{D.5}$$