Number Theoretic Transform

Thomas Ammer and Katharina Kreuzer

March 17, 2025

Abstract

This entry contains an Isabelle formalization of the *Number Theoretic Transform (NTT)* which is the analogue to a *Discrete Fourier Transform (DFT)*, just over a finite field. Roots of unity in the complex numbers are replaced by those in a finite field.

First, we define both *NTT* and the inverse transform *INTT* in Isabelle and prove them to be mutually inverse.

DFT can be efficiently computed by the recursive *Fast Fourier Transform (FFT)*. In our formalization, this algorithm is adapted to the setting of the *NTT*: We implement a *Fast Number Theoretic Transform (FNTT)* based on the Butterfly scheme by Cooley and Tukey [1]. Additionally, we provide an inverse transform *IFNTT* and prove it mutually inverse to *FNTT*.

Afterwards, a recursive formalization of the *FNTT* running time is examined and the famous $\mathcal{O}(n \log n)$ bounds are proven.

Contents

1 Introduction

The *Discrete Fourier Transform (DFT)* is used to analyze a periodic signal given by equidistant samples for its frequencies. For an introduction to *DFT* one may have a look at [2]. However, one may generalize the setting and consider any algebraic structure with roots of unity. For finite fields, we call the analogue to *DFT* a *Number Theoretic Transform (NTT)*. It can be used for fast Integer multiplications and post-quantum lattice-based cryptography [3].

Starting our formalization, we provide some initial setup, namely roots of unity by an argument on generating elements in $\mathbb{Z}_p$ (Sections 2.1, 2.2, 2.3) and lemmas on summation (Section 2.4), especially geometric sums (Section 2.5).

We continue with a mathematical definition of *NTT* [4] and formalize it in Isabelle (Section 3.1). Let us consider a definition of *DFT*:

$$\mathsf{DFT}(\vec{x})_k = \sum_{l=0}^{n-1} x_l \cdot e^{-\frac{i2\pi}{n} \cdot k \cdot l} \qquad \text{where } i = \sqrt{-1}$$

In this equation, $e^{-\frac{i2\pi}{n}}$ is a root of unity. Let ω be a n-th root of unity in $\mathbb{Z}_p$ and we can state analogously:

$$\mathsf{NTT}(\vec{x})_k = \sum_{l=0}^{n-1} x_l \cdot \omega^{kl}$$

Throughout the paper, we stick to this definition. An inverse tranform *INTT* is obtained by replacing ω by its field inverse μ (i.e. $\mu \cdot \omega \equiv 1 \mod p$). We prove *NTT* and *INTT* to be mutually inverse in Section 3.2.

For computing *DFT* more efficiently than $\mathcal{O}(n^2)$, a divide and conquer approach can be applied. By a smart rearranging, the sum can be split into two subproblems of size $\frac{n}{2}$ which gives an $\mathcal{O}(n \log n)$ algorithm. We call this the *Fast Nuber Theoretic Transform (FNTT)* [3] and *IFNTT* respectively. The corresponding procedure is treated in Section 4. We prove equality between *(I)NTT* and *(I)FNTT* and can infer that both are mutually inverse by previos results.

DFT and similar transforms like *NTT* are especially famous for algorithms with $\mathcal{O}(n \log n)$ running times. Thus, it is appropriate to formalize some related arguments. We loosely follow a generic approach for verifying resource bounds of functional data structures and algorithms in Isabelle [5].

During the formalization, we also present some informal arguments in order to give a better intution of what's going on in the formal proofs.

The present formalization was developed during a practical course on specification and verification at the TUM Chair of Logic and Verification.

theory *Preliminary-Lemmas*
imports *Berlekamp-Zassenhaus.Finite-Field*
HOL−Number-Theory.Number-Theory

begin

2 Preliminary Lemmas

2.1 A little bit of Modular Arithmetic

An obvious lemma. Just for simplification.

lemma *two-powrs-div*:
  assumes $j < (i::nat)$
  shows $((2\hat{\ }i)\ div\ ((2::nat)\hat{\ }(Suc\ j)))*2 = ((2\hat{\ }i)\ div\ (2\hat{\ }j))$
⟨*proof*⟩

lemma *two-powr-div*:
  assumes $j < (i::nat)$
  shows $((2\hat{\ }i)\ div\ ((2::nat)\hat{\ }j)) = 2\hat{\ }(i-j)$
  ⟨*proof*⟩

The order of an element is the same whether we consider it as an integer or as a natural number.

lemma *ord-int*: $ord\ (int\ p)\ (int\ x) = ord\ p\ x$
⟨*proof*⟩

lemma *not-residue-primroot-1*:
  assumes $n > 2$
  shows $\neg residue\text{-}primroot\ n\ 1$
  ⟨*proof*⟩

lemma *residue-primroot-not-cong-1*:
  assumes $residue\text{-}primroot\ n\ g\ n > 2$
  shows $[g \neq 1]\ (mod\ n)$
  ⟨*proof*⟩

We want to show the existence of a generating element of $\mathbb{Z}_p$ where p is prime.

Non-trivial order of an element g modulo p in a ring implies $g \neq 1$. Although this lemma applies to all rings, it's only intended to be used in connection with *nat*s or *int*s

lemma *prime-not-2-order-not-1*:
  assumes $prime\ p$
    $p > 2$
    $ord\ p\ g > 2$
  shows $g \neq 1$
⟨*proof*⟩

The same for modular arithmetic.

lemma *prime-not-2-order-not-1-mod*:
 assumes $prime\ p$
    $p > 2$
    $ord\ p\ g > 2$

shows $[g \neq 1]$ $(mod\ p)$
⟨*proof*⟩

Now we formulate our lemma about generating elements in residue classes: There is an element $g \in \mathbb{Z}_p$ such that for any $x \in \mathbb{Z}_p$ there is a natural i such that $g^i \equiv x \pmod p$.

lemma *generator-exists*:
assumes *prime* $(p::nat)$ $p > 2$
shows $\exists\ g.\ [g \neq 1]\ (mod\ p) \wedge (\forall\ x.\ (0<x \wedge x < p\ ) \longrightarrow (\exists\ i.\ [g\hat{}i = x]\ (mod\ p)))$
⟨*proof*⟩

2.2 General Lemmas in a Finite Field

We make certain assumptions: From now on, we will calculate in a finite field which is the ring of integers modulo a prime p. Let n be the length of vectors to be transformed. By Dirichlet's theorem on arithmetic progressions we can assume that there is a natural number k and a prime p with $p = k \cdot n + 1$. In order to avoid some special cases and even contradictions, we additionally assume that $p \geq 3$ and $n \geq 2$.

locale *preliminary* =
fixes
a-type::(*'a*::*prime-card*) *itself*
and *p*::*nat*
and *n*::*nat*
and *k*::*nat*
assumes
p-def: $p = CARD('a)$ **and** *p-lst3*: $p > 2$ **and** *p-fact*: $p = k*n + 1$
and *n-lst2*: $n \geq 2$
begin

lemma *exp-rule*: $((c::('a)\ mod\text{-}ring) * d\ )\hat{}e = (c\hat{}e) * (d\hat{}e)$
⟨*proof*⟩

lemma $\exists\ y.\ x \neq 0 \longrightarrow (x::(('a)\ mod\text{-}ring)) * \ y = 1$
⟨*proof*⟩

lemma *test*: *prime p*
⟨*proof*⟩

lemma *k-bound*: $k > 0$
⟨*proof*⟩

We show some homomorphisms.

lemma *homomorphism-add*: (*of-int-mod-ring x*)+(*of-int-mod-ring y*) =
((*of-int-mod-ring* (*x*+*y*)) ::(('*a*::*prime-card*) *mod-ring*))
⟨*proof*⟩

lemma *homomorphism-mul-on-ring*: (*of-int-mod-ring x*)∗(*of-int-mod-ring y*) =
((*of-int-mod-ring* (*x*∗*y*)) ::(('*a*::*prime-card*) *mod-ring*))
⟨*proof*⟩

lemma *exp-homo*:(*of-int-mod-ring* $(x \hat{} i)$) = ((*of-int-mod-ring* x)$\hat{}i$ ::(('*a*::*prime-card*) *mod-ring*))
⟨*proof*⟩

lemma *mod-homo*: ((*of-int-mod-ring* x)::(('*a*::*prime-card*) *mod-ring*)) = *of-int-mod-ring* (x *mod* p)
⟨*proof*⟩

lemma *int-exp-hom*: *int* x $\hat{}i$ = *int* $(x\hat{}i)$
⟨*proof*⟩

lemma *coprime-nat-int*: *coprime* (*int* p) (*to-int-mod-ring pr*) $\longleftrightarrow$ *coprime* p (*nat*(*to-int-mod-ring pr*))
⟨*proof*⟩

lemma *nat-int-mod*:[*nat* (*to-int-mod-ring pr*) $\hat{}$ d = *1*] (*mod* p) =
[(*to-int-mod-ring pr*) $\hat{}$ d = *1*] (*mod* (*int* p))
⟨*proof*⟩

Order of p doesn't change when interpreting it as an integer.

lemma *ord-lift*: *ord* (*int* p) (*to-int-mod-ring pr*) = *ord* p (*nat* (*to-int-mod-ring pr*))
⟨*proof*⟩

A primitive root has order $p-1$.

lemma *primroot-ord*: *residue-primroot* p g $\Longrightarrow$ *ord* p g = p −*1*
⟨*proof*⟩

If $x^l = 1$ in $\mathbb{Z}_p$, then l is an upper bound for the order of x in $\mathbb{Z}_p$.

lemma *ord-max*:
assumes $l \neq 0$ (x :: (('*a*::*prime-card*) *mod-ring*))$\hat{}l$ = *1*
shows *ord* p (*to-int-mod-ring* x) $\leq l$
⟨*proof*⟩

2.3 Existence of n-th Roots of Unity in the Finite Field

We obtain an element in the finite field such that its reinterpretation as a *nat* will be a primitive root in the residue class modulo p. The difference between residue classes, their representatives in the Integers and elements of the finite field is notable. When conducting informal proofs, this distinction is usually blurred, but Isabelle enforces the explicit conversion between those structures.

lemma *primroot-ex*:
obtains *primroot*::('*a*::*prime-card*) *mod-ring* **where**
primroot$\hat{}(p-1)$ = *1*
primroot $\neq$ *1*
residue-primroot p (*nat* (*to-int-mod-ring primroot*))
⟨*proof*⟩

From this, we obtain an n-th root of unity ω in the finite field of characteristic p. Note that in this step we will use the assumption $p = k \cdot n + 1$ from locale *preliminary*: The k-th power of a primitive root *pr* modulo p will have the property $(pr^k)^n \equiv 1 \mod p$.

lemma *omega-properties-ex*:
obtains ω ::(('a::prime-card) mod-ring)
where ω^n = 1
$\omega \neq 1$
$\forall$ m. ω^m = 1 $\wedge$ m$\neq$0 $\longrightarrow$ m $\geq$ n
⟨proof⟩

We define an n-th root of unity ω for NTT.

theorem *omega-exists*: $\exists$ ω ::(('a::prime-card) mod-ring) .
ω^n = 1 $\wedge$ $\omega \neq 1$ $\wedge$ ($\forall$ m. ω^m = 1 $\wedge$ m$\neq$0 $\longrightarrow$ m $\geq$ n)
⟨proof⟩

definition (omega::(('a::prime-card) mod-ring)) =
(SOME ω . (ω^n = 1 $\wedge$ $\omega \neq$ 1$\wedge$ ($\forall$ m. ω^m = 1 $\wedge$ m$\neq$0 $\longrightarrow$ m $\geq$ n)))

lemma *omega-properties*: omega^n = 1 omega $\neq$ 1
($\forall$ m. omega^m = 1 $\wedge$ m$\neq$0 $\longrightarrow$ m $\geq$ n)
⟨proof⟩

We define the multiplicative inverse μ of ω.

definition mu = omega ^(n − 1)

lemma *mu-properties*: mu * omega = 1 mu $\neq$ 1
⟨proof⟩

2.4 Some Lemmas on Sums

The following lemmas concern sums over a finite field. Most of the propositions are intuitive.

lemma *sum-in*: ($\sum$ i=0..<(x::nat). f i * (y ::('a mod-ring))) = ($\sum$ i=0..<x. f i)* (y)
⟨proof⟩

lemma *sum-eq*: ($\bigwedge$ i. i < x $\Longrightarrow$ f i = g i)
$\Longrightarrow$ ($\sum$ i=0..<(x::nat). f i) = ($\sum$ i=0..<x. g i)
⟨proof⟩

lemma *sum-diff-in*: ($\sum$ i=0..<(x::nat). (f i)::('a mod-ring)) − ($\sum$ i=0..<x. g i) =
($\sum$ i=0..<x. f i − g i)
⟨proof⟩

lemma *sum-swap*: ($\sum$ i=0..<(x::nat). $\sum$ j=0..<(y::nat). f i j) =
($\sum$ j=0..<(y::nat). $\sum$ i=0..<(x::nat). f i j)
⟨proof⟩

lemma *sum-const*: ($\sum$ i=0..<(x::nat). (c::('a::prime-card) mod-ring)) = (of-int-mod-ring x) * c
⟨proof⟩

lemma *sum-split*: (r1::nat) < r2 $\Longrightarrow$ ($\sum$ l = 0..<r1. ((f l)::(('a::prime-card) mod-ring)))
+($\sum$ l = r1..<r2. f l) = ($\sum$ l = 0..<r2. f l)

$\langle proof \rangle$

lemma *sum-index-shift*: $(\sum l = (a{::}nat)..< b.\ f(l+c)) = (\sum l = (a+c)..< (b+c).\ f\ l\ )$
$\langle proof \rangle$

One may sum over even and odd indices independently. The lemma statement was taken from a formalization of FFT [6]. We give an alternative proof adapted to the finite field $\mathbb{Z}_p$.

lemma *sum-splice*:
$(\sum i{::}nat = 0..<2*nn.\ f\ i) = (\sum i = 0..<nn.\ f\ (2*i)) + (\sum i = 0..<nn.\ f\ (2*i+1))$
$\langle proof \rangle$

lemma *sum-even-odd-split*: $even\ (a{::}nat) \Longrightarrow (\sum j{=}0..<(a\ div\ 2).\ f\ (2*j)) + (\sum j{=}0..<(a\ div\ 2).\ f\ (2*j+1)) = (\sum j{=}0..<a.\ f\ j)$
$\langle proof \rangle$

lemma *sum-splice-other-way-round*: $(\sum j{=}(0{::}nat)..<i.\ f\ (2*j)) + (\sum j{=}0..<i.\ f\ (2*j+1)) = (\sum j{=}(0{::}nat)..<2*i.\ f\ j\ )$
$\langle proof \rangle$

lemma *sum-neg-in*: $-\ (\sum j = 0..<l.\ (f\ j){::}('a\ mod\text{-}ring)) = (\sum j = 0..<l.\ -\ f\ j)$
$\langle proof \rangle$

2.5 Geometric Sums

This lemma will be important for proving properties on NTT. At first, an informal proof sketch:

$$\begin{aligned}(1-x)\cdot\sum_{l=0}^{r-1}x^l &= \sum_{l=0}^{r-1}x^l - x\cdot\sum_{l=0}^{r-1}x^l\\ &= \sum_{l=0}^{r-1}x^l - \sum_{l=1}^{r}x^l\\ &= 1-x^r\end{aligned}$$

The same lemma for integers can be found in [7]. Our version is adapted to finite fields.

lemma *geo-sum*:
assumes $x \neq 1$
shows $(1-x)*(\sum l = 0..<r.\ (x{::}('a\ mod\text{-}ring))\hat{\ }l) = (1-x\hat{\ }r)$
$\langle proof \rangle$

lemmas *sum-rules = sum-in sum-eq sum-diff-in sum-swap sum-const sum-split sum-index-shift*

end
end

theory *NTT*

imports *Preliminary-Lemmas*
begin

3 Number Theoretic Transform and Inverse Transform

locale *ntt = preliminary TYPE ('a ::prime-card) +*
fixes ω :: *('a::prime-card mod-ring)*
fixes μ :: *('a mod-ring)*
assumes *omega-properties*: $\omega\widehat{\ }n = 1$ $\omega \neq 1$ $(\forall\ m.\ \omega\widehat{\ }m = 1 \wedge m{\neq}0 \longrightarrow m \geq n)$
assumes *mu-properties*: $\mu * \omega = 1$
begin

lemma *mu-properties'*: $\mu \neq 1$
⟨*proof*⟩

3.1 Definition of *NTT* and *INTT*

Now we can state an analogue to the *DFT* on finite fields, namely the *Number Theoretic Transform*. First, let us look at an informal definition of NTT [4]:

$$\mathsf{NTT}(\vec{x}) = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2\cdot(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2\cdot(n-1)} & \omega^{3\cdot(n-1)} & \cdots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \cdot \vec{x}$$

Or for single vector entries:

$$\mathsf{NTT}(\vec{x})_i = \sum_{j=0}^{n-1} x_j \cdot \omega^{i \cdot j}$$

Formally:

definition *ntt::(('a ::prime-card) mod-ring) list ⇒ nat ⇒ 'a mod-ring* **where**
ntt numbers i = $(\sum j{=}0..{<}n.\ (numbers\ !\ j) * \omega\widehat{\ }(i{*}j))$

definition *NTT numbers = map (ntt numbers) [0..<n]*

We define the inverse transform INTT by matrices:

$$\mathsf{INTT}(\vec{y}) = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \mu & \mu^2 & \mu^3 & \cdots & \mu^{n-1} \\ 1 & \mu^2 & \mu^4 & \mu^6 & \cdots & \mu^{2\cdot(n-1)} \\ 1 & \mu^3 & \mu^6 & \mu^9 & \cdots & \mu^{3\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \mu^{n-1} & \mu^{2\cdot(n-1)} & \mu^{3\cdot(n-1)} & \cdots & \mu^{(n-1)\cdot(n-1)} \end{pmatrix} \cdot \vec{y}$$

Per component:

$$\mathsf{INTT}(\vec{y})_i = \sum_{j=0}^{n-1} y_j \cdot \mu^{i \cdot j}$$

definition *intt xs i* = (∑ *j*=*0*..<*n*. (*xs* ! *j*) ∗ *μ* ^(*i*∗*j*))

definition *INTT xs* = *map* (*intt xs*) [*0*..<*n*]

Vector length is preserved.

lemma *length-NTT*:
assumes *n-def*: *length numbers* = *n*
shows *length* (*NTT numbers*) = *n*
⟨*proof*⟩

lemma *length-INTT*:
assumes *n-def*: *length numbers* = *n*
shows *length* (*INTT numbers*) = *n*
⟨*proof*⟩

3.2 Correctness Proof of *NTT* and *INTT*

We prove NTT and INTT correct: By taking $\mathsf{INTT}(\mathsf{NTT}(x))$ we obtain x scaled by n. Analogue to DFT, one can get rid of the factor n by a simple rescaling. First, consider an informal proof sketch using the matrix form:

$$\mathsf{INTT}(\mathsf{NTT}(\vec{x})) =$$

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mu & \mu^2 & \cdots & \mu^{n-1} \\ 1 & \mu^2 & \mu^4 & \cdots & \mu^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \mu^{n-1} & \mu^{2\cdot(n-1)} & \cdots & \mu^{(n-1)\cdot(n-1)} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2\cdot(n-1)} & \cdots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \cdot \vec{x}$$

A resulting entry is of the following form:

$$\mathsf{INTT}(\mathsf{NTT}(x))_i = \sum_{j=0}^{n-1} (\sum_{k=0}^{n-1} \mu^{i \cdot k} \cdot \omega^{j \cdot k}) \cdot x_j$$

Now, we analyze the interior sum by cases on $i = j$.

Case $i = j$.

$$\begin{aligned}\sum_{k=0}^{n-1} \mu^{i \cdot k} \cdot \omega^{j \cdot k} &= \sum_{k=0}^{n-1} (\mu \cdot \omega)^{i \cdot k} \\ &= n \cdot (\mu \cdot \omega)^{i \cdot k} \\ &= n \cdot 1^{i \cdot k} \\ &= n\end{aligned}$$

Note that ω and μ are mutually inverse.

Case $i \neq j$. Wlog assume $i > j$, otherwise replace ω by μ and $i - j$ by $j - i$ respectively.

$$\begin{aligned}\sum_{k=0}^{n-1} \mu^{i \cdot k} \cdot \omega^{j \cdot k} &= \sum_{k=0}^{n-1} (\mu \cdot \omega)^{j \cdot k} \cdot \omega^{(i-j) \cdot k} \\ &= \sum_{k=0}^{n-1} \omega^{(i-j) \cdot k} \\ &= (1 - \omega^{(i-j) \cdot n}) \cdot (1 - \omega^{i-j})^{-1} && \text{by lemma on geometric sum} \\ &= (1 - 1^n) \cdot (1 - \omega^{i-j})^{-1} \\ &= 0\end{aligned}$$

We conclude that $\sum_{j=0}^{n-1} (\sum_{k=0}^{n-1} \mu^{i \cdot k} \cdot \omega^{j \cdot k}) \cdot x_j = n \cdot x_i$.

theorem *ntt-correct*:
  assumes *n-def*: *length numbers* = *n*
  shows *INTT* (*NTT numbers*) = *map* (λ *x*. (*of-int-mod-ring n*) $*$ *x*) *numbers*
⟨*proof*⟩

Now we prove the converse to be true: $\mathsf{NTT}(\mathsf{INTT}(\vec{x})) = n \cdot \vec{x}$. The proof proceeds analogously with exchanged roles of ω and μ.

theorem *inv-ntt-correct*:
  assumes *n-def*: *length numbers* = *n*
  shows *NTT* (*INTT numbers*) = *map* (λ *x*. (*of-int-mod-ring n*) $*$ *x*) *numbers*
⟨*proof*⟩

end
end

theory *Butterfly*
  imports *NTT HOL−Library.Discrete-Functions*
begin

4 Butterfly Algorithms

Several recursive algorithms for FFT based on the divide and conquer principle have been developed in order to speed up the transform. A method for reducing complexity is the butterfly scheme. In this formalization, we consider the butterfly algorithm by Cooley and Tukey [1] adapted to the setting of NTT.

We additionally assume that n is power of two.

locale *butterfly* = *ntt* +
  fixes N
  assumes *n-two-pot*: $n = 2\hat{~}N$
begin

4.1 Recursive Definition

Let's recall the definition of a transformed vector element:

$$\mathsf{NTT}(\vec{x})_i = \sum_{j=0}^{n-1} x_j \cdot \omega^{i \cdot j}$$

We assume $n = 2^N$ and obtain:

$$\begin{aligned}
&\sum_{j=0}^{<2^N} x_j \cdot \omega^{i \cdot j} \\
&\qquad = \sum_{j=0}^{<2^{N-1}} x_{2j} \cdot \omega^{i \cdot 2j} + \sum_{j=0}^{<2^{N-1}} x_{2j+1} \cdot \omega^{i \cdot (2j+1)} \\
&\qquad = \sum_{j=0}^{<2^{N-1}} x_{2j} \cdot (\omega^2)^{i \cdot j} + \omega^i \cdot \sum_{j=0}^{<2^{N-1}} x_{2j+1} \cdot (\omega^2)^{i \cdot j} \\
&\qquad = (\sum_{j=0}^{<2^{N-2}} x_{4j} \cdot (\omega^4)^{i \cdot j} + \omega^i \cdot \sum_{j=0}^{<2^{N-2}} x_{4j+2} \cdot (\omega^4)^{i \cdot j}) \\
&\qquad\qquad + \omega^i \cdot (\sum_{j=0}^{<2^{N-2}} x_{4j+1} \cdot (\omega^4)^{i \cdot j} + \omega^i \cdot \sum_{j=0}^{<2^{N-2}} x_{4j+3} \cdot (\omega^4)^{i \cdot j}) \text{ etc.}
\end{aligned}$$

which gives us a recursive algorithm:

- Compose vectors consisting of elements at even and odd indices respectively
- Compute a transformation of these vectors recursively where the dimensions are halved.
- Add results after scaling the second subresult by ω^i

Now we give a functional definition of the analogue to FFT adapted to finite fields. A gentle introduction to FFT can be found in [2]. For the fast implementation of Number Theoretic Transform in particular, have a look at [3].

(The following lemma is needed to obtain an automated termination proof of $FNTT$.)

lemma *FNTT-termination-aux* [*simp*]: *length* (*filter P* [*0..<l*]) < *Suc l*
⟨*proof*⟩

Please note that we closely adhere to the textbook definition which just talks about elements at even and odd indices. We model the informal definition by predefined functions, since this seems to be more handy during proofs. An algorithm splitting the elements smartly will be presented afterwards.

fun *FNTT*::('*a mod-ring*) *list* ⇒ ('*a mod-ring*) *list* **where**
FNTT [] = []|
FNTT [*a*] = [*a*]|
FNTT nums = (*let nn* = *length nums*;
nums1 = [*nums*!*i*. *i* ← *filter even* [*0..<nn*]];
nums2 = [*nums*!*i*. *i* ← *filter odd* [*0..<nn*]];
fntt1 = *FNTT nums1*;
fntt2 = *FNTT nums2*;
sum1 = *map2* (+) *fntt1* (*map2* (λ *x k*. *x*∗(ω ^ ((*n div nn*) ∗ *k*))) *fntt2* [*0..<*(*nn div 2*)]);
sum2 = *map2* (−) *fntt1* (*map2* (λ *x k*. *x*∗(ω ^ ((*n div nn*) ∗ *k*))) *fntt2* [*0..<*(*nn div 2*)])
in sum1@*sum2*)

lemmas [*simp del*] = *FNTT-termination-aux*

Finally, we want to prove correctness, i.e. $FNTT\ xs = NTT\ xs$. Since we consider a recursive algorithm, some kind of induction is appropriate: Assume the claim for $\frac{2^d}{2} = 2^{d-1}$ and prove it for 2^d, where 2^d is the vector length. This implies that we have to talk about NTTs with respect to some powers of ω. In particular, we decide to annotate NTT with a degree *degr* indicating the referred vector length. There is a correspondence to the current level l of recursion:

$$degr = 2^{N-l}$$

A generalized version of NTT keeps track of all levels during recursion:

definition *ntt-gen numbers degr i* = (∑ *j*=*0..<*(*length numbers*). (*numbers* ! *j*) ∗ ω ^ ((*n div degr*)∗*i*∗*j*))

definition *NTT-gen degr numbers* = *map* (*ntt-gen numbers* (*degr*)) [*0..< length numbers*]

Whenever generalized NTT is applied to a list of full length, then its actually equal to the defined NTT.

lemma *NTT-gen-NTT-full-length*:
assumes *length numbers* =*n*
shows *NTT-gen n numbers* = *NTT numbers*
⟨*proof*⟩

4.2 Arguments on Correctness

First some general lemmas on list operations.

lemma *length-even-filter*: $length\ [f\ i\ .\ \ i <- (filter\ even\ [0..<l])] = l-l\ div\ 2$
⟨*proof*⟩

lemma *length-odd-filter*: $length\ [f\ i\ .\ \ i <- (filter\ odd\ [0..<l])] = l\ div\ 2$
⟨*proof*⟩

lemma *map2-length*: $length\ (map2\ f\ xs\ ys) = \ min\ (length\ xs)\ (length\ ys)$
⟨*proof*⟩

lemma *map2-index*: $i < length\ xs \Longrightarrow i < length\ ys \Longrightarrow (map2\ f\ xs\ ys)\ !\ i = f\ (xs\ !\ i)\ (ys\ !\ i)$
⟨*proof*⟩

lemma *filter-last-not*: $\neg\ P\ x \Longrightarrow filter\ P\ (xs@[x]) = filter\ P\ xs$
⟨*proof*⟩

lemma *filter-even-map*: $filter\ even\ [0..<2*(x::nat)] = map\ ((*)\ (2::nat))\ [0..<x]$
⟨*proof*⟩

lemma *filter-even-nth*: $2*j < l \Longrightarrow 2*x = l \Longrightarrow (filter\ even\ [0..<l]\ !\ j) = (2*j)$
⟨*proof*⟩

lemma *filter-odd-map*: $filter\ odd\ [0..<2*(x::nat)] = map\ (\lambda\ y.\ (2::nat)*y\ +1)\ [0..<x]$
⟨*proof*⟩

lemma *filter-odd-nth*: $2*j < l \Longrightarrow 2*x = l \Longrightarrow (filter\ odd\ [0..<l]\ !\ j) = (2*j+1)$
⟨*proof*⟩

Lemmas by using the assumption $n = 2^N$.

(-1 denotes the additive inverse of 1 in the finite field.)

lemma *n-min1-2*: $n = 2 \ \Longrightarrow \omega \ = -1$
⟨*proof*⟩

lemma *n-min1-gr2*:
assumes $n > 2$
shows $\omega\hat{}(n\ div\ 2) = -1$
⟨*proof*⟩

lemma *div-exp-sub*: $2\hat{}l \ < n \Longrightarrow n\ div\ (2\hat{}l) = 2\hat{}(N-l)$⟨*proof*⟩

lemma *omega-div-exp-min1*:
assumes $2\hat{}(Suc\ l) \leq n$
shows $(\omega\ \hat{}(n\ div\ 2\hat{}(Suc\ l)))\hat{}(2\hat{}l) = -1$
⟨*proof*⟩

lemma *omg-n-2-min1*: $\omega\hat{}(n\ div\ 2) = -1$
⟨*proof*⟩

lemma *neg-cong*: $-(x::('a\ mod\text{-}ring)) = -\ y \Longrightarrow x = y$ ⟨*proof*⟩

Generalized *NTT* indeed describes all recursive levels, and thus, it is actually equivalent to the ordinary *NTT* definition.

theorem *FNTT-NTT-gen-eq*: *length numbers* = *2^l* ⟹ *2^l* ≤ *n* ⟹ *FNTT numbers* = *NTT-gen* (*length numbers*) *numbers*
⟨*proof*⟩

Major Correctness Theorem for Butterfly Algorithm.

We have already shown:

- Generalized *NTT* with degree annotation 2^N equals usual *NTT*.
- Generalized *NTT* tracks all levels of recursion in *FNTT*.

Thus, *FNTT* equals *NTT*.

theorem *FNTT-correct*:
  assumes *length numbers* = *n*
  shows *FNTT numbers* = *NTT numbers*
  ⟨*proof*⟩

4.3 Inverse Transform in Butterfly Scheme

We also formalized the inverse transform by using the butterfly scheme. Proofs are obtained by adaption of arguments for *FNTT*.

lemmas [*simp*] = *FNTT-termination-aux*

```
fun IFNTT where
IFNTT [] = []|
IFNTT [a] = [a]|
IFNTT nums = (let nn = length nums;
                 nums1 = [nums!i .  i <− (filter even [0..<nn])];
                 nums2 = [nums!i .  i <− (filter odd [0..<nn])];
                 ifntt1 = IFNTT nums1;
                 ifntt2 = IFNTT nums2;
                 sum1 = map2 (+) ifntt1 (map2 ( λ x k.  x∗(μ^( (n div nn) ∗ k))) ifntt2 [0..<(nn div
2)]);
                 sum2 = map2 (−) ifntt1 (map2 ( λ x k.  x∗(μ^( (n div nn) ∗ k))) ifntt2 [0..<(nn div
2)])
                  in sum1@sum2)
```

lemmas [*simp del*] = *FNTT-termination-aux*

definition *intt-gen numbers degr i* = ($\sum$ *j*=*0*..<(*length numbers*). (*numbers* ! *j*) ∗ μ ^((*n div degr*)∗*i*∗*j*))

definition *INTT-gen degr numbers* = *map* (*intt-gen numbers* (*degr*)) [*0*..< *length numbers*]

lemma *INTT-gen-INTT-full-length*:
  assumes *length numbers* =*n*
  shows *INTT-gen n numbers* = *INTT numbers*
  ⟨*proof*⟩

lemma *my-div-exp-min1*:
  assumes *2*^(*Suc l*) ≤ *n*
  shows (*μ* ^(*n div 2*^(*Suc l*)))^(*2*^*l*) = −*1*
  ⟨*proof*⟩

lemma *my-n-2-min1*: *μ*^(*n div 2*) = −*1*
  ⟨*proof*⟩

Correctness proof by common induction technique. Same strategies as for $FNTT$.

theorem *IFNTT-INTT-gen-eq*:
length numbers = *2*^*l* ⟹ *2*^*l* ≤ *n* ⟹ *IFNTT numbers* = *INTT-gen* (*length numbers*) *numbers*
⟨*proof*⟩

Correctness of the butterfly scheme for the inverse *INTT*.

theorem *IFNTT-correct*:
  assumes *length numbers* = *n*
  shows *IFNTT numbers* = *INTT numbers*
  ⟨*proof*⟩

Also $FNTT$ and $IFNTT$ are mutually inverse

theorem *IFNTT-inv-FNTT*:
  assumes *length numbers* = *n*
  shows *IFNTT* (*FNTT numbers*) = *map* ((∗) (*of-int-mod-ring* (*int n*))) *numbers*
  ⟨*proof*⟩

The other way round:

theorem *FNTT-inv-IFNTT*:
  assumes *length numbers* = *n*
  shows *FNTT* (*IFNTT numbers*) = *map* ((∗) (*of-int-mod-ring* (*int n*))) *numbers*
⟨*proof*⟩

4.4 An Optimization

Currently, we extract elements on even and odd positions respectively by a list comprehension over even and odd indices. Due to the definition in Isabelle, an index access has linear time complexity. This results in quadratic running time complexity for every level in the recursion tree of the *FNTT*. In order to reach the $\mathcal{O}(n \log n)$ time bound, we have find a better way of splitting the elements at even or odd indices respectively.

A core of this optimization is the *evens-odds* function, which splits the vectors in linear time.

fun *evens-odds*::*bool* ⇒'*b list* ⇒ '*b list* **where**

```
evens-odds - [] = []|
evens-odds True (x#xs)= (x# evens-odds False xs)|
evens-odds False (x#xs) = evens-odds True xs
```

```
lemma map-filter-shift: map f (filter even [0..<Suc g]) =
      f 0 # map (λ x. f (x+1)) (filter odd [0..<g])
  ⟨proof⟩
```

```
lemma map-filter-shift': map f (filter odd [0..<Suc g]) =
        map (λ x. f (x+1)) (filter even [0..<g])
  ⟨proof⟩
```

A splitting by the *evens-odds* function is equivalent to the more textbook-like list comprehension.

```
lemma filter-compehension-evens-odds:
     [xs ! i. i <− filter even [0..<length xs]] = evens-odds True xs ∧
      [xs ! i. i <− filter odd [0..<length xs]] = evens-odds False xs
  ⟨proof⟩
```

For automated termination proof.

```
lemma [simp]: length (evens-odds True vc) < Suc (length vc)
              length (evens-odds False vc) < Suc (length vc)
  ⟨proof⟩
```

The $FNTT$ definition from above was suitable for matters of proof conduction. However, the naive decomposition into elements at odd and even indices induces a complexity of n^2 in every recursive step. As mentioned, the *evens-odds* function filters for elements on even or odd positions respectively. The list has to be traversed only once which gives *linear* complexity for every recursive step.

```
fun FNTT' where
FNTT' [] = []|
FNTT' [a] = [a]|
FNTT' nums = (let nn = length nums;
                nums1 = evens-odds  True nums;
                nums2 = evens-odds False nums;
                fntt1 = FNTT' nums1;
                fntt2 = FNTT' nums2;
                fntt2-omg =  (map2 ( λ x k.  x∗(ω^( (n div nn) ∗ k))) fntt2 [0..<(nn div 2)]);
                sum1 = map2 (+) fntt1 fntt2-omg;
                sum2 = map2 (−) fntt1 fntt2-omg
                 in sum1@sum2)
```

The optimized $FNTT$ is equivalent to the naive NTT.

```
lemma FNTT'-FNTT: FNTT' xs = FNTT xs
  ⟨proof⟩
```

It is quite surprising that some inaccuracies in the interpretation of informal textbook definitions - even when just considering such a simple algorithm - can indeed affect time complexity.

4.5 Arguments on Running Time

FFT is especially known for its $\mathcal{O}(n \log n)$ running time. Unfortunately, Isabelle does not provide a built-in time formalization. Nonetheless we can reason about running time after defining some "reasonable" consumption functions by hand. Our approach loosely follows a general pattern by Nipkow et al. [5]. First, we give running times and lemmas for the auxiliary functions used during FNTT.
General ideas behind the $\mathcal{O}(n \log n)$ are:

- By recursively halving the problem size, we obtain a tree of depth $\mathcal{O}(\log n)$.
- For every level of that tree, we have to process all elements which gives $\mathcal{O}(n)$ time.

Time for splitting the list according to even and odd indices.

fun $T\text{-}_{eo}$::*bool* ⇒ *'c list* ⇒ *nat* **where**
$T\text{-}_{eo}$ - [] = *1*|
$T\text{-}_{eo}$ *True* (*x#xs*)= (*1*+ $T\text{-}_{eo}$ *False xs*)|
$T\text{-}_{eo}$ *False* (*x#xs*) = (*1*+ $T\text{-}_{eo}$ *True xs*)

lemma *T-eo-linear*: $T\text{-}_{eo}$ *b xs* = *length xs* + *1*
⟨*proof*⟩

Time for length.

fun T_{length} **where**
T_{length} [] = *1* |
T_{length} (*x#xs*) = *1*+ T_{length} *xs*

lemma *T-length-linear*: T_{length} *xs* = *length xs* +*1*
⟨*proof*⟩

Time for index access.

fun T_{nth} **where**
T_{nth} [] *i* = *1* |
T_{nth} (*x#xs*) *0* = *1*|
T_{nth} (*x#xs*) (*Suc i*) = *1* + T_{nth} *xs i*

lemma *T-nth-linear*: T_{nth} *xs i* ≤ *length xs* +*1*
⟨*proof*⟩

Time for mapping two lists into one result.

fun T_{map2} **where**
T_{map2} *t* [] - = *1*|
T_{map2} *t* - [] = *1*|
T_{map2} *t* (*x#xs*) (*y#ys*) = (*t x y* + *1* + T_{map2} *t xs ys*)

lemma *T-map-2-linear*:
c > *0* ⟹
(⋀ *x y*. *t x y* ≤ *c*) ⟹ T_{map2} *t xs ys* ≤ *min* (*length xs*) (*length ys*) ∗ (*c*+*1*) + *1*

⟨*proof*⟩

lemma *T-map-2-linear′*:
$c > 0 \Longrightarrow$
$(\bigwedge x\ y.\ t\ x\ y = c) \Longrightarrow T_{map2}\ t\ xs\ ys = min\ (length\ xs)\ (length\ ys)\ *\ (c+1) + 1$
⟨*proof*⟩

Time for append.

fun T_{app} **where**
T_{app} [] - = *1*|
T_{app} (*x*#*xs*) *ys* = *1* + T_{app} *xs ys*

lemma *T-app-linear*: T_{app} *xs ys* = *length xs* +*1*
⟨*proof*⟩

Running Time of (optimized) $FNTT$.

fun T_{FNTT}::(′*a mod-ring*) *list* ⇒ *nat* **where**
T_{FNTT} [] = *1*|
T_{FNTT} [*a*] = *1*|
T_{FNTT} *nums* = (*1* +T_{length} *nums*+ *3*+

(*let nn* = *length nums*;
nums1 = *evens-odds True nums*;
nums2 = *evens-odds False nums*
in
T-$_{eo}$ *True nums* + *T*-$_{eo}$ *False nums* + *2* +
(*let*
fntt1 = *FNTT nums1*;
fntt2 = *FNTT nums2*
in
(T_{FNTT} *nums1*) + (T_{FNTT} *nums2*) +
(*let*
sum1 = *map2* (+) *fntt1* (*map2* (λ *x k*. *x*∗(ω ^((*n div nn*) ∗ *k*))) *fntt2* [*0*..<(*nn div 2*)]);
sum2 = *map2* (−) *fntt1* (*map2* (λ *x k*. *x*∗(ω ^((*n div nn*) ∗ *k*))) *fntt2* [*0*..<(*nn div 2*)])
in
2∗ T_{map2} (λ *x y*. *1*) *fntt2* [*0*..<(*nn div 2*)] +
2∗ T_{map2} (λ *x y*. *1*) *fntt1* (*map2* (λ *x k*. *x*∗(ω ^((*n div nn*) ∗ *k*))) *fntt2* [*0*..<(*nn div 2*)]) +
T_{app} *sum1 sum2*))))

lemma *mono*: ((*f x*)::*nat*) ≤ *f y* ⟹ *f y* ≤ *fz* ⟹ *f x* ≤ *fz* ⟨*proof*⟩

lemma *evens-odds-length*:
length (*evens-odds True xs*) = (*length xs*+*1*) *div 2* ∧
length (*evens-odds False xs*) = (*length xs*) *div 2*
⟨*proof*⟩

Length preservation during $FNTT$.

lemma *FNTT-length*: *length numbers* = *2^l* ⟹ *length* (*FNTT numbers*) = *length numbers*
⟨*proof*⟩

lemma *add-cong*: (*a1*::*nat*) + *a2*+*a3* +*a4*= *b* ⟹ *a1* +*a2*+ *c* + *a3*+*a4*= *c* +*b*
⟨*proof*⟩

lemma *add-mono*:*a* ≤ (*b*::*nat*) ⟹ *c* ≤ *d* ⟹ *a* + *c* ≤ *b* +*d* ⟨*proof*⟩

lemma *xyz*: *Suc* (*Suc* (*length xs*)) = *2* ^ *l* ⟹ *length* (*x* # *evens-odds True xs*) = *2* ^(*l* − *1*)
⟨*proof*⟩

lemma *zyx*: *Suc* (*Suc* (*length xs*)) = *2* ^ *l* ⟹ *length* (*y* # *evens-odds False xs*) = *2* ^(*l* − *1*)
⟨*proof*⟩

When $length\ xs = 2^l$, then $length\ (evens\text{-}odds\ xs) = 2^{l-1}$.

lemma *evens-odds-power-2*:
fixes *x*::′*b* **and** *y*::′*b*
assumes *Suc* (*Suc* (*length* (*xs*::′*b list*))) = *2* ^ *l*
shows *Suc*(*length* (*evens-odds b xs*)) = *2* ^(*l*−*1*)
⟨*proof*⟩

Major Lemma: We rewrite the Running time of $FNTT$ in this proof and collect constraints for the time bound. Using this, bounds are chosen in a way such that the induction goes through properly.

We define:

$$T(2^0) = 1$$

$$T(2^l) = (2^l - 1) \cdot 14apply + 15 \cdot l \cdot 2^{l-1} + 2^l$$

We want to show:

$$T_{FNTT}(2^l) = T(2^l)$$

(Note that by abuse of types, the 2^l denotes a list of length 2^l.)

First, let's informally check that T is indeed an accurate description of the running time:

$$\begin{aligned}
T_{FNTT}(2^l) &= 14 + 15 \cdot 2^{l-1} + 2 \cdot T_{FNTT}(2^{l-1}) \qquad \text{by analyzing the running time function} \\
&\overset{I.H.}{=} 14 + 15 \cdot 2^{l-1} + 2 \cdot ((2^{l-1} - 1) \cdot 14 + (l-1) \cdot 15 \cdot 2^{l-2} + 2^{l-1}) \\
&= 14 \cdot 2^l - 14 + 15 \cdot 2^{l-1} + 15 \cdot l \cdot 2^{l-1} - 15 \cdot 2^{l-1} + 2^l \\
&= (2^l - 1) \cdot 14 + 15 \cdot l \cdot 2^{l-1} + 2^l \\
&\overset{def.}{=} T(2^l)
\end{aligned}$$

The base case is trivially true.

theorem *tight-bound*:
assumes *T-def*: $\bigwedge$ *numbers l. length numbers = 2^l* $\Longrightarrow$ *l > 0* $\Longrightarrow$
T numbers = (2^l − 1) ∗ 14 + l ∗15∗2^(l−1) + 2^l
$\bigwedge$ *numbers l. l =0* $\Longrightarrow$ *length numbers = 2^l* $\Longrightarrow$ *T numbers = 1*
shows *length numbers = 2^l* $\Longrightarrow$ T_{FNTT} *numbers = T numbers*
⟨*proof*⟩

We can finally state that $FNTT$ has $\mathcal{O}(n \log n)$ time complexity.

theorem *log-lin-time*:
assumes *length numbers = 2^l*
shows T_{FNTT} *numbers* ≤ *30 ∗ l ∗ length numbers + 1*
⟨*proof*⟩

theorem *log-lin-time-explicitly*:
assumes *length numbers = 2^l*
shows T_{FNTT} *numbers* ≤ *30 ∗ floor-log (length numbers) ∗ length numbers + 1*
⟨*proof*⟩

end
end

References

[1] I. J. Good. “Introduction to Cooley and Tukey (1965) An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Breakthroughs in Statistics*. Ed. by S. Kotz and N. L. Johnson. New York, NY: Springer New York, 1997, pp. 201–216. ISBN: 978-1-4612-0667-5. DOI: 10.1007/978-1-4612-0667-5_9.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[3] P. Longa and M. Naehrig. *Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography*. Cryptology ePrint Archive, Paper 2016/504. https://eprint.iacr.org/2016/504. 2016.

[4] Nayuki. *Number-theoretic transform (integer DFT)*. https://www.nayuki.io/page/number-theoretic-transform-integer-dft. 2022.

[5] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, Bohua Zhan. *Functional Algorithms, Verified!* https://functional-algorithms-verified.org/. 2021.

[6] C. Ballarin. “Fast Fourier Transform”. In: *Archive of Formal Proofs* (2005). https://isa-afp.org/entries/FFT.html, Formal proof development. ISSN: 2150-914x.

[7] M. Eberl. “Dirichlet Series”. In: *Archive of Formal Proofs* (2017). https://isa-afp.org/entries/Dirichlet_Series.html, Formal proof development. ISSN: 2150-914x.