

Number Theoretic Transform

Thomas Ammer and Katharina Kreuzer

October 27, 2022

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 | 3 |
| 2 | Preliminary Lemmas | 4 |
| 2.1 | A little bit of Modular Arithmetic | 4 |
| 2.2 | General Lemmas in a Finite Field | 6 |
| 2.3 | Existence of n -th Roots of Unity in the Finite Field | 8 |
| 2.4 | Some Lemmas on Sums | 10 |
| 2.5 | Geometric Sums | 12 |
| 3 | Number Theoretic Transform and Inverse Transform | 12 |
| 3.1 | Definition of NTT and $INTT$ | 13 |
| 3.2 | Correctness Proof of NTT and $INTT$ | 14 |
| 4 | Butterfly Algorithms | 22 |
| 4.1 | Recursive Definition | 22 |
| 4.2 | Arguments on Correctness | 24 |
| 4.3 | Inverse Transform in Butterfly Scheme | 32 |
| 4.4 | An Optimization | 39 |
| 4.5 | Arguments on Running Time | 41 |

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*:

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

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:

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

Throughout the paper, we stick to this definition. An inverse transform *INTT* is obtained by replacing ω by its field inverse μ (i.e. $\mu \cdot \omega \equiv 1 \pmod{p}$). 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 Number Theoretic Transform (FNNT)* [3] and *IFNNT* respectively. The corresponding procedure is treated in Section 4. We prove equality between *(I)NTT* and *(I)FNNT* and can infer that both are mutually inverse by previous 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 intuition 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-pows-div:
  assumes  $j < (i::nat)$ 
  shows  $((2^i) \text{ div } ((2::nat)^\wedge(Suc\ j)))*2 = ((2^i) \text{ div } (2^j))$ 
proof-
  have  $((2::nat)^\wedge i) \text{ div } (2^\wedge(Suc\ j)) = 2^\wedge(i-1) \text{ div } (2^j)$  using assms
  by (smt (z3) One-nat-def add-le-cancel-left diff-Suc-Suc div-by-Suc-0 div-if less-nat-zero-code plus-1-eq-Suc power-diff-power-eq zero-neq-numeral)
  thus ?thesis
  by (metis Suc-diff-Suc Suc-leI assms less-imp-le-nat mult.commute power-Suc power-diff-power-eq zero-neq-numeral)
qed
```

```
lemma two-powr-div:
  assumes  $j < (i::nat)$ 
  shows  $((2^i) \text{ div } ((2::nat)^\wedge j)) = 2^\wedge(i-j)$ 
  by (simp add: assms less-or-eq-imp-le power-diff)
```

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

```
lemma ord-int:  $\text{ord } (int\ p) (int\ x) = \text{ord } p\ x$ 
proof (cases coprime p x)
  case False
  thus ?thesis
  by (auto simp: ord-def)
next
  case True
  have (LEAST  $d. 0 < d \wedge [int\ x^\wedge d = 1] \pmod{int\ p}$ ) =  $\text{ord } p\ x$ 
  proof (intro Least-equality conjI)
    show  $[int\ x^\wedge \text{ord } p\ x = 1] \pmod{int\ p}$ 
    using True by (metis cong-int-iff of-nat-1 of-nat-power ord-works)
    show  $\text{ord } p\ x \leq y$  if  $0 < y \wedge [int\ x^\wedge y = 1] \pmod{int\ p}$  for  $y$ 
    using that by (metis cong-int-iff int-ops(2) linorder-not-less of-nat-power ord-minimal)
  qed (use True in auto)
  thus ?thesis
  by (auto simp: ord-def)
qed
```

```
lemma not-residue-primroot-1:
  assumes  $n > 2$ 
  shows  $\neg \text{residue-primroot } n\ 1$ 
  using assms totient-gt-1[of n] by (auto simp: residue-primroot-def)
```

lemma *residue-primroot-not-cong-1*:
assumes *residue-primroot* n g $n > 2$
shows $[g \neq 1] \pmod n$
using *residue-primroot-cong not-residue-primroot-1 assms* **by** *metis*

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 *nats* or *ints*

lemma *prime-not-2-order-not-1*:
assumes *prime* p
 $p > 2$
 $\text{ord } p \ g > 2$
shows $g \neq 1$
proof
assume $g = 1$
hence $\text{ord } p \ g = 1$ **unfolding** *ord-def*
by (*simp add: Least-equality*)
then show *False* **using** *assms* **by** *auto*
qed

The same for modular arithmetic.

lemma *prime-not-2-order-not-1-mod*:
assumes *prime* p
 $p > 2$
 $\text{ord } p \ g > 2$
shows $[g \neq 1] \pmod p$
proof
assume $[g = 1] \pmod p$
hence $\text{ord } p \ g = 1$ **unfolding** *ord-def*
by(*split if-split, metis assms(1) assms(2) assms(3) ord-cong prime-not-2-order-not-1*)
then show *False* **using** *assms* **by** *auto*
qed

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* ($p::\text{nat}$) $p > 2$
shows $\exists g. [g \neq 1] \pmod p \wedge (\forall x. (0 < x \wedge x < p) \longrightarrow (\exists i. [g^i = x] \pmod p))$
proof–
obtain g **where** *g-prim-root:residue-primroot* p g
using *assms prime-gt-1-nat prime-primitive-root-exists*
by (*metis One-nat-def*)
have *g-not-1*: $[g \neq 1] \pmod p$
using *residue-primroot-not-cong-1 assms g-prim-root* **by** *blast*

have $\exists i. [g^i = x] \pmod p$ **if** *x-bounds*: $x > 0 \ x < p$ **for** x
proof –

```

have 1:coprime p x
  using assms prime-nat-iff'' x-bounds by blast
have 2:ord p g = p-1
  by (metis assms(1) g-prim-root residue-primroot-def totient-prime)
hence bij: bij-betw ( $\lambda i. g \hat{=} i \bmod p$ )  $\{..<totient\ p\}$  (totatives p)
  using residue-primroot-is-generator[of p g] totatives-def[of p]
    1 totient-def[of p] assms g-prim-root prime-gt-1-nat by blast
have 3: $x \bmod p \in totatives\ p$ 
  by (simp add: 1 coprime-commute in-totatives-iff order-le-less x-bounds)
have  $\{..<totient\ p\} \neq \{\}$ 
  by (metis assms(1) lessThan-empty-iff prime-nat-iff'' totient-0-iff)
then obtain i where  $g \hat{=} i \bmod p = x \bmod p$ 
  using bij-betw-inv[of ( $\lambda i. g \hat{=} i \bmod p$ )  $\{..<totient\ p\}$  (totatives p)]
    3 bij
  by (metis (no-types, lifting) bij-betw-iff-bijections)
then show ?thesis
  using cong-def by blast
qed
thus ?thesis
  using g-prim-root g-not-1 by auto
qed

```

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:  $((c::('a) \bmod\ ring) * d) \hat{=} e = (c \hat{=} e) * (d \hat{=} e)$ 
  by (simp add: power-mult-distrib)

lemma  $\exists y. x \neq 0 \longrightarrow (x::('a) \bmod\ ring) * y = 1$ 
  by (metis dvd-field-iff unit-dvdE)

lemma test: prime p
  by (simp add: p-def prime-card)

```

lemma *k-bound*: $k > 0$
using *p-fact prime-nat-iff'' test* **by** *force*

We show some homomorphisms.

lemma *homomorphism-add*: $(\text{of-int-mod-ring } x) + (\text{of-int-mod-ring } y) =$
 $((\text{of-int-mod-ring } (x+y)) :: ('a::\text{prime-card mod-ring}))$
by (*metis of-int-hom.hom-add of-int-of-int-mod-ring*)

lemma *homomorphism-mul-on-ring*: $(\text{of-int-mod-ring } x) * (\text{of-int-mod-ring } y) =$
 $((\text{of-int-mod-ring } (x*y)) :: ('a::\text{prime-card mod-ring}))$
by (*metis of-int-mult of-int-of-int-mod-ring*)

lemma *exp-homo*: $(\text{of-int-mod-ring } (x \hat{\ } i)) = ((\text{of-int-mod-ring } x) \hat{\ } i :: ('a::\text{prime-card mod-ring}))$
by (*induction i*) (*metis of-int-of-int-mod-ring of-int-power*) +

lemma *mod-homo*: $((\text{of-int-mod-ring } x) :: ('a::\text{prime-card mod-ring})) = \text{of-int-mod-ring } (x \bmod p)$
using *p-def unfolding of-int-mod-ring-def* **by** *simp*

lemma *int-exp-hom*: $\text{int } x \hat{\ } i = \text{int } (x \hat{\ } i)$
by *simp*

lemma *coprime-nat-int*: $\text{coprime } (\text{int } p) (\text{to-int-mod-ring } pr) \longleftrightarrow \text{coprime } p (\text{nat } (\text{to-int-mod-ring } pr))$
unfolding *coprime-def to-int-mod-ring-def*
by (*smt (z3) Rep-mod-ring atLeastLessThan-iff dvd-trans int-dvd-int-iff int-nat-eq int-ops(2) prime-divisor-exists prime-nat-int-transfer primes-dvd-imp-eq test to-int-mod-ring.rep-eq to-int-mod-ring-def*)

lemma *nat-int-mod*: $[\text{nat } (\text{to-int-mod-ring } pr) \hat{\ } d = 1] (\bmod p) =$
 $[\text{to-int-mod-ring } pr \hat{\ } d = 1] (\bmod (\text{int } p))$
unfolding *to-int-mod-ring-def*
by (*metis Rep-mod-ring atLeastLessThan-iff cong-int-iff int-exp-hom int-nat-eq int-ops(2) to-int-mod-ring.rep-eq to-int-mod-ring-def*)

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

lemma *ord-lift*: $\text{ord } (\text{int } p) (\text{to-int-mod-ring } pr) = \text{ord } p (\text{nat } (\text{to-int-mod-ring } pr))$

proof –

have *to-int-mod-ring pr = int (nat (to-int-mod-ring pr))*
by (*metis Rep-mod-ring atLeastLessThan-iff int-nat-eq to-int-mod-ring.rep-eq*)
thus *?thesis*
using *ord-int* **by** *metis*

qed

A primitive root has order $p - 1$.

lemma *primroot-ord*: $\text{residue-primroot } p \ g \implies \text{ord } p \ g = p - 1$
by (*simp add: residue-primroot-def test totient-prime*)

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 $l \neq 0 \ (x :: ('a::\text{prime-card mod-ring})) \hat{\ } l = 1$

```

shows ord p (to-int-mod-ring x) ≤ l
proof–
  have [(to-int-mod-ring x) ^ l = 1] (mod p)
  by (metis assms(2) cong-def exp-homo of-int-mod-ring.rep-eq of-int-mod-ring-to-int-mod-ring one-mod-card-int
one-mod-ring.rep-eq p-def)
  thus ?thesis unfolding ord-def using assms
  by (smt (z3) Least-le less-imp-le-nat not-gr0)
qed

```

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 ^ (p-1) = 1
  primroot ≠ 1
  residue-primroot p (nat (to-int-mod-ring primroot))
proof–
  obtain g where g-Def: residue-primroot p g ∧ g ≠ 1
  using prime-nat-iff' prime-primitive-root-exists test
  by (metis bigger-prime euler-theorem ord-1-right power-one-right prime-nat-iff'' residue-primroot.cases
residue-primroot-cong)
  hence [g ≠ 1] (mod p) using prime-not-2-order-not-1-mod[of p g]
  by (metis One-nat-def p-lst3 less-numeral-extra(4) ord-eq-Suc-0-iff residue-primroot.cases totient-gt-1)
  hence [g ^ (p-1) = 1] (mod p) using g-Def
  by (metis coprime-commute euler-theorem residue-primroot-def test totient-prime)
  moreover hence int (g ^ (p - 1)) mod int p = (1::int)
  by (metis cong-def int-ops(2) mod-less of-nat-mod prime-gt-1-nat test)
  moreover hence of-int-mod-ring (int (g ^ (p - 1)) mod int p) =
    ((of-int-mod-ring 1) :: ('a::prime-card) mod-ring) by simp
  ultimately have (of-int-mod-ring (g ^ (p-1))) = (1 :: ('a::prime-card) mod-ring)
  using mod-homo[of g ^ (p-1)] by (metis exp-homo power-0)
  hence ((of-int-mod-ring g) ^ (p-1) :: ('a::prime-card) mod-ring) = 1
  using exp-homo[of int g p-1] by simp
  moreover
  have ((of-int-mod-ring g) :: ('a::prime-card) mod-ring) ≠ 1
  proof
  assume ((of-int-mod-ring g) :: ('a::prime-card) mod-ring) = 1
  hence [int g = 1] (mod p) using p-def unfolding of-int-mod-ring-def
  by (metis ‹of-int-mod-ring (int g) = 1› cong-def of-int-mod-ring.rep-eq one-mod-card-int one-mod-ring.rep-eq)
  hence [g=1] (mod p)
  by (metis cong-int-iff int-ops(2))
  thus False
  using ‹[g ≠ 1] (mod p)› by auto

```


qed
thus *?thesis using that g-Def calculation*
by (*metis Euclidean-Division.pos-mod-bound Euclidean-Division.pos-mod-sign mod-homo nat-int of-nat-0-less-iff of-nat-mod p-def residue-primroot-mod to-int-mod-ring-of-int-mod-ring zero-less-card-finite*)
qed

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 $(pr^k)^n \equiv 1 \pmod{p}$.

lemma *omega-properties-ex:*

obtains $\omega :: ('a::prime-card) \text{ mod-ring}$

where $\omega^n = 1$

$\omega \neq 1$

$\forall m. \omega^m = 1 \wedge m \neq 0 \longrightarrow m \geq n$

proof–

obtain $pr :: ('a::prime-card) \text{ mod-ring}$ **where** $a: pr^{p-1} = 1$ **and** $b: pr \neq 1$

and $c: \text{residue-primroot } p \text{ (nat (to-int-mod-ring } pr))$

using *primroot-ex* **by** *blast*

moreover hence $(pr^k)^n = 1$

by (*simp add: p-fact power-mult*)

moreover have $pr^k \neq 1$

proof

assume $pr^k = 1$

hence $(\text{to-int-mod-ring } pr)^k \text{ mod } p = 1$

by (*metis exp-homo of-int-mod-ring.rep-eq of-int-mod-ring-to-int-mod-ring one-mod-ring.rep-eq p-def*)

hence $\text{ord } p \text{ (to-int-mod-ring } pr) \leq k$

by (*simp add: <pr^k = 1> k-bound ord-max*)

hence $\text{ord } p \text{ (nat (to-int-mod-ring } pr)) \leq k$

by (*metis ord-lift*)

also have $\text{ord } p \text{ (nat (to-int-mod-ring } pr)) = p - 1$

using c *primroot-ord*[*of (nat (to-int-mod-ring } pr))*] **by** *blast*

also have $\dots = k * n$

using p -*fact* **by** *simp*

finally have $n \leq 1$

using k -*bound* **by** *simp*

thus *False*

using n -*lst2* **by** *linarith*

qed

moreover have $\forall m. (pr^k)^m = 1 \wedge m \neq 0 \longrightarrow m \geq n$

proof(*rule ccontr*)

assume $\neg (\forall m. (pr^k)^m = 1 \wedge m \neq 0 \longrightarrow m \geq n)$

then obtain m **where** $(pr^k)^m = 1 \wedge m \neq 0 \wedge m < n$ **by** *force*

hence $\text{ord } p \text{ (to-int-mod-ring } pr) \leq k * m$ **using** *ord-max*[*of k*m pr*]

by (*metis calculation(5) mult-is-0 power-mult*)

moreover have $\text{ord } p \text{ (nat (to-int-mod-ring } pr)) = p - 1$ **using** c *primroot-ord* *ord-lift* **by** *simp*

ultimately show *False*

by (*metis <(pr^k)^m = 1 & m < n> add-diff-cancel-right' nat-0-less-mult-iff nat-mult-le-cancel-disj not-less ord-lift p-def p-fact prime-card prime-gt-1-nat zero-less-diff*)

qed

ultimately show *?thesis*

using *that* by *simp*

qed

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

theorem *omega-exists*: $\exists \omega :: ('a::\text{prime-card}) \text{ mod-ring} .$

$$\omega^n = 1 \wedge \omega \neq 1 \wedge (\forall m. \omega^m = 1 \wedge m \neq 0 \longrightarrow m \geq n)$$

using *omega-properties-ex* by *metis*

definition (*omega*::('a::prime-card) mod-ring) =

$$(SOME \omega . (\omega^n = 1 \wedge \omega \neq 1 \wedge (\forall m. \omega^m = 1 \wedge m \neq 0 \longrightarrow m \geq n)))$$

lemma *omega-properties*: $\omega^n = 1 \wedge \omega \neq 1$

$$(\forall m. \omega^m = 1 \wedge m \neq 0 \longrightarrow m \geq n)$$

unfolding *omega-def* using *omega-exists*

by (*smt (verit, best) verit-sko-ex'*)⁺

We define the multiplicative inverse μ of ω .

definition $\mu = \omega^{(n-1)}$

lemma *mu-properties*: $\mu * \omega = 1 \wedge \mu \neq 1$

proof –

have $\omega^{(n-1)} * \omega = \omega^{Suc (n-1)}$

by *simp*

also have $\omega^{Suc (n-1)} = \omega^n$

using *n-lst2* by *simp*

also have $\omega^n = 1$

using *omega-properties(1)* by *auto*

finally show $\mu * \omega = 1$

by (*simp add: mu-def*)

next

show $\mu \neq 1$

using *omega-properties n-lst2* by (*auto simp: mu-def*)

qed

2.4 Some Lemmas on Sums

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

lemma *sum-in*: $(\sum_{i=0..<x::\text{nat}} f i * (y :: ('a \text{ mod-ring}))) = (\sum_{i=0..<x} f i) * (y)$

by(*induction x*) (*auto simp add: algebra-simps*)

lemma *sum-eq*: $(\bigwedge i. i < x \implies f i = g i)$

$$\implies (\sum_{i=0..<x::\text{nat}} f i) = (\sum_{i=0..<x} g i)$$

by(*induction x*) (*auto simp add: algebra-simps*)

lemma *sum-diff-in*: $(\sum_{i=0..<x::\text{nat}} (f i) :: ('a \text{ mod-ring})) - (\sum_{i=0..<x} g i) =$

$$(\sum_{i=0..<x} f i - g i)$$

by(*induction x*) (*auto simp add: algebra-simps*)

lemma *sum-swap*: $(\sum i=0..<(x::nat). \sum j=0..<(y::nat). f i j) =$
 $(\sum j=0..<(y::nat). \sum i=0..<(x::nat). f i j)$
using *Groups-Big.comm-monoid-add-class.sum.swap* **by fast**

lemma *sum-const*: $(\sum i=0..<(x::nat). (c::('a::prime-card) mod-ring)) = (of-int-mod-ring x) * c$
by(*induction x, simp add: algebra-simps, simp add: algebra-simps*)
(*metis distrib-left mult.right-neutral of-int-of-int-mod-ring of-int-of-nat-eq of-nat-Suc*)

lemma *sum-split*: $(r1::nat) < r2 \implies (\sum l = 0..<r1. ((f l)::('a::prime-card) mod-ring)))$
 $+ (\sum l = r1..<r2. f l) = (\sum l = 0..<r2. f l)$
by (*meson less-or-eq-imp-le sum.atLeastLessThan-concat zero-le*)

lemma *sum-index-shift*: $(\sum l = (a::nat)..< b. f(l+c)) = (\sum l = (a+c)..< (b+c). f l)$
by(*induction a arbitrary: b c*) (*metis sum.shift-bounds-nat-ivl*)**+**

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*:

$$(\sum i::nat = 0..<2*nn. f i) = (\sum i = 0..<nn. f (2*i)) + (\sum i = 0..<nn. f (2*i+1))$$

proof(*induction nn*)

case (*Suc n*)

have $(\sum i::nat = 0..<2*(n+1). f i) = (\sum i::nat = 0..<(2*n). f i) + f(2*n+1) + f(2*n)$

by(*simp add: algebra-simps*)

also have $\dots = (\sum i::nat = 0..<n. f (2*i)) + (\sum i::nat = 0..<n. f (2*i+1)) + f(2*n+1) + f(2*n)$

using *Suc* **by simp**

also have $\dots = (\sum i::nat = 0..<(Suc n). f (2*i)) + (\sum i::nat = 0..<(Suc n). f (2*i+1))$

by(*simp add: algebra-simps*)

finally show *?case* **by simp**

qed *simp*

lemma *sum-even-odd-split*: $even (a::nat) \implies (\sum j=0..<(a div 2). f (2*j)) + (\sum j=0..<(a div 2). f (2*j+1)) = (\sum j=0..<a. f j)$

by (*induction a, simp*)(*metis even-two-times-div-two sum-splice*)

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

by (*metis sum-splice*)

lemma *sum-neg-in*: $-(\sum j = 0..<l. (f j)::('a mod-ring)) = (\sum j = 0..<l. - f j)$

by (*simp add: sum-negf*)

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 $(1-x) * (\sum l = 0..<r. (x::('a \text{ mod-ring}))^\wedge l) = (1-x^\wedge r)$

proof–

have $0: x * (\sum l = 0..<r. x^\wedge l) = (\sum l = 0..<r. x^\wedge (\text{Suc } l))$ **using** *sum-in[of $\lambda l. x^\wedge l$ x r]*

by (*simp add: algebra-simps*)

have $1: (\sum l = 0..<r. x^\wedge l) - (\sum l = 0..<r. x^\wedge (\text{Suc } l)) = (\sum l = 0..<r. x^\wedge l - x^\wedge (\text{Suc } l))$

by (*rule sum-diff-in*)

have $2: (\sum l = 0..<r. x^\wedge l - x^\wedge (\text{Suc } l)) = 1 - x^\wedge r$

by (*induction r simp+*)

thus *?thesis*

by (*simp add: lessThan-atLeast0 one-diff-power-eq*)

qed

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 $\omega :: ('a :: prime-card \text{ mod-ring})$

fixes $\mu :: ('a \text{ mod-ring})$

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

assumes *mu-properties*: $\mu * \omega = 1$

begin

lemma *mu-properties'*: $\mu \neq 1$

using *omega-properties mu-properties* **by** *auto*

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]:

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

Or for single vector entries:

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

Formally:

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

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

We define the inverse transform INTT by matrices:

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

Per component:

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

definition *intt xs* $i = (\sum_{j=0..<n}. (xs ! j) * \mu^{i \cdot j})$

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

unfolding *NTT-def ntt-def* **using** *n-def length-map*[of - [0..<n]]

by *simp*

lemma *length-INTT*:

assumes *n-def*: *length numbers = n*

shows *length (INTT numbers) = n*

unfolding *INTT-def intt-def* **using** *n-def length-map*[*of - [0..<n]*]

by *simp*

3.2 Correctness Proof of NTT and INTT

We prove NTT and INTT correct: By taking $\text{INTT}(\text{NTT}(x))$ 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:

$$\text{INTT}(\text{NTT}(\vec{x})) = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \mu & \mu^2 & \dots & \mu^{n-1} \\ 1 & \mu^2 & \mu^4 & \dots & \mu^{2 \cdot (n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \mu^{n-1} & \mu^{2 \cdot (n-1)} & \dots & \mu^{(n-1) \cdot (n-1)} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2 \cdot (n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2 \cdot (n-1)} & \dots & \omega^{(n-1) \cdot (n-1)} \end{pmatrix} \cdot \vec{x}$$

A resulting entry is of the following form:

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

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 –

have $0: \bigwedge i. i < n \implies (\text{INTT } (\text{NTT numbers})) ! i = \text{intt } (\text{NTT numbers}) i$ **using** *n-def length-NTT*

unfolding *INTT-def NTT-def intt-def* **by** *simp*

Major sublemma.

have $1: \bigwedge i. i < n \implies \text{intt } (\text{NTT numbers}) i = (\text{of-int-mod-ring } n) * \text{numbers} ! i$

proof –

fix *i*

assume *i-assms: i < n*

First, simplify by some chains of equations.

hence $1: \text{intt } (\text{NTT numbers}) i =$

$$\begin{aligned}
&(\sum l = 0..<n. \\
&(\sum j = 0..<n. \text{numbers} ! j * \omega^{\wedge(l * j)} * \mu^{\wedge(i * l)})
\end{aligned}$$

unfolding *NTT-def intt-def ntt-def* **using** *n-def length-map nth-map* **by** *simp*

also have $2: \dots =$

$$\begin{aligned}
&(\sum l = 0..<n. \\
&(\sum j = 0..<n. (\text{numbers} ! j * \omega^{\wedge(l * j)} * \mu^{\wedge(i * l)}))
\end{aligned}$$

using *sum-in* **by** (*simp add: sum-distrib-right*)

also have $3: \dots =$

$$\begin{aligned}
&(\sum j = 0..<n. \\
&(\sum l = 0..<n. (\text{numbers} ! j * \omega^{\wedge(l * j)} * \mu^{\wedge(i * l)}))) \text{ **using** } \textit{sum-swap} \text{ **by** } \textit{fast}
\end{aligned}$$

As in the informal proof, we consider three cases. First $j = i$.

have $iisj: \bigwedge j. j = i \implies (\sum l = 0..<n. (\text{numbers} ! j * \omega^{\wedge(l * j)} * \mu^{\wedge(i * l)})) = (\text{numbers} ! j) * (\text{of-int-mod-ring } n)$

proof –

fix *j*

assume $j=i$

hence $\bigwedge l. l < n \implies (\text{numbers} ! j * \omega^{\wedge(l * j)} * \mu^{\wedge(i * l)}) = (\text{numbers} ! j)$

by (simp add: left-right-inverse-power mult.commute mu-properties(1))
moreover have $\bigwedge l. l < n \implies \text{numbers ! } j * \omega^{(l * j)} * \mu^{(i * l)} = \text{numbers ! } j$
using calculation **by** blast

$\omega^{il} \cdot \omega^{jl} = 1$. Thus, we sum over 1 n times, which gives the goal.

ultimately show $(\sum l = 0..<n. (\text{numbers ! } j * \omega^{(l * j)} * \mu^{(i * l)})) =$
 $(\text{numbers ! } j) * (\text{of-int-mod-ring } n)$
using n-def sum-const[of numbers ! j n] exp-rule[of ω μ] mu-properties(1)
by (metis (no-types, lifting) atLeastLessThan-iff mult.commute sum.cong)

qed

Case $j < i$.

have jlsi: $\bigwedge j. j < i \implies (\sum l = 0..<n. (\text{numbers ! } j * \omega^{(l * j)} * \mu^{(i * l)})) = 0$

proof–

fix j
assume j-assms: $j < i$
hence 00: $\bigwedge (c :: ('a :: \text{prime-card}) \text{ mod-ring}) a b. c * a^{j * b} * b^i = (a * b)^j * (c * b^{i-j})$
using algebra-simps
by (smt (z3) le-less ordered-cancel-comm-monoid-diff-class.add-diff-inverse power-add)

A geometric sum over μ^l remains.

have 01: $(\sum l = 0..<n. (\text{numbers ! } j * \omega^{(l * j)} * \mu^{(i * l)})) =$
 $(\sum l = 0..<n. (\text{numbers ! } j * (\mu^l)^{i-j}))$
apply(rule sum-eq)
using mu-properties(1) 00 algebra-simps(23)
by (smt (z3) mult.commute mult.left-neutral power-mult power-one)

have 02: $\dots = \text{numbers ! } j * (\sum l = 0..<n. ((\mu^l)^{i-j}))$
using sum-in[of $\lambda l. \text{numbers ! } j * (\mu^l)^{i-j}$ numbers ! j n]
by (simp add: mult-hom.hom-sum)

moreover have 03: $(\sum l = 0..<n. ((\mu^l)^{i-j})) =$
 $(\sum l = 0..<n. ((\mu^{i-j})^l))$

by(rule sum-eq) (metis mult.commute power-mult)

have $\mu^{i-j} \neq 1$

proof

assume $\mu^{i-j} = 1$

hence ord p (to-int-mod-ring μ) $\leq i-j$

by (simp add: j-assms not-le ord-max)

moreover hence ord p (to-int-mod-ring ω) $\leq i-j$

by (metis $\langle \mu^{i-j} = 1 \rangle$ diff-is-0-eq exp-rule j-assms leD mult.comm-neutral mult.commute mu-properties(1) ord-max)

moreover hence $i-j < n$

using j-assms i-assms p-fact k-bound n-lst2 **by** linarith

moreover have ord p (to-int-mod-ring ω) = n **using** omega-properties n-lst2 **unfolding** ord-def

by (metis (no-types) $\langle \mu^{i-j} = 1 \rangle$ calculation(3) diff-is-0-eq j-assms leD left-right-inverse-power mult.comm-neutral mult-cancel-left mu-properties(1) omega-properties(3) zero-neq-one)

ultimately show False **by** simp

qed

Application of the lemma for geometric sums.

ultimately have $(1 - \mu^{i-j}) * (\sum l = 0..<n. ((\mu^{i-j})^{\wedge} l)) = (1 - (\mu^{i-j})^{\wedge} n)$
using *geo-sum*[of μ^{i-j} n] **by** *simp*
moreover have $(\mu^{i-j})^{\wedge} n = 1$
by (*metis* (*no-types*) *left-right-inverse-power* *mult.commute* *mult.right-neutral* *mu-properties*(1) *omega-properties*(1) *power-mult* *power-one*)

The sum for the current index is 0.

ultimately have $(\sum l = 0..<n. ((\mu^{i-j})^{\wedge} l)) = 0$
by (*metis* $\langle \mu^{i-j} \neq 1 \rangle$ *divisors-zero* *eq-iff-diff-eq-0*)
thus $(\sum l = 0..<n. \text{numbers} ! j * \omega^{l * j} * \mu^{i * l}) = 0$ **using** 01 02 03 **by** *simp*
qed

Case $i < j$. We also rewrite the whole summation until the lemma for geometric sums is applicable. From this, we conclude that the term is 0.

have $\text{ilsj} : \wedge j. i < j \wedge j < n \implies (\sum l = 0..<n. (\text{numbers} ! j * \omega^{l * j} * \mu^{i * l})) = 0$

proof–

fix j

assume $ij\text{-Assm} : i < j \wedge j < n$

hence 00: $\wedge (c :: ('a :: \text{prime-card}) \text{mod-ring}) a b. (a * b)^{\wedge} i * (c * b^{j-i}) = c * a^{\wedge} i * b^{\wedge} j$

by (*auto simp*: *field-simps* *simp flip*: *power-add*)

have 01: $(\sum l = 0..<n. (\text{numbers} ! j * \omega^{l * j} * \mu^{i * l})) =$
 $(\sum l = 0..<n. (\text{numbers} ! j * (\omega^{\wedge} l)^{\wedge} (j-i)))$

apply(*rule sum-eq*) **subgoal for** l

using *mu-properties*(1) 00[*of* $\omega^{\wedge} l$ $\mu^{\wedge} l$ *numbers* ! j] *algebra-simps*(23)

by (*smt* (*z3*) 00 *left-right-inverse-power* *mult.assoc* *mult.commute* *mult.right-neutral* *power-mult*)
done

moreover have 02: $(\sum l = 0..<n. (\text{numbers} ! j * (\omega^{\wedge} l)^{\wedge} (j-i))) =$
 $\text{numbers} ! j * (\sum l = 0..<n. ((\omega^{\wedge} l)^{\wedge} (j-i)))$

by (*simp add*: *mult-hom.hom-sum*)

moreover have 03: $(\sum l = 0..<n. ((\omega^{\wedge} l)^{\wedge} (j-i))) =$
 $(\sum l = 0..<n. (((\omega^{\wedge} l)^{\wedge} (j-i))^{\wedge} l))$

by(*rule sum-eq*) (*metis* *mult.commute* *power-mult*)

have $\omega^{j-i} \neq 1$

proof

assume $\omega^{j-i} = 1$

hence $\text{ord } p$ (*to-int-mod-ring* ω) $\leq j-i$ **using** *ord-max*[*of* $j-i$ ω] *ij-Assm* **by** *simp*

moreover have $\text{ord } p$ (*to-int-mod-ring* ω) $= p-1$

by (*meson* $\langle \omega^{j-i} = 1 \rangle$ *diff-is-0-eq* *diff-le-self* *ij-Assm* *leD* *le-trans* *omega-properties*(3))

ultimately show *False*

by (*meson* $\langle \omega^{j-i} = 1 \rangle$ *diff-is-0-eq* *diff-le-self* *ij-Assm* *leD* *le-trans* *omega-properties*(3))

qed

Geometric sum.

ultimately have $(1 - \omega^{j-i}) * (\sum l = 0..<n. ((\omega^{j-i})^{\wedge} l)) = (1 - (\omega^{j-i})^{\wedge} n)$

using *geo-sum*[*of* ω^{j-i} n] **by** *simp*

moreover have $(\omega^{j-i})^{\wedge} n = 1$

by (*metis* (*no-types*) *mult.commute* *omega-properties*(1) *power-mult* *power-one*)

ultimately have $(\sum l = 0..<n. ((\omega^{j-i})^{\wedge} l)) = 0$

by (metis $\langle \omega \wedge (j - i) \neq 1 \rangle$ eq-iff-diff-eq-0 no-zero-divisors)
thus $(\sum l = 0..<n. \text{numbers} ! j * \omega \wedge (l * j) * \mu \wedge (i * l)) = 0$ **using** 01 02 03 **by simp**
qed

We compose the cases $j < i$, $j = i$ and $j > i$ to a complete summation over index j .

have $(\sum j = 0..<i. \sum l = 0..<n. \text{numbers} ! j * \omega \wedge (l * j) * \mu \wedge (i * l)) = 0$ **using** jlsi **by simp**
moreover have $(\sum j = i..<i+1. \sum l = 0..<n. \text{numbers} ! j * \omega \wedge (l * j) * \mu \wedge (i * l)) = \text{numbers} ! i * (\text{of-int-mod-ring } n)$ **using** iisj **by simp**
moreover have $(\sum j = (i+1)..<n. \sum l = 0..<n. \text{numbers} ! j * \omega \wedge (l * j) * \mu \wedge (i * l)) = 0$
using ilsj **by simp**
ultimately have $(\sum j = 0..<n. \sum l = 0..<n. \text{numbers} ! j * \omega \wedge (l * j) * \mu \wedge (i * l)) = \text{numbers} ! i * (\text{of-int-mod-ring } n)$ **using** i-assms sum-split
by (smt (z3) add.commute add.left-neutral int-ops(2) less-imp-of-nat-less of-nat-add of-nat-eq-iff of-nat-less-imp-less)

Index-wise equality can be shown.

thus $\text{intt } (NTT \text{ numbers}) i = \text{of-int-mod-ring } (int \ n) * \text{numbers} ! i$ **using** 1 2 3
by (metis mult.commute)
qed
have 2: $\bigwedge i. i < n \implies (\text{map } ((* \ (\text{of-int-mod-ring } (int \ n)))) \ \text{numbers}) ! i = (\text{of-int-mod-ring } (int \ n)) * (\text{numbers} ! i)$
by (simp add: n-def)

We relate index-wise equality to the function definition.

show ?thesis
apply(rule nth-equalityI)
subgoal my-subgoal
unfolding INTT-def NTT-def
apply (simp add: n-def)
done
subgoal for i
using 0 1 2 n-def algebra-simps my-subgoal length-map
apply auto
done
done
qed

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

theorem inv-ntt-correct:

assumes n-def: length numbers = n

shows $NTT (INTT \text{ numbers}) = \text{map } (\lambda x. (\text{of-int-mod-ring } n) * x) \ \text{numbers}$

proof–

have 0: $\bigwedge i. i < n \implies (NTT (INTT \text{ numbers})) ! i = \text{ntt } (INTT \text{ numbers}) i$ **using** n-def length-NTT

unfolding INTT-def NTT-def intt-def **by simp**

have 1: $\bigwedge i. i < n \implies \text{ntt } (INTT \text{ numbers}) i = (\text{of-int-mod-ring } n) * \text{numbers} ! i$

proof–

fix i

assume i-assms: $i < n$

hence $1: ntt (INTT\ numbers) i =$
 $(\sum l = 0..<n. (\sum j = 0..<n. numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l))$
unfolding *INTT-def ntt-def intt-def using n-def length-map nth-map by simp*
hence $2: \dots = (\sum l = 0..<n. (\sum j = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)))$ **using** *sum-in by simp*
have $3: \dots = (\sum j = 0..<n. (\sum l = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)))$ **using** *sum-swap by fast*
have $iisj: \bigwedge j. j = i \implies (\sum l = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)) = (numbers ! j) *$
(of-int-mod-ring n)
proof–
fix j
assume $j=i$
hence $\bigwedge l. l < n \implies (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l) = (numbers ! j)$
by *(simp add: left-right-inverse-power mult.commute mu-properties(1))*
moreover have $\bigwedge l. l < n \implies numbers ! j * \mu \wedge (l * j) * \omega \wedge (i * l) = numbers ! j$
using *calculation by blast*
ultimately show $(\sum l = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)) = (numbers ! j) *$
(of-int-mod-ring n)
using *n-def sum-const[of numbers ! j n] exp-rule[of ω μ] mu-properties(1)*
by *(metis (no-types, lifting) atLeastLessThan-iff mult.commute sum.cong)*
qed
have $jlsi: \bigwedge j. j < i \implies (\sum l = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)) = 0$
proof–
fix j
assume $j-assms: j < i$
hence $00: \bigwedge (c :: ('a::prime-card) mod-ring) a b. c * a \wedge j * b \wedge i = (a * b) \wedge j * (c * b \wedge (i - j))$
using *algebra-simps*
by *(smt (z3) le-less ordered-cancel-comm-monoid-diff-class.add-diff-inverse power-add)*
have $01: (\sum l = 0..<n. (numbers ! j * \mu \wedge (l * j)) * \omega \wedge (i * l)) =$
 $(\sum l = 0..<n. (numbers ! j * (\omega \wedge l) \wedge (i - j)))$
apply *(rule sum-eq)*
using *mu-properties(1) 00 algebra-simps(23)*
by *(smt (z3) mult.commute mult.left-neutral power-mult power-one)*
moreover have $02: \dots = numbers ! j * (\sum l = 0..<n. ((\omega \wedge l) \wedge (i - j)))$
using *sum-in[of $\lambda l. numbers ! j * (\mu \wedge l) \wedge (i - j)$ numbers ! j n]*
by *(simp add: mult-hom.hom-sum)*
moreover have $03: (\sum l = 0..<n. ((\omega \wedge l) \wedge (i - j))) =$
 $(\sum l = 0..<n. ((\omega \wedge (i - j)) \wedge l))$
by *(rule sum-eq) (metis mult.commute power-mult)*
have $\omega \wedge (i - j) \neq 1$
proof
assume $\omega \wedge (i - j) = 1$
hence $ord\ p (to-int-mod-ring\ \omega) \leq i - j$
by *(simp add: j-assms not-le ord-max)*
moreover have $ord\ p (to-int-mod-ring\ \omega) = n$ **using** *omega-properties n-lst2 unfolding*
ord-def
by *(meson $\langle \omega \wedge (i - j) = 1 \rangle$ diff-is-0-eq diff-le-self i-assms j-assms leD le-trans)*
ultimately show *False*

by (metis i-assms leD less-imp-diff-less)
 qed
 ultimately have $(1 - \omega^{i-j}) * (\sum l = 0..<n. ((\omega^{i-j})^{\wedge} l)) = (1 - (\omega^{i-j})^{\wedge} n)$
 using geo-sum[of ω^{i-j} n] by simp
 moreover have $(\omega^{i-j})^{\wedge} n = 1$
 by (metis (no-types) mult.commute omega-properties(1) power-mult power-one)
 ultimately have $(\sum l = 0..<n. ((\omega^{i-j})^{\wedge} l)) = 0$
 by (metis $\langle \omega^{i-j} \neq 1 \rangle$ divisors-zero eq-iff-diff-eq-0)
 thus $(\sum l = 0..<n. \text{numbers} ! j * \mu^{l * j} * \omega^{i * l}) = 0$ using 01 02 03 by simp
 qed
 have $!j : \bigwedge j. i < j \wedge j < n \implies (\sum l = 0..<n. (\text{numbers} ! j * \mu^{l * j} * \omega^{i * l})) = 0$
 proof-
 fix j
 assume ij-Assm: $i < j \wedge j < n$
 hence 00: $\bigwedge (c :: ('a :: \text{prime-card}) \text{mod-ring}) a b. (a * b)^{\wedge} i * (c * b^{j-i}) = c * a^{\wedge} i * b^{\wedge} j$
 by (simp add: field-simps flip: power-add)
 have 01: $(\sum l = 0..<n. (\text{numbers} ! j * \mu^{l * j} * \omega^{i * l})) =$
 $(\sum l = 0..<n. (\text{numbers} ! j * (\mu^{\wedge} l)^{\wedge} (j-i)))$
 apply(rule sum-eq) subgoal for l
 using mu-properties(1) 00[of $\omega^{\wedge} l \mu^{\wedge} l$ numbers ! j] algebra-simps(23)
 by (smt (z3) 00 left-right-inverse-power mult.assoc mult.commute mult.right-neutral power-mult)
 done
 moreover have 02: $(\sum l = 0..<n. (\text{numbers} ! j * (\mu^{\wedge} l)^{\wedge} (j-i))) =$
 $\text{numbers} ! j * (\sum l = 0..<n. ((\mu^{\wedge} l)^{\wedge} (j-i)))$
 by (simp add: mult-hom.hom-sum)
 moreover have 03: $(\sum l = 0..<n. ((\mu^{\wedge} l)^{\wedge} (j-i))) =$
 $(\sum l = 0..<n. (((\mu^{\wedge} (j-i))^{\wedge} l))$
 by(rule sum-eq) (metis mult.commute power-mult)
 have $\mu^{\wedge} (j-i) \neq 1$
 proof
 assume $\mu^{\wedge} (j-i) = 1$
 hence ord p (to-int-mod-ring μ) $\leq j-i$
 by (simp add: ij-Assm not-le ord-max)
 moreover hence ord p (to-int-mod-ring ω) $\leq j-i$
 by (metis $\langle \mu^{\wedge} (j-i) = 1 \rangle$ diff-is-0-eq exp-rule ij-Assm leD mult.comm-neutral mult.commute
 mu-properties(1) ord-max)
 moreover hence $j-i < n$ using ij-Assm i-assms p-fact k-bound n-lst2 by linarith
 moreover have ord p (to-int-mod-ring ω) = n using omega-properties n-lst2 unfolding ord-def
 by (metis (no-types) $\langle \mu^{\wedge} (j-i) = 1 \rangle$ calculation(3) diff-is-0-eq ij-Assm leD left-right-inverse-power
 mult.comm-neutral mult-cancel-left mu-properties(1) omega-properties(3) zero-neq-one)
 ultimately show False by simp
 qed
 ultimately have $(1 - \mu^{\wedge} (j-i)) * (\sum l = 0..<n. ((\mu^{\wedge} (j-i))^{\wedge} l)) = (1 - (\mu^{\wedge} (j-i))^{\wedge} n)$
 using geo-sum[of $\mu^{\wedge} (j-i)$ n] by simp
 moreover have $(\mu^{\wedge} (j-i))^{\wedge} n = 1$
 by (metis (no-types) left-right-inverse-power mult.commute mult.right-neutral mu-properties(1)
 omega-properties(1) power-mult power-one)
 ultimately have $(\sum l = 0..<n. ((\mu^{\wedge} (j-i))^{\wedge} l)) = 0$
 by (metis $\langle \mu^{\wedge} (j-i) \neq 1 \rangle$ eq-iff-diff-eq-0 no-zero-divisors)

```

    thus ( $\sum l = 0..<n. \text{numbers} ! j * \mu \wedge(l * j) * \omega \wedge(i * l) = 0$ ) using 01 02 03 by simp
  qed
  have ( $\sum j = 0..<i. \sum l = 0..<n. \text{numbers} ! j * \mu \wedge(l * j) * \omega \wedge(i * l) = 0$ ) using jlsi by simp
  moreover have ( $\sum j = i..<i+1. \sum l = 0..<n. \text{numbers} ! j * \mu \wedge(l * j) * \omega \wedge(i * l) = \text{numbers} ! i * (\text{of-int-mod-ring } n)$ ) using iisj by simp
  moreover have ( $\sum j = (i+1)..<n. \sum l = 0..<n. \text{numbers} ! j * \mu \wedge(l * j) * \omega \wedge(i * l) = 0$ )
using ilsj by simp
  ultimately have ( $\sum j = 0..<n. \sum l = 0..<n. \text{numbers} ! j * \mu \wedge(l * j) * \omega \wedge(i * l) =$ 
 $\text{numbers} ! i * (\text{of-int-mod-ring } n)$ ) using i-assms sum-split
  by (smt (z3) add.commute add.left-neutral int-ops(2) less-imp-of-nat-less of-nat-add of-nat-eq-iff
of-nat-less-imp-less)
  thus ntt (INTT numbers) i = of-int-mod-ring (int n) * numbers ! i using 1 2 3
  by (metis mult.commute)
  qed
  have 2:  $\bigwedge i. i < n \implies (\text{map } ((*)) (\text{of-int-mod-ring } (\text{int } n))) \text{ numbers} ! i = (\text{of-int-mod-ring } (\text{int } n)) * (\text{numbers} ! i)$ 
  by (simp add: n-def)
  show ?thesis
  apply(rule nth-equalityI)
  subgoal my-little-subgoal
  unfolding INTT-def NTT-def
  apply (simp add: n-def)
  done
  subgoal for i
  using 0 1 2 n-def algebra-simps my-little-subgoal length-map
  apply auto
  done
  done
  qed
end
end

```

```

theory Butterfly
  imports NTT HOL-Library.Discrete
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^N
begin

```

4.1 Recursive Definition

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

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

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

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

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]: $\text{length (filter } P [0..<l]) < \text{Suc } l$
by (*metis diff-zero le-imp-less-Suc length-filter-le length-upt*)

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  $\Rightarrow$  ('a mod-ring) list where
FNTT [] = []
FNTT [a] = [a]
FNTT nums = (let nn = length nums;
               nums1 = [nums!i. i  $\leftarrow$  filter even [0..<nn]];
               nums2 = [nums!i. i  $\leftarrow$  filter odd [0..<nn]];
               fntt1 = FNTT nums1;
               fntt2 = FNTT nums2;
               sum1 = map2 (+) fntt1 (map2 ( $\lambda$  x k. x*( $\omega^{\wedge}$ (n div nn) * k)) fntt2 [0..<(nn div
2)]));
               sum2 = map2 (-) fntt1 (map2 ( $\lambda$  x k. x*( $\omega^{\wedge}$ (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 *NTT*s 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:

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

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

definition *nntt-gen numbers degr i* = $(\sum_{j=0..<(\text{length numbers})} (\text{numbers } ! j) * \omega^{\wedge}((n \text{ div } \text{degr}) * i * j))$

definition *NTT-gen degr numbers* = $\text{map (nntt-gen numbers (degr)) } [0..< \text{length numbers}]$

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 $\text{length numbers} = n$

shows *NTT-gen n numbers = NTT numbers*
unfolding *NTT-gen-def ntt-gen-def NTT-def ntt-def*
using *assms* **by** *simp*

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*
by(*induction l*) *auto*

lemma *length-odd-filter: length [f i . i <- (filter odd [0..<l])] = l div 2*
by(*induction l*) *auto*

lemma *map2-length: length (map2 f xs ys) = min (length xs) (length ys)*
by (*induction xs arbitrary: ys*) *auto*

lemma *map2-index: i < length xs \implies i < length ys \implies (map2 f xs ys) ! i = f (xs ! i) (ys ! i)*
by (*induction xs arbitrary: ys i*) *auto*

lemma *filter-last-not: $\neg P x \implies$ filter P (xs@[x]) = filter P xs*
by *simp*

lemma *filter-even-map: filter even [0..<2*(x::nat)] = map ((* (2::nat)) [0..<x])*
by(*induction x*) *simp+*

lemma *filter-even-nth: 2*j < l \implies 2*x = l \implies (filter even [0..<l] ! j) = (2*j)*
using *filter-even-map[of x] nth-map[of j filter even [0..<l] (* 2)]* **by** *auto*

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

lemma *filter-odd-nth: 2*j < l \implies 2*x = l \implies (filter odd [0..<l] ! j) = (2*j+1)*
using *filter-odd-map[of x] nth-map[of j filter even [0..<l] (* 2)]* **by** *auto*

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 \implies $\omega = -1$*
using *omega-properties(1) omega-properties(2) power2-eq-1-iff* **by** *blast*

lemma *n-min1-gr2:*
assumes $n > 2$
shows $\omega \frown (n \text{ div } 2) = -1$
proof -
have $\omega \frown (n \text{ div } 2) \neq -1 \implies \text{False}$
proof -
assume $\omega \frown (n \text{ div } 2) \neq -1$
hence *False*
proof(*cases $\omega \frown (n \text{ div } 2) = 1$*)

case *True*
then show *?thesis using omega-properties(3)*
by (*metis Euclidean-Division.div-eq-0-iff div-less-dividend leD less-le-trans n-lst2 one-less-numeral-iff pos2 semiring-norm(76) zero-neq-numeral*)
next
case *False*
hence $(\omega \wedge (n \text{ div } 2)) \wedge (2::\text{nat}) \neq 1$
by (*smt (verit, ccfv-threshold) n-two-pot One-nat-def $\langle \omega \wedge (n \text{ div } 2) \neq -1 \rangle$ diff-zero leD n-lst2 not-less-eq omega-properties(1) one-less-numeral-iff one-power2 power2-eq-square power-mult power-one-right power-strict-increasing-iff semiring-norm(76) square-eq-iff two-powr-div two-powrs-div*)
moreover have $(n \text{ div } 2) * 2 = n$ **using** *n-two-pot n-lst2*
by (*metis One-nat-def Suc-lessD assms div-by-Suc-0 one-less-numeral-iff power-0 power-one-right power-strict-increasing-iff semiring-norm(76) two-powrs-div*)
ultimately show *?thesis using omega-properties(1)*
by (*metis power-mult*)
qed
thus *False by simp*
qed
then show *?thesis by auto*
qed

lemma *div-exp-sub*: $2^l < n \implies n \text{ div } (2^l) = 2^{(N-l)}$ **using** *n-two-pot*
by (*smt (z3) One-nat-def diff-is-0-eq diff-le-diff-pow div-if div-le-dividend eq-imp-le le-0-eq le-Suc-eq n-lst2 nat-less-le not-less-eq-eq numeral-2-eq-2 power-0 two-powr-div*)

lemma *omega-div-exp-min1*:
assumes $2^{(Suc\ l)} \leq n$
shows $(\omega \wedge (n \text{ div } 2^{(Suc\ l)})) \wedge (2^l) = -1$
proof–
have $(\omega \wedge (n \text{ div } 2^{(Suc\ l)})) \wedge (2^l) = \omega \wedge ((n \text{ div } 2^{(Suc\ l)}) * 2^l)$
by (*simp add: power-mult*)
moreover have $(n \text{ div } 2^{(Suc\ l)}) = 2^{(N - Suc\ l)}$ **using** *assms div-exp-sub*
by (*metis n-two-pot eq-imp-le le-neq-implies-less one-less-numeral-iff power-diff power-inject-exp semiring-norm(76) zero-neq-numeral*)
moreover have $N \geq Suc\ l$ **using** *assms n-two-pot*
by (*metis diff-is-0-eq diff-le-diff-pow gr0I leD le-refl*)
moreover hence $(2::\text{nat})^{(N - Suc\ l)} * 2^l = 2^{(N - 1)}$
by (*metis Nat.add-diff-assoc diff-Suc-1 diff-diff-cancel diff-le-self le-add1 le-add-diff-inverse plus-1-eq-Suc power-add*)
ultimately show *?thesis*
by (*metis n-two-pot One-nat-def $\langle n \text{ div } 2^{(Suc\ l)} = 2^{(N - Suc\ l)} \rangle$ diff-Suc-1 div-exp-sub n-lst2 n-min1-2 n-min1-gr2 nat-less-le nat-power-eq-Suc-0-iff one-less-numeral-iff power-inject-exp power-one-right semiring-norm(76)*)
qed

lemma *omg-n-2-min1*: $\omega \wedge (n \text{ div } 2) = -1$
by (*metis n-lst2 n-min1-2 n-min1-gr2 nat-less-le numeral-Bit0-div-2 numerals(1) power-one-right*)

lemma *neg-cong*: $-(x::('a \text{ mod-ring})) = -y \implies x = y$ **by** *simp*

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

theorem *FNTT-NTT-gen-eq*: $\text{length numbers} = 2^l \implies 2^l \leq n \implies \text{FNTT numbers} = \text{NTT-gen}(\text{length numbers}) \text{ numbers}$

proof(*induction l arbitrary: numbers*)

case 0

then show *?case unfolding NTT-gen-def ntt-gen-def*

by (*auto simp: length-Suc-conv*)

next

case (*Suc l*)

We define some lists that are used during the recursive call.

define *numbers1* **where** $\text{numbers1} = [\text{numbers!}i \ . \ i <- (\text{filter even } [0..<\text{length numbers}])]$

define *numbers2* **where** $\text{numbers2} = [\text{numbers!}i \ . \ i <- (\text{filter odd } [0..<\text{length numbers}])]$

define *fnnt1* **where** $\text{fnnt1} = \text{FNTT numbers1}$

define *fnnt2* **where** $\text{fnnt2} = \text{FNTT numbers2}$

define *sum1* **where**

$\text{sum1} = \text{map2 } (+) \ \text{fnnt1} \ (\text{map2 } (\lambda x k. \ x * (\omega^{\wedge} (n \ \text{div} \ (\text{length numbers})) * k)))$
 $\text{fnnt2 } [0..<((\text{length numbers}) \ \text{div} \ 2)]$

define *sum2* **where**

$\text{sum2} = \text{map2 } (-) \ \text{fnnt1} \ (\text{map2 } (\lambda x k. \ x * (\omega^{\wedge} (n \ \text{div} \ (\text{length numbers})) * k)))$
 $\text{fnnt2 } [0..<((\text{length numbers}) \ \text{div} \ 2)]$

define *l1* **where** $l1 = \text{length numbers1}$

define *l2* **where** $l2 = \text{length numbers2}$

define *llen* **where** $llen = \text{length numbers}$

Properties of those lists.

have *numbers1-even*: $\text{length numbers1} = 2^l$

using *numbers1-def length-even-filter Suc* **by** *simp*

have *numbers2-even*: $\text{length numbers2} = 2^l$

using *numbers2-def length-odd-filter Suc* **by** *simp*

have *numbers1-fntt*: $\text{fnnt1} = \text{NTT-gen } (2^l) \ \text{numbers1}$

using *fnnt1-def Suc.IH[of numbers1] numbers1-even Suc(3)* **by** *simp*

hence *fnnt1-by-index*: $\text{fnnt1 } ! \ i = \text{ntt-gen numbers1 } (2^l) \ i$ **if** $i < 2^l$ **for** i

unfolding *NTT-gen-def* **by** (*simp add: numbers1-even that*)

have *numbers2-fntt*: $\text{fnnt2} = \text{NTT-gen } (2^l) \ \text{numbers2}$

using *fnnt2-def Suc.IH[of numbers2] numbers2-even Suc(3)* **by** *simp*

hence *fnnt2-by-index*: $\text{fnnt2 } ! \ i = \text{ntt-gen numbers2 } (2^l) \ i$ **if** $i < 2^l$ **for** i

unfolding *NTT-gen-def*

by (*simp add: numbers2-even that*)

have *fnnt1-length*: $\text{length fnnt1} = 2^l$ **unfolding** *numbers1-fntt NTT-gen-def numbers1-def*

using *numbers1-def numbers1-even* **by** *force*

have *fnnt2-length*: $\text{length fnnt2} = 2^l$ **unfolding** *numbers2-fntt NTT-gen-def numbers2-def*

using *numbers2-def numbers2-even* **by** *force*

We show that the list resulting from *FNTT* is equal to the *NTT* list. First, we prove *FNTT* and *NTT* to be equal concerning their first halves.

have *before-half*: $\text{map } (\text{ntt-gen numbers } llen) \ [0..<(llen \ \text{div} \ 2)] = \text{sum1}$

proof–

Length is important, since we want to use list lemmas later on.

```

have 00:length (map (ntt-gen numbers llen) [0..<(llen div 2)]) = length sum1
  unfolding sum1-def llen-def
  using Suc(2) map2-length[of - fntt2 [0..<length numbers div 2]]
    map2-length[of (+) fntt1 (map2 ( $\lambda x y. x * \omega^{(n \text{ div } \text{length numbers} * y)}$ ) fntt2 [0..<length
numbers div 2])]
  fntt1-length fntt2-length by (simp add: mult-2)
have 01:length sum1 =  $2^{\lceil l \rceil}$  unfolding sum1-def
  using 00 Suc.premis(1) sum1-def unfolding llen-def by auto

```

We show equality by extensionality w.r.t. indices.

```

have 02:(map (ntt-gen numbers llen) [0..<(llen div 2)]) ! i = sum1 ! i
  if  $i < 2^{\lceil l \rceil}$  for  $i$ 
proof-

```

First simplify this term.

```

  have 000:(map (ntt-gen numbers llen) [0..<(llen div 2)]) ! i =
    ntt-gen numbers llen i
  using 00 01 that by auto

```

Expand the definition of *sum1* and massage the result.

```

  moreover have 001:sum1 ! i = (fntt1!i) + (fntt2!i) * ( $\omega^{(n \text{ div } \text{llen}) * i}$ )
  unfolding sum1-def using map2-index
  00 01 NTT-gen-def add.left-neutral diff-zero fntt1-length length-map length-upt map2-map-map
map-nth nth-upt numbers2-even numbers2-fntt that llen-def by force
  moreover have 002:(fntt1!i) = ( $\sum_{j=0..<l1. (\text{numbers1} ! j) * \omega^{(n \text{ div } (2^{\lceil l \rceil}) * i * j)}$ )
  unfolding l1-def
  using fntt1-by-index[of i] that unfolding ntt-gen-def by simp
  have 003:... = ( $\sum_{j=0..<l1. (\text{numbers} ! (2 * j)) * \omega^{(n \text{ div } \text{llen}) * i * (2 * j)}$ )
  apply (rule sum-rules(2))
  subgoal for  $j$  unfolding numbers1-def
  apply(subst llen-def[symmetric])
  proof-
  assume  $\text{ass}: j < l1$ 
  hence map (! numbers) (filter even [0..<length numbers]) ! j = numbers ! (filter even
[0..<length numbers] ! j)
  using nth-map[of j filter even [0..<length numbers] (! numbers) ]
  unfolding l1-def numbers1-def
  by (metis length-map)
  moreover have filter even [0..<llen] ! j =  $2 * j$  using
  filter-even-nth[of j llen  $2^{\lceil l \rceil}$ ] Suc(2) ass numbers1-def numbers1-even
  unfolding llen-def l1-def by fastforce
  moreover have  $n \text{ div } \text{llen} * (2 * j) = ((n \text{ div } (2^{\lceil l \rceil})) * j)$ 
  using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
  by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
  ultimately show map (! numbers) (filter even [0..<llen]) ! j *  $\omega^{(n \text{ div } 2^{\lceil l \rceil} * i * j)}$  =
  numbers ! (2 * j) *  $\omega^{(n \text{ div } \text{llen}) * i * (2 * j)}$ 
  unfolding llen-def l1-def l2-def by (metis (mono-tags, lifting) mult.assoc mult.left-commute)

```

```

qed
done
moreover have 004:
  (fntt2!i) * (ω(n div llen) * i) =
    (∑ j=0..(n div (2l))*i*j + (n div llen) * i)
  apply(rule trans[where s = (∑ j = 0..(n div 2l * i * j) * ω(n div llen * i))]])
  subgoal
    unfolding l2-def llen-def
  ω] using fntt2-by-index[of i] that sum-in[of - (ω(n div llen) * i) l2] comm-semiring-1-class.semiring-normalization

    unfolding ntt-gen-def
    using sum-rules apply presburger
    done
  apply (rule sum-rules(2))
  subgoal for j
  ω] using fntt2-by-index[of i] that sum-in[of - (ω(n div llen) * i) l2] comm-semiring-1-class.semiring-normalization

    unfolding ntt-gen-def
    apply auto
    done
  done
have 005: ... = (∑ j=0..(n div llen)*i*(2*j+1)))
  apply (rule sum-rules(2))
  subgoal for j unfolding numbers2-def
  apply(subst llen-def[symmetric])
  proof-
  assume ass: j < l2
  hence map (!) numbers (filter odd [0..l] Suc(2) ass numbers2-def numbers2-even
  unfolding l2-def numbers2-def llen-def by fastforce
  moreover have n div llen * (2 * j) = ((n div (2l)) * j)
    using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
  by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
  ultimately show
    map (!) numbers (filter odd [0..(n div 2l * i * j + n div llen * i)
      = numbers ! (2 * j + 1) * ω(n div llen * i * (2 * j + 1)) unfolding llen-def
    by (smt (z3) Groups.mult-ac(2) distrib-left mult.right-neutral mult-2 mult-cancel-left)
  qed
done
then show ?thesis
  using 000 001 002 003 004 005
  unfolding sum1-def llen-def l1-def l2-def
  using sum-splICE-other-way-round[of λ d. numbers ! d * ω(n div length numbers * i * d)
2l] Suc(2)
  unfolding ntt-gen-def

```

```

    by (smt (z3) Groups.mult-ac(2) numbers1-even numbers2-even power-Suc2)
  qed
  then show ?thesis
    by (metis 00 01 nth-equalityI)
  qed

```

We show equality for the indices in the second halves.

```

have after-half: map (ntt-gen numbers llen) [(llen div 2)..<llen] = sum2

```

```

proof-

```

```

  have 00:length (map (ntt-gen numbers llen) [(llen div 2)..<llen]) = length sum2
    unfolding sum2-def llen-def
    using Suc(2) map2-length map2-length fntt1-length fntt2-length by (simp add: mult-2)
  have 01:length sum2 = 2^l unfolding sum1-def
    using 00 Suc.premis(1) sum1-def llen-def by auto

```

Equality for every index.

```

have 02:(map (ntt-gen numbers llen) [(llen div 2)..<llen]) ! i = sum2 ! i
  if i < 2^l for i

```

```

proof-

```

```

  have 000:(map (ntt-gen numbers llen) [(llen div 2)..<llen]) ! i = ntt-gen numbers llen (2^l+i)
    unfolding llen-def by (simp add: Suc.premis(1) that)
  have 001:(map2 (λx y. x * ω ^ (n div llen * y)) fntt2 [0..<llen div 2]) ! i =
    fntt2 ! i * ω ^ (n div llen * i)
    using Suc(2) that by (simp add: fntt2-length llen-def)
  have 003: - fntt2 ! i * ω ^ (n div llen * i) =
    fntt2 ! i * ω ^ (n div llen * (i + llen div 2))
    using Suc(2) omega-div-exp-min1[of l] unfolding llen-def
    by (smt (z3) Suc.premis(2) mult.commute mult.left-commute mult-1s-ring-1(2) neq0-conv
nonzero-mult-div-cancel-left numeral-One pos2 power-Suc power-add power-mult)
  hence 004:sum2 ! i = (fntt1!i) - (fntt2!i) * (ω ^ ((n div llen) * i))
    unfolding sum2-def llen-def
    by (simp add: Suc.premis(1) fntt1-length fntt2-length that)
  have 005:(fntt1!i) =
    (∑ j=0..<l1. (numbers1 ! j) * ω ^ ((n div (2^l))*i*j))
    using fntt1-by-index that unfolding ntt-gen-def l1-def by simp
  have 006:... = (∑ j=0..<l1. (numbers ! (2*j)) * ω ^ ((n div llen)*i*(2*j)))
    apply (rule sum-rules(2))
  subgoal for j unfolding numbers1-def
    apply(subst llen-def[symmetric])
  proof-
    assume ass: j < l1
    hence map (!) numbers (filter even [0..<llen]) ! j = numbers ! (filter even [0..<llen]) ! j
      using nth-map unfolding llen-def l1-def numbers1-def by (metis length-map)
    moreover have filter even [0..<llen] ! j = 2 * j using
      filter-even-nth Suc(2) ass numbers1-def numbers1-even llen-def l1-def by fastforce
    moreover have n div llen * (2 * j) = ((n div (2^l)) * j)
      using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
      by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)

```

```

ultimately show
  map ((! numbers) (filter even [0..<l1]) ! j * ω ^ (n div 2 ^ l * i * j) =
    numbers ! (2 * j) * ω ^ (n div l1 * i * (2 * j))
  by (metis (mono-tags, lifting) mult.assoc mult.left-commute)
qed
done
have 007: ... = (∑ j=0..<l1. (numbers ! (2*j)) * ω ^ ((n div l1)*(2^l + i)*(2*j)))
apply (rule sum-rules(2))
subgoal for j
  using Suc(2) Suc(3) omega-div-exp-min1 [of l] llen-def l1-def numbers1-def
  apply (smt (verit, del-insts) add commute minus-power-mult-self mult-2 mult-minus1-right
power-add power-mult)
done
done
moreover have 008: (fntt2!i) * (ω ^ (n div l1) * i) =
  (∑ j=0..<l2. (numbers2 ! j) * ω ^ ((n div (2^l))*i*j + (n div l1) * i))
apply (rule trans[where s = (∑ j = 0..<l2. numbers2 ! j * ω ^ (n div 2 ^ l * i * j) * ω ^ (n
div l1 * i))])
subgoal
using fntt2-by-index[of i] that sum-in comm-semiring-1-class.semiring-normalization-rules(26)[of
ω]
  unfolding ntt-gen-def
  using sum-rules l2-def apply presburger
done
apply (rule sum-rules(2))
subgoal for j
using fntt2-by-index[of i] that sum-in comm-semiring-1-class.semiring-normalization-rules(26)[of
ω]
  unfolding ntt-gen-def
  apply auto
done
done
have 009: ... = (∑ j=0..<l2. (numbers ! (2*j+1) * ω ^ ((n div l1)*i*(2*j+1))))
apply (rule sum-rules(2))
subgoal for j unfolding numbers2-def
  apply (subst llen-def[symmetric])
  proof-
  assume ass: j < l2
  hence map ((! numbers) (filter odd [0..<l1]) ! j = numbers ! (filter odd [0..<l1] ! j)
    using nth-map llen-def l2-def numbers2-def by (metis length-map)
  moreover have filter odd [0..<l1] ! j = 2 * j + 1 using
    filter-odd-nth Suc(2) ass numbers2-def numbers2-even llen-def l2-def by fastforce
  moreover have n div l1 * (2 * j) = ((n div (2 ^ l)) * j)
    using Suc(2) two-pows-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
  by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
ultimately show
  map ((! numbers) (filter odd [0..<l1]) ! j * ω ^ (n div 2 ^ l * i * j + n div l1 * i)
    = numbers ! (2 * j + 1) * ω ^ (n div l1 * i * (2 * j + 1))

```

```

    by (smt (z3) Groups.mult-ac(2) distrib-left mult.right-neutral mult-2 mult-cancel-left)
  qed
done
have 010: (fntt2!i) * (ω(n div llen) * i) = (∑j=0..<l2. (numbers ! (2*j+1) * ω(n div llen)*i*(2*j+1))))
  using 008 009 by presburger
have 011: - (fntt2!i) * (ω(n div llen) * i) =
  (∑j=0..<l2. - (numbers ! (2*j+1) * ω(n div llen)*i*(2*j+1))))
  apply(rule neg-cong)
  apply(rule trans[of - fntt2 ! i * ω(n div llen) * i])
  subgoal by simp
  apply(rule trans[where s=(∑j=0..<l2. (numbers ! (2*j+1) * ω(n div llen)*i*(2*j+1)))])
  subgoal using 008 009 by simp
  apply(rule sym)
  using sum-neg-in[of - l2]
  apply simp
done
have 012: ... = (∑j=0..<l2. (numbers ! (2*j+1) * ω(n div llen)*(2l+i)*(2*j+1))))
  apply(rule sum-rules(2))
  subgoal for j
    using Suc(2) Suc(3) omega-div-exp-min1[of l] llen-def l2-def
    apply (smt (z3) add.commute exp-rule mult.assoc mult-minus1-right plus-1-eq-Suc power-add
power-minus1-odd power-mult)
  done
done
have 013:fntt1 ! i = (∑j = 0..<2l. numbers!(2*j) * ω(n div llen * (2l + i) * (2*j)))
  using 005 006 007 numbers1-even llen-def l1-def by auto
have 014: (∑j = 0..<2l. numbers ! (2*j + 1) * ω(n div llen * (2l + i) * (2*j + 1))) =
  - fntt2 ! i * ω(n div llen * i)
using trans[OF l2-def numbers2-even] sym[OF 012] sym[OF 011] by simp
have ntt-gen numbers llen (2l + i) = (fntt1!i) - (fntt2!i) * (ω(n div llen) * i)
  unfolding ntt-gen-def apply(subst Suc(2))
  using sum-splite[of λ d. numbers ! d * ω(n div llen * (2l+i) * d) 2l] sym[OF 013] 014
Suc(2) by simp
thus ?thesis using 000 sym[OF 001] 004 sum2-def by simp
qed
then show ?thesis
  by (metis 00 01 list-eq-iff-nth-eq)
qed
obtain x y xs where xyxs: numbers = x#y#xs using Suc(2)
  by (metis FNTT.cases add.left-neutral even-Suc even-add length-Cons list.size(3) mult-2 power-Suc
power-eq-0-iff zero-neq-numeral)
show ?case
  apply(subst xyxs)
  apply(subst FNTT.sims(3))
  apply(subst xyxs[symmetric])+
  unfolding Let-def
  using map-append[of ntt-gen numbers llen [0..<llen div 2] [llen div 2..<llen]] before-half after-half

```

```

unfolding llen-def sum1-def sum2-def fntt1-def fntt2-def NTT-gen-def
apply (metis (no-types, lifting) Suc.prem1 numbers1-def length-odd-filter mult-2 numbers2-def
numbers2-even power-Suc upt-add-eq-append zero-le-numeral zero-le-power)
done
qed

```

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*

using *FNTT-NTT-gen-eq NTT-gen-NTT-full-length assms n-two-pot* **by force**

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* = $(\sum_{j=0..<(\text{length numbers})}. (\text{numbers} ! j) * \mu^{\wedge}((n \text{ div } \text{degr}) * i * j))$

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

lemma *INTT-gen-INTT-full-length*:

assumes $length\ numbers = n$
shows $INTT\text{-}gen\ n\ numbers = INTT\ numbers$
unfolding $INTT\text{-}gen\text{-}def\ intt\text{-}gen\text{-}def\ INTT\text{-}def\ intt\text{-}def$
using $assms$ **by** $simp$

lemma $my\text{-}div\text{-}exp\text{-}min1$:

assumes $2^{\wedge}(Suc\ l) \leq n$

shows $(\mu\ \wedge(n\ div\ 2^{\wedge}(Suc\ l)))\ \wedge(2^{\wedge}l) = -1$

by ($metis\ assms\ divide\ minus1\ mult\ zero\ right\ mu\ properties(1)\ nonzero\ mult\ div\ cancel\ right\ omega\ div\ exp\ min1\ power\ one\ over\ zero\ neq\ one$)

lemma $my\text{-}n\text{-}2\text{-}min1$: $\mu\ \wedge(n\ div\ 2) = -1$

by ($metis\ divide\ minus1\ mult\ zero\ right\ mu\ properties(1)\ nonzero\ mult\ div\ cancel\ right\ omg\ n\ 2\ min1\ power\ one\ over\ zero\ neq\ one$)

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

theorem $IFNTT\text{-}INTT\text{-}gen\text{-}eq$:

$length\ numbers = 2^{\wedge}l \implies 2^{\wedge}l \leq n \implies IFNTT\ numbers = INTT\text{-}gen\ (length\ numbers)\ numbers$

proof ($induction\ l\ arbitrary:\ numbers$)

case 0

hence $local.\ IFNTT\ numbers = [numbers\ !\ 0]$

by ($metis\ IFNTT.\ simps(2)\ One\ nat\ def\ Suc\ length\ conv\ length\ 0\ conv\ nth\ Cons\ 0\ power\ 0$)

then show $?case$ **unfolding** $INTT\text{-}gen\text{-}def\ intt\text{-}gen\text{-}def$

using 0 **by** $simp$

next

case $(Suc\ l)$

We define some lists that are used during the recursive call.

define $numbers1$ **where** $numbers1 = [numbers!i . i <- (filter\ even\ [0..<length\ numbers])]$

define $numbers2$ **where** $numbers2 = [numbers!i . i <- (filter\ odd\ [0..<length\ numbers])]$

define $ifntt1$ **where** $ifntt1 = IFNTT\ numbers1$

define $ifntt2$ **where** $ifntt2 = IFNTT\ numbers2$

define $sum1$ **where**

$sum1 = map2\ (+)\ ifntt1\ (map2\ (\lambda\ x\ k.\ x * (\mu\ \wedge(n\ div\ (length\ numbers)) * k)))$
 $ifntt2\ [0..<((length\ numbers)\ div\ 2)]$

define $sum2$ **where**

$sum2 = map2\ (-)\ ifntt1\ (map2\ (\lambda\ x\ k.\ x * (\mu\ \wedge(n\ div\ (length\ numbers)) * k)))$
 $ifntt2\ [0..<((length\ numbers)\ div\ 2)]$

define $l1$ **where** $l1 = length\ numbers1$

define $l2$ **where** $l2 = length\ numbers2$

define $llen$ **where** $llen = length\ numbers$

Properties of those lists

have $numbers1\text{-}even$: $length\ numbers1 = 2^{\wedge}l$

using $numbers1\text{-}def\ length\ even\ filter\ Suc$ **by** $simp$

have $numbers2\text{-}even$: $length\ numbers2 = 2^{\wedge}l$

using $numbers2\text{-}def\ length\ odd\ filter\ Suc$ **by** $simp$

have $numbers1\text{-}ifntt$: $ifntt1 = INTT\text{-}gen\ (2^{\wedge}l)\ numbers1$

using $ifntt1\text{-}def\ Suc.IH[of\ numbers1]\ numbers1\text{-}even\ Suc(3)$ **by** $simp$

hence *ifntt1-by-index*: $\text{ifntt1} ! i = \text{intt-gen numbers1 } (2^l) i$ **if** $i < 2^l$ **for** i
unfolding *INTT-gen-def* **by** (*simp add: numbers1-even that*)
have *numbers2-ifntt*: $\text{ifntt2} = \text{INTT-gen } (2^l) \text{ numbers2}$
using *ifntt2-def Suc.IH[of numbers2] numbers2-even Suc(3)* **by** *simp*
hence *ifntt2-by-index*: $\text{ifntt2} ! i = \text{intt-gen numbers2 } (2^l) i$ **if** $i < 2^l$ **for** i
unfolding *INTT-gen-def* **by** (*simp add: numbers2-even that*)
have *ifntt1-length*: $\text{length ifntt1} = 2^l$ **unfolding** *numbers1-ifntt INTT-gen-def numbers1-def*
using *numbers1-def numbers1-even* **by** *force*
have *ifntt2-length*: $\text{length ifntt2} = 2^l$ **unfolding** *numbers2-ifntt INTT-gen-def numbers2-def*
using *numbers2-def numbers2-even* **by** *force*

Same proof structure as for the *FNTT* proof. ω s are just replaced by μ s.

have *before-half*: $\text{map } (\text{intt-gen numbers llen}) [0..<(\text{llen div } 2)] = \text{sum1}$
proof–

Length is important, since we want to use list lemmas later on.

have *00:length* ($\text{map } (\text{intt-gen numbers llen}) [0..<(\text{llen div } 2)]$) = length sum1
unfolding *sum1-def llen-def*
using *Suc(2) map2-length[of - ifntt2 [0..<length numbers div 2]]*
 $\text{map2-length[of (+) ifntt1 (map2 } (\lambda x y. x * \mu^{(n \text{ div } \text{length numbers} * y)}) \text{ifntt2 } [0..<\text{length numbers div } 2])]$
ifntt1-length ifntt2-length **by** (*simp add: mult-2*)
have *01:length sum1* = 2^l **unfolding** *sum1-def*
using *00 Suc.premis(1) sum1-def unfolding llen-def* **by** *auto*

We show equality by extensionality on indices.

have *02:(map (intt-gen numbers llen) [0..<(\text{llen div } 2)]) ! i* = $\text{sum1} ! i$
if $i < 2^l$ **for** i
proof–

First simplify this term.

have *000:(map (intt-gen numbers llen) [0..<(\text{llen div } 2)]) ! i* = $\text{intt-gen numbers llen } i$
using *00 01 that* **by** *auto*

Expand the definition of *sum1* and massage the result.

moreover **have** *001:sum1 ! i* = $(\text{ifntt1}!i) + (\text{ifntt2}!i) * (\mu^{(n \text{ div } \text{llen})} * i)$
unfolding *sum1-def* **using** *map2-index*
 $00 \ 01 \ \text{INTT-gen-def add.left-neutral diff-zero ifntt1-length length-map length-upt map2-map-map}$
 $\text{map-nth nth-upt numbers2-even numbers2-ifntt that llen-def}$ **by** *force*
moreover **have** *002:(ifntt1!i)* = $(\sum_{j=0..<l1}. (\text{numbers1} ! j) * \mu^{(n \text{ div } (2^l))*i*j})$
unfolding *l1-def*
using *ifntt1-by-index[of i] that unfolding intt-gen-def* **by** *simp*
have *003:... =* $(\sum_{j=0..<l1}. (\text{numbers} ! (2*j)) * \mu^{(n \text{ div } \text{llen})*i*(2*j)})$
apply (*rule sum-rules(2)*)
subgoal for j **unfolding** *numbers1-def*
apply(*subst llen-def[symmetric]*)
proof–
assume *ass: j < l1*
hence $\text{map } (! \text{ numbers}) (\text{filter even } [0..<\text{length numbers}]) ! j = \text{numbers} ! (\text{filter even } [0..<\text{length numbers}] ! j)$

```

using nth-map[of j filter even [0.. $length\ numbers$ ] (!) numbers ]
unfolding l1-def numbers1-def
by (metis length-map)
moreover have filter even [0.. $llen$ ] ! j = 2 * j using
filter-even-nth[of j llen 2l] Suc(2) ass numbers1-def numbers1-even
unfolding llen-def l1-def by fastforce
moreover have n div llen * (2 * j) = ((n div (2l)) * j)
using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
ultimately show map ((!) numbers) (filter even [0.. $llen$ ]) ! j *  $\mu^{(n\ div\ 2^l * i * j)}$  =
numbers ! (2 * j) *  $\mu^{(n\ div\ llen * i * (2 * j))}$ 
unfolding llen-def l1-def l2-def by (metis (mono-tags, lifting) mult.assoc mult.left-commute)
qed
done
moreover have 004:
( $ifntt2!i$ ) * ( $\mu^{(n\ div\ llen * i)}$ ) =
( $\sum_{j=0..<l2. (numbers2\ !\ j) * \mu^{(n\ div\ (2^l)*i*j + (n\ div\ llen * i)}}$ )
apply(rule trans[where s = ( $\sum_{j=0..<l2. numbers2\ !\ j * \mu^{(n\ div\ 2^l * i * j) * \mu^{(n\ div\ llen * i)}}$ )])
subgoal
unfolding l2-def llen-def
using ifntt2-by-index[of i] that sum-in[of - ( $\mu^{(n\ div\ llen * i)}$ ) l2] comm-semiring-1-class.semiring-normalization
 $\mu]$ 
unfolding intt-gen-def
using sum-rules apply presburger
done
apply (rule sum-rules(2))
subgoal for j
using ifntt2-by-index[of i] that sum-in[of - ( $\mu^{(n\ div\ llen * i)}$ ) l2] comm-semiring-1-class.semiring-normalization
 $\mu]$ 
unfolding intt-gen-def
apply auto
done
done
have 005: ... = ( $\sum_{j=0..<l2. (numbers\ !\ (2*j+1) * \mu^{(n\ div\ llen)*i*(2*j+1)}}$ )
apply (rule sum-rules(2))
subgoal for j unfolding numbers2-def
apply(subst llen-def[symmetric])
proof-
assume ass: j < l2
hence map ((!) numbers) (filter odd [0.. $llen$ ]) ! j = numbers ! (filter odd [0.. $llen$ ] ! j)
using nth-map unfolding l2-def numbers2-def llen-def by (metis length-map)
moreover have filter odd [0.. $llen$ ] ! j = 2 * j + 1 using
filter-odd-nth[of j length numbers 2l] Suc(2) ass numbers2-def numbers2-even
unfolding l2-def numbers2-def llen-def by fastforce
moreover have n div llen * (2 * j) = ((n div (2l)) * j)
using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff

```

```

power-inject-exp zero-neq-numeral)
  ultimately show
    map (!) numbers (filter odd [0..<llen]) ! j * μ ^ (n div 2 ^ l * i * j + n div llen * i)
      = numbers ! (2 * j + 1) * μ ^ (n div llen * i * (2 * j + 1)) unfolding llen-def
    by (smt (z3) Groups.mult-ac(2) distrib-left mult.right-neutral mult-2 mult-cancel-left)
  qed
done
then show ?thesis
  using 000 001 002 003 004 005
  unfolding sum1-def llen-def l1-def l2-def
  using sum-splice-other-way-round[of λ d. numbers ! d * μ ^ (n div length numbers * i * d)
2^l] Suc(2)
  unfolding intt-gen-def
  by (smt (z3) Groups.mult-ac(2) numbers1-even numbers2-even power-Suc2)
  qed
then show ?thesis
  by (metis 00 01 nth-equalityI)
  qed

```

We show index-wise equality for the second halves

```

have after-half: map (intt-gen numbers llen) [(llen div 2)..<llen] = sum2

```

proof–

```

have 00:length (map (intt-gen numbers llen) [(llen div 2)..<llen]) = length sum2
  unfolding sum2-def llen-def
  using Suc(2) map2-length map2-length ifntt1-length ifntt2-length by (simp add: mult-2)
have 01:length sum2 = 2^l unfolding sum1-def
  using 00 Suc.prem(1) sum1-def llen-def by auto

```

Equality for every index

```

have 02:(map (intt-gen numbers llen) [(llen div 2)..<llen]) ! i = sum2 ! i
  if i < 2^l for i

```

proof–

```

have 000:(map (intt-gen numbers llen) [(llen div 2)..<llen]) ! i = intt-gen numbers llen (2^l+i)
  unfolding llen-def by (simp add: Suc.prem(1) that)
have 001:(map2 (λx y. x * μ ^ (n div llen * y)) ifntt2 [0..<llen div 2]) ! i =
  ifntt2 ! i * μ ^ (n div llen * i)
  using Suc(2) that by (simp add: ifntt2-length llen-def)
have 003: – ifntt2 ! i * μ ^ (n div llen * i) = ifntt2 ! i * μ ^ (n div llen * (i+ llen div 2))
  using Suc(2) my-div-exp-min1[of l] unfolding llen-def
  by (smt (z3) Suc.prem(2) mult commute mult.left-commute mult-1s-ring-1(2) neq0-conv
nonzero-mult-div-cancel-left numeral-One pos2 power-Suc power-add power-mult)
hence 004:sum2 ! i = (ifntt1!i) – (ifntt2!i) * (μ ^ (n div llen) * i)
  unfolding sum2-def llen-def
  by (simp add: Suc.prem(1) ifntt1-length ifntt2-length that)
have 005:(ifntt1!i) =
  (∑ j=0..<l1. (numbers1 ! j) * μ ^ (n div (2^l)*i*j))
  using ifntt1-by-index that unfolding intt-gen-def l1-def by simp
have 006:... = (∑ j=0..<l1. (numbers ! (2*j)) * μ ^ (n div llen)*i*(2*j))
  apply (rule sum-rules(2))

```

```

subgoal for  $j$  unfolding numbers1-def
apply(subst llen-def[symmetric])
proof–
  assume ass:  $j < l1$ 
  hence map (!) numbers (filter even [0..<llen]) !  $j = \text{numbers} ! (\text{filter even } [0..<llen] ! j)$ 
    using nth-map unfolding llen-def l1-def numbers1-def by (metis length-map)
  moreover have filter even [0..<llen] !  $j = 2 * j$  using
    filter-even-nth Suc(2) ass numbers1-def numbers1-even llen-def l1-def by fastforce
  moreover have  $n \text{ div } llen * (2 * j) = ((n \text{ div } (2 \wedge l)) * j)$ 
    using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
    by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
  ultimately show
    map (!) numbers (filter even [0..<llen]) !  $j * \mu \wedge (n \text{ div } 2 \wedge l * i * j) =$ 
      numbers !  $(2 * j) * \mu \wedge (n \text{ div } llen * i * (2 * j))$ 
    by (metis (mono-tags, lifting) mult.assoc mult.left-commute)
  qed
done
have 007:  $\dots = (\sum j=0..<l1. (\text{numbers} ! (2*j)) * \mu \wedge ((n \text{ div } llen)*(2\wedge l + i)*(2*j)))$ 
apply (rule sum-rules(2))
subgoal for  $j$ 
  using Suc(2) Suc(3) my-div-exp-min1[of l] llen-def l1-def numbers1-def
  apply(smt (verit, del-insts) add commute minus-power-mult-self mult-2 mult-minus1-right
power-add power-mult)
  done
  done
  moreover have 008:  $(\text{ifntt2}!i) * (\mu \wedge (n \text{ div } llen) * i) =$ 
     $(\sum j=0..<l2. (\text{numbers2} ! j) * \mu \wedge ((n \text{ div } (2\wedge l))*i*j + (n \text{ div } llen) * i))$ 
  apply(rule trans[where s = (\sum j = 0..<l2. numbers2 ! j * \mu \wedge (n \text{ div } 2 \wedge l * i * j) * \mu \wedge (n
div llen * i))])
  subgoal
using ifntt2-by-index[of i] that sum-in comm-semiring-1-class.semiring-normalization-rules(26)[of
 $\mu]$ 
  unfolding intt-gen-def
  using sum-rules l2-def apply presburger
  done
  apply (rule sum-rules(2))
  subgoal for  $j$ 
using ifntt2-by-index[of i] that sum-in comm-semiring-1-class.semiring-normalization-rules(26)[of
 $\mu]$ 
  unfolding intt-gen-def
  apply auto
  done
  done
have 009:  $\dots = (\sum j=0..<l2. (\text{numbers} ! (2*j+1)) * \mu \wedge ((n \text{ div } llen)*i*(2*j+1)))$ 
apply (rule sum-rules(2))
subgoal for  $j$  unfolding numbers2-def
apply(subst llen-def[symmetric])
proof–

```

```

assume ass:  $j < l2$ 
hence map (! numbers) (filter odd [ $0..<l2$ ]) !  $j = \text{numbers} ! (\text{filter odd } [0..<l2] ! j)$ 
  using nth-map llen-def l2-def numbers2-def by (metis length-map)
moreover have filter odd [ $0..<l2$ ] !  $j = 2 * j + 1$  using
  filter-odd-nth Suc(2) ass numbers2-def numbers2-even llen-def l2-def by fastforce
moreover have  $n \text{ div } llen * (2 * j) = ((n \text{ div } (2 \wedge l)) * j)$ 
  using Suc(2) two-powrs-div[of l N] n-two-pot two-powr-div Suc(3) llen-def
  by (metis One-nat-def div-if mult.assoc nat-less-le not-less-eq numeral-2-eq-2 power-eq-0-iff
power-inject-exp zero-neq-numeral)
ultimately show
  map (! numbers) (filter odd [ $0..<l2$ ]) !  $j * \mu \wedge (n \text{ div } 2 \wedge l * i * j + n \text{ div } llen * i)$ 
    = numbers !  $(2 * j + 1) * \mu \wedge (n \text{ div } llen * i * (2 * j + 1))$ 
  by (smt (z3) Groups.mult-ac(2) distrib-left mult.right-neutral mult-2 mult-cancel-left)
qed
done
have 010: (ifntt2!  $i$ ) *  $(\mu \wedge (n \text{ div } llen) * i) = (\sum j=0..<l2. (\text{numbers} ! (2*j+1) * \mu \wedge (n \text{ div } llen)*i*(2*j+1)))$ 
  using 008 009 by presburger
have 011:  $-(\text{ifntt2}! i) * (\mu \wedge (n \text{ div } llen) * i) =$ 
   $(\sum j=0..<l2. -(\text{numbers} ! (2*j+1) * \mu \wedge (n \text{ div } llen)*i*(2*j+1)))$ 
  apply(rule neg-cong)
apply(rule trans[where  $s=(\sum j=0..<l2. (\text{numbers} ! (2*j+1) * \mu \wedge (n \text{ div } llen)*i*(2*j+1))$ )]])
subgoal using 008 009 by simp
apply(rule sym)
using sum-neg-in[of - l2]
apply simp
done
have 012:  $\dots = (\sum j=0..<l2. (\text{numbers} ! (2*j+1) * \mu \wedge (n \text{ div } llen)*(2\wedge l+i)*(2*j+1)))$ 
  apply(rule sum-rules(2))
subgoal for  $j$ 
  using Suc(2) Suc(3) my-div-exp-min1[of l] llen-def l2-def
  apply (smt (z3) add commute exp-rule mult.assoc mult-minus1-right plus-1-eq-Suc power-add
power-minus1-odd power-mult)
  done
done
have 013: ifntt1 !  $i = (\sum j = 0..<2 \wedge l. \text{numbers}!(2*j) * \mu \wedge (n \text{ div } llen * (2\wedge l + i) * (2*j)))$ 
  using 005 006 007 numbers1-even llen-def l1-def by auto
have 014:  $(\sum j = 0..<2 \wedge l. \text{numbers} ! (2*j + 1) * \mu \wedge (n \text{ div } llen * (2\wedge l + i) * (2*j + 1))) =$ 
   $-\text{ifntt2} ! i * \mu \wedge (n \text{ div } llen * i)$ 
using trans[OF l2-def numbers2-even] sym[OF 012] sym[OF 011] by simp
have intt-gen numbers llen  $(2 \wedge l + i) = (\text{ifntt1}! i) - (\text{ifntt2}! i) * (\mu \wedge (n \text{ div } llen) * i)$ 
  unfolding intt-gen-def
  apply(subst Suc(2))
using sum-splice[of  $\lambda d. \text{numbers} ! d * \mu \wedge (n \text{ div } llen * (2\wedge l+i) * d) 2\wedge l$ ] sym[OF 013] 014
Suc(2) by simp
thus ?thesis using 000 sym[OF 001] 004 sum2-def by simp
qed
then show ?thesis
  by (metis 00 01 list-eq-iff-nth-eq)

```

```

qed
obtain x y xs where xyxs: numbers = x#y#xs using Suc(2)
by (metis FNTT.cases add.left-neutral even-Suc even-add length-Cons list.size(3) mult-2 power-Suc
power-eq-0-iff zero-neq-numeral)
show ?case
apply(subst xyxs)
apply(subst IFNTT.simps(3))
apply(subst xyxs[symmetric])+
  unfolding Let-def
  using map-append[of innt-gen numbers llen [0..

```

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

theorem *IFNTT-correct*:

assumes *length numbers = n*

shows *IFNTT numbers = INTT numbers*

using *IFNTT-INTT-gen-eq INTT-gen-INTT-full-length assms n-two-pot* **by** *force*

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*

by (*simp add: FNTT-correct IFNTT-correct assms length-NTT ntt-correct*)

The other way round:

theorem *FNTT-inv-IFNTT*:

assumes *length numbers = n*

shows *FNTT (IFNTT numbers) = map ((* (of-int-mod-ring (int n))) numbers*

by (*simp add: FNTT-correct IFNTT-correct assms inv-ntt-correct length-INTT*)

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* \Rightarrow *'b list* \Rightarrow *'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])*
by (*induction g*) *auto*

lemma *map-filter-shift'*: *map f (filter odd [0..<Suc g]) = map (λ x. f (x+1)) (filter even [0..<g])*
by (*induction g*) *auto*

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

lemma *filter-comprehension-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

apply (*induction xs*)

apply *simp*

subgoal for *x xs*

apply *rule*

subgoal

apply (*subst evens-odds.simps*)

apply (*rule trans[of - map (!) (x # xs) (filter even [0..<Suc (length xs)])]*)

subgoal by *simp*

apply (*rule trans[OF map-filter-shift[of (!) (x # xs) length xs]]*)

apply *simp*

done

apply (*subst evens-odds.simps*)

apply (*rule trans[of - map (!) (x # xs) (filter odd [0..<Suc (length xs)])]*)

subgoal by *simp*

apply (*rule trans[OF map-filter-shift'[of (!) (x # xs) length xs]]*)

apply *simp*

done

done

For automated termination proof.

lemma [*simp*]: *length (evens-odds True vc) < Suc (length vc)*
length (evens-odds False vc) < Suc (length vc)

by (*metis filter-comprehension-evens-odds le-imp-less-Suc length-filter-le length-map map-nth*)**+**

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.. $(nn \text{ div } 2)$ ]);
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
apply(induction xs rule: FNTT'.induct)
subgoal by simp
subgoal by simp
apply(subst FNTT'.simps(3))
apply(subst FNTT'.simps(3))
subgoal for a b xs
  unfolding Let-def
  apply (metis filter-comprehension-evens-odds)
done
done

```

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 Teo::bool ⇒ 'c list ⇒ nat where
  Teo - [] = 1 |
  Teo True (x#xs) = (1 + Teo False xs) |
  Teo False (x#xs) = (1 + Teo True xs)

```

```

lemma Teo-linear: Teo b xs = length xs + 1
by (induction b xs rule: Teo.induct) auto

```

Time for length.

```

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

```

lemma *T-length-linear*: $T_{length} xs = length\ xs + 1$
by (*induction xs*) *auto*

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*: $T_{nth} xs\ i \leq length\ xs + 1$
by (*induction xs i rule: T_nth.induct*) *auto*

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*:

$c > 0 \implies$

$(\bigwedge x\ y.\ t\ x\ y \leq c) \implies T_{map2}\ t\ xs\ ys \leq \min (length\ xs)\ (length\ ys) * (c+1) + 1$

apply(*induction t xs ys rule: T_map2.induct*)

subgoal by *simp*

subgoal by *simp*

subgoal for $t\ x\ xs\ y\ ys$

apply(*subst T_map2.simps, subst length-Cons, subst length-Cons*)

using *min-add-distrib-right*[of 1]

by (*smt (z3) Suc-eq-plus1 add.assoc add.commute add-le-mono le-numeral-extra(4) min-def mult.commute mult-Suc-right*)

done

lemma *T-map-2-linear'*:

$c > 0 \implies$

$(\bigwedge x\ y.\ t\ x\ y = c) \implies T_{map2}\ t\ xs\ ys = \min (length\ xs)\ (length\ ys) * (c+1) + 1$

by(*induction t xs ys rule: T_map2.induct*) *simp+*

Time for append.

```

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

```

lemma *T-app-linear*: $T_{app} xs\ ys = length\ xs + 1$
by(*induction xs*) *auto*

Running Time of (optimized) *FNTT*.

```

fun  $T_{FNTT}::('a\ mod\ ring)\ list \Rightarrow nat$  where

```

$T_{FNTT} [] = 1$
 $T_{FNTT} [a] = 1$
 $T_{FNTT} \text{ nums} = (1 + T_{\text{length}} \text{ nums} + 3 +$

$(\text{let } nn = \text{length } \text{ nums};$
 $\text{ nums1} = \text{evens-odds True } \text{ nums};$
 $\text{ nums2} = \text{evens-odds False } \text{ nums}$
 in
 $T_{-eo} \text{ True } \text{ nums} + T_{-eo} \text{ False } \text{ nums} + 2 +$
 $(\text{let}$
 $\text{ fntt1} = \text{FNTT } \text{ nums1};$
 $\text{ fntt2} = \text{FNTT } \text{ nums2}$
 in
 $(T_{FNTT} \text{ nums1}) + (T_{FNTT} \text{ nums2}) +$
 $(\text{let}$
 $\text{ sum1} = \text{map2 } (+) \text{ fntt1 } (\text{map2 } (\lambda x k. x * (\omega \wedge (n \text{ div } nn) * k))) \text{ fntt2 } [0..<(nn \text{ div}$
 $2]));$
 $\text{ sum2} = \text{map2 } (-) \text{ fntt1 } (\text{map2 } (\lambda x k. x * (\omega \wedge (n \text{ div } nn) * k))) \text{ fntt2 } [0..<(nn \text{ div}$
 $2))$
 in
 $2 * T_{\text{map2}} (\lambda x y. 1) \text{ fntt2 } [0..<(nn \text{ div } 2)] +$
 $2 * T_{\text{map2}} (\lambda x y. 1) \text{ fntt1 } (\text{map2 } (\lambda x k. x * (\omega \wedge (n \text{ div } nn) * k))) \text{ fntt2 } [0..<(nn$
 $\text{ div } 2)] +$
 $T_{\text{app}} \text{ sum1 } \text{ sum2}))))$

lemma mono: $((f x)::\text{nat}) \leq f y \implies f y \leq f z \implies f x \leq f z$ **by simp**

lemma evens-odds-length:

$\text{length } (\text{evens-odds True } xs) = (\text{length } xs + 1) \text{ div } 2 \wedge$
 $\text{length } (\text{evens-odds False } xs) = (\text{length } xs) \text{ div } 2$

by(*induction xs*) *simp+*

Length preservation during *FNTT*.

lemma FNTT-length: $\text{length } \text{ numbers} = 2^\wedge l \implies \text{length } (\text{FNTT } \text{ numbers}) = \text{length } \text{ numbers}$

proof(*induction l arbitrary: numbers*)

case (*Suc l*)

define numbers1 where $\text{numbers1} = [\text{numbers!}i \mid i <- (\text{filter even } [0..<\text{length } \text{ numbers}])]$

define numbers2 where $\text{numbers2} = [\text{numbers!}i \mid i <- (\text{filter odd } [0..<\text{length } \text{ numbers}])]$

define fntt1 where $\text{fntt1} = \text{FNTT } \text{ numbers1}$

define fntt2 where $\text{fntt2} = \text{FNTT } \text{ numbers2}$

define presum where

$\text{presum} = (\text{map2 } (\lambda x k. x * (\omega \wedge (n \text{ div } (\text{length } \text{ numbers})) * k)))$
 $\text{ fntt2 } [0..<((\text{length } \text{ numbers}) \text{ div } 2)]$

define sum1 where

$\text{sum1} = \text{map2 } (+) \text{ fntt1 } \text{ presum}$

define sum2 where

$\text{sum2} = \text{map2 } (-) \text{ fntt1 } \text{ presum}$

have $\text{length } \text{ numbers1} = 2^\wedge l$

by (*metis Suc.prem numbers1-def diff-add-inverse2 length-even-filter mult-2 nonzero-mult-div-cancel-left*)

```

power-Suc zero-neq-numeral)
  hence length fntt1 = 2^l
  by (simp add: Suc.IH fntt1-def)
  hence length presum = 2^l unfolding presum-def
  using map2-length Suc.IH Suc.prem1 fntt2-def length-odd-filter numbers2-def by force
  hence length sum1 = 2^l
  by (simp add: ⟨length fntt1 = 2 ^ l⟩ sum1-def)
  have length numbers2 = 2^l
  by (metis Suc.prem1 numbers2-def length-odd-filter nonzero-mult-div-cancel-left power-Suc zero-neq-numeral)
  hence length fntt2 = 2^l
  by (simp add: Suc.IH fntt2-def)
  hence length sum2 = 2^l unfolding sum2-def
  using ⟨length sum1 = 2 ^ l⟩ sum1-def by force
  hence final:length (sum1@sum2) = 2^(Suc l)
  by (simp add: ⟨length sum1 = 2 ^ l⟩)
  obtain x y xs where xyxs-Def: numbers = x#y#xs
  by (metis ⟨length numbers2 = 2 ^ l⟩ evens-odds.elims filter-comprehension-evens-odds length-0-conv
  neq-Nil-conv numbers2-def power-eq-0-iff zero-neq-numeral)
  show ?case
  apply(subst xyxs-Def, subst FNTT.simps(3), subst xyxs-Def[symmetric])
  unfolding Let-def
  using final
  unfolding sum1-def sum2-def presum-def fntt1-def fntt2-def numbers1-def numbers2-def
  using Suc by (metis xyxs-Def)
qed (metis FNTT.simps(2) Suc-length-conv length-0-conv nat-power-eq-Suc-0-iff)

```

lemma add-cong: $(a1::nat) + a2+a3 + a4 = b \implies a1 + a2+ c + a3+a4 = c + b$
by simp

lemma add-mono: $a \leq (b::nat) \implies c \leq d \implies a + c \leq b + d$ **by** simp

lemma xyz: $Suc (Suc (length xs)) = 2 ^ l \implies length (x \# evens-odds True xs) = 2 ^ (l - 1)$
by (metis (no-types, lifting) Nat.add-0-right Suc-eq-plus1 div2-Suc-Suc div-mult-self2 evens-odds-length
length-Cons nat.distinct(1) numeral-2-eq-2 one-div-two-eq-zero plus-1-eq-Suc power-eq-if)

lemma zyx: $Suc (Suc (length xs)) = 2 ^ l \implies length (y \# evens-odds False xs) = 2 ^ (l - 1)$
by (smt (z3) One-nat-def Suc-pred diff-Suc-1 div2-Suc-Suc evens-odds-length le-numeral-extra(4)
length-Cons nat-less-le neq0-conv power-0 power-diff power-one-right zero-less-diff zero-neq-numeral)

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

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–

have $Suc (length (evens-odds b xs)) = length (evens-odds b (x\#y\#xs))$

by (metis (full-types) evens-odds.simps(2) evens-odds.simps(3) length-Cons)

have $length (x\#y\#xs) = 2^l$ **using** assms **by** simp

have $length (evens-odds b (x\#y\#xs)) = 2^{l-1}$

apply (*cases b*)
apply (*smt (z3) Suc-eq-plus1 Suc-pred (length (x # y # xs) = 2 ^ l) add.commute add-diff-cancel-left' assms filter-compehension-evens-odds gr0I le-add1 le-imp-less-Suc length-even-filter mult-2 nat-less-le power-diff power-eq-if power-one-right zero-neq-numeral*)
by (*smt (z3) One-nat-def Suc-inject (length (x # y # xs) = 2 ^ l) assms evens-odds-length le-zero-eq nat.distinct(1) neq0-conv not-less-eq-eq pos2 power-Suc0-right power-diff-power-eq power-eq-if*)
then show *?thesis*
by (*metis (Suc (length (evens-odds b xs)) = length (evens-odds b (x # y # xs)))*)
qed

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:

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

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

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:

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

The base case is trivially true.

theorem *tight-bound:*

assumes $T\text{-def}: \bigwedge \text{numbers } l. \text{length numbers} = 2^l \implies l > 0 \implies$

$$T \text{ numbers} = (2^l - 1) * 14 + l * 15 * 2^{l-1} + 2^l$$

$$\bigwedge \text{numbers } l. l = 0 \implies \text{length numbers} = 2^l \implies T \text{ numbers} = 1$$

shows $\text{length numbers} = 2^l \implies T_{FNTT} \text{ numbers} = T \text{ numbers}$

proof(*induction numbers arbitrary: l rule: T_{FNTT}.induct*)

case (*3 x y numbers*)

Some definitions for making term rewriting simpler.

```

define nn where nn = length (x # y # numbers)
define nums1 where nums1 = evens-odds True (x # y # numbers)
define nums2 where nums2 = evens-odds False (x # y # numbers)
define fnTT1 where fnTT1 = local.FNTT nums1
define fnTT2 where fnTT2 = local.FNTT nums2
define sum1 where sum1 = map2 (+) fnTT1 (map2 ( $\lambda x y. x * \omega ^{(n \text{ div } nn * y)})$ ) fnTT2 [0..nn

define sum2 where sum2 = map2 (-) fnTT1 (map2 ( $\lambda x y. x * \omega ^{(n \text{ div } nn * y)})$ ) fnTT2 [0..nn

Unfolding the running time function and combining it with the definitions above.



```

have TFNNT-simp: TFNNT (x # y # numbers) =
 1 + Tlength (x # y # numbers) + 3 +
 Teo True (x # y # numbers) + Teo False (x # y # numbers) + 2 +
 local.TFNNT nums1 + local.TFNNT nums2 +
 2 * Tmap2 ($\lambda x y. 1$) fnTT2 [0..nn div 2] +
 2 *
 Tmap2 ($\lambda x y. 1$) fnTT1 (map2 ($\lambda x y. x * \omega ^{(n \text{ div } nn * y)})$) fnTT2 [0..nn div 2]) +
 Tapp sum1 sum2
apply(subst TFNNT.simps(3))
unfolding Let-def unfolding sum2-def sum1-def fnTT1-def fnTT2-def nums1-def nums2-def nn-def
apply simp
done

```



Application of lemmas related to running times of auxiliary functions.



```

have length-nums1: length nums1 = (2::nat) $\lceil l-1$
unfolding nums1-def
using evens-odds-length[of x # y # numbers] 3(3) xyz by fastforce
have length-nums2: length nums2 = (2::nat) $\lceil l-1$
unfolding nums2-def
using evens-odds-length[of x # y # numbers] 3(3)
by (metis One-nat-def le-0-eq length-Cons lessI list.size(4) neq0-conv not-add-less2 not-less-eq-eq
pos2 power-Suc0-right power-diff-power-eq power-eq-if)
have length-simp: Tlength (x # y # numbers) = (2::nat) $\lceil l + 1$
using T-length-linear[of x#y#numbers] 3(3) by simp
have even-odd-simp: Teo b (x # y # numbers) = (2::nat) $\lceil l + 1$ for b
by (metis 3.prems T-eo-linear) +
have 02: (length fnTT2) = (length [0..nn div 2]) unfolding fnTT2-def
apply(subst FNNT-length[of - l-1])
unfolding nums2-def
using length-nums2 nums2-def apply fastforce
by (simp add: evens-odds-length nn-def)
have 03: (length fnTT1) = (length [0..nn div 2]) unfolding fnTT1-def
apply(subst FNNT-length[of - l-1])
unfolding nums1-def
using length-nums1 nums1-def apply fastforce
by (metis 02 FNNT-length fnTT2-def length-nums1 length-nums2 nums1-def)
have map21-simp: Tmap2 ($\lambda x y. 1$) fnTT2 [0..nn div 2] = (2::nat) $\lceil l + 1$
apply(subst T-map-2-linear[of 1])

```



46


```

```

subgoal by simp subgoal by simp
by (smt (z3) 02 3(3) FNTT-length div-less evens-odds-length fntt2-def length-nums2 lessI less-numeral-extra(3)
min.idem mult.commute nat-1-add-1 nums2-def plus-1-eq-Suc power-eq-if power-not-zero zero-power2)
have map22-simp:  $T_{map2} (\lambda x y. 1) fntt1 (map2 (\lambda x y. x * \omega ^ (n \text{ div } nn * y)) fntt2 [0..<nn \text{ div } 2]) =$ 
   $(2::nat)^\wedge l + 1$ 
apply(subst T-map-2-linear'[of 1])
subgoal by simp subgoal by simp apply simp
unfolding fntt1-def fntt2-def unfolding nn-def
apply(subst FNTT-length[of - l-1], (rule length-nums1)?, (rule length-nums2)?,
(subst length-nums1)?, (subst length-nums2)?, (subst 3(3))?)
apply (metis (no-types, lifting) 3(3) div-less evens-odds-length length-nums2 lessI min-def mult-2
nat-1-add-1 nums2-def plus-1-eq-Suc power-eq-if power-not-zero zero-neq-numeral)
done
have sum1-simp:  $length \text{ sum1} = 2^\wedge(l-1)$ 
unfolding sum1-def
apply(subst map2-length)+
apply(subst 02, subst 03)
unfolding nn-def using 3(3)
by (metis 02 FNTT-length fntt2-def length-nums2 min.idem nn-def)
have app-simp:  $T_{app} \text{ sum1 } \text{ sum2} = (2::nat)^\wedge(l-1) + 1$ 
by(subst T-app-linear, subst sum1-simp, simp)
let ?T1 =  $(2^\wedge(l-1) - 1) * 14 + (l-1) * 15 * 2^\wedge(l-1 - 1) + 2^\wedge(l-1)$ 

```

Induction hypotheses

```

have IH-pluged1: local.TFNTT nums1 = ?T1
apply(subst 3.IH(1)[of nn nums1 nums2 fntt1 fntt2 l-1,
OF nn-def nums1-def nums2-def fntt1-def fntt2-def length-nums1])
apply(cases l ≤ 1)
subgoal
apply(subst T-def(2)[of l-1])
subgoal by simp
apply(rule length-nums1)
apply simp
done
apply(subst T-def(1)[OF length-nums1])
subgoal by simp
subgoal by simp
done

have IH-pluged2: local.TFNTT nums2 = ?T1
apply(subst 3.IH(2)[of nn nums1 - fntt1 fntt2 l-1, OF nn-def nums1-def nums2-def fntt1-def
fntt2-def length-nums2 ])
apply(cases l ≤ 1)
subgoal
apply(subst T-def(2)[of l-1])
subgoal by simp
apply(rule length-nums2)
apply simp

```

```

done
apply(subst T-def(1)[OF length-nums2])
subgoal by simp
subgoal by simp
done

```

```

have T_FNNTT (x # y # numbers) =
  14 + (3 * 2 ^ l + (local.T_FNNTT nums1 +
    (local.T_FNNTT nums2 + (5 * 2^(l-1) + 4 * (2 ^ l div 2))))))
apply(subst TFNNT-simp, subst map21-simp, subst map22-simp, subst length-simp,
  subst app-simp, subst even-odd-simp, subst even-odd-simp)
apply(auto simp add: algebra-simps power-eq-if[of 2 l])
done

```

Proof that the term $T-def$ indeed fulfills the recursive properties, i.e. $t(2^l) = 2 \cdot t(2^{l-1}) + s$

```

also have ... = 14 + (3 * 2 ^ l + (?T1 + (?T1 + (5 * 2^(l-1) + 4 * (2 ^ l div 2))))))
apply(subst IH-pluged1, subst IH-pluged2)
by simp
also have ... = 14 + (6 * 2 ^ (l-1) +
  2*((2 ^ (l-1) - 1) * 14 + (l-1) * 15 * 2 ^ (l-1-1) + 2 ^ (l-1)) +
  (5 * 2 ^ (l-1) + 4 * (2 ^ l div 2)))
by (smt (verit) 3(3) add.assoc div-less evens-odds-length left-add-twice length-nums2 lessI mult.assoc
mult-2-right nat-1-add-1 numeral-Bit0 nums2-def plus-1-eq-Suc power-eq-if power-not-zero zero-neq-numeral)
also have ... = 14 + 15 * 2 ^ (l-1) +
  2*((2 ^ (l-1) - 1) * 14 + (l-1) * 15 * 2 ^ (l-1-1) + 2 ^ (l-1))
by (smt (z3) 3(3) add.assoc add commute calculation diff-diff-left distrib-left div2-Suc-Suc evens-odds-length
left-add-twice length-Cons length-nums2 mult.assoc mult.commute mult-2 mult-2-right numeral-Bit0
numeral-Bit1 numeral-plus-numeral nums2-def one-add-one)
also have ... = 14 + 15 * 2 ^ (l-1) +
  (2 ^ l - 2) * 14 + (l-1) * 15 * 2 ^ (l-1) + 2 ^ l
apply(cases l > 1)
apply (smt (verit, del-insts) add.assoc diff-is-0-eq distrib-left-numeral left-diff-distrib' less-imp-le-nat
mult.assoc mult-2 mult-2-right nat-1-add-1 not-le not-one-le-zero power-eq-if)
by (smt (z3) 3(3) add commute add.right-neutral cancel-comm-monoid-add-class.diff-cancel diff-add-inverse2
diff-is-0-eq div-less-dividend evens-odds-length length-nums2 mult-2 mult-eq-0-iff nat-1-add-1 not-le
nums2-def power-eq-if)
also have ... = 15 * 2 ^ (l-1) + (2 ^ l - 1) * 14 + (l-1) * 15 * 2 ^ (l-1) + 2 ^ l
by (smt (z3) 3(3) One-nat-def add commute combine-common-factor diff-add-inverse2 diff-diff-left
list.size(4) nat-1-add-1 nat-mult-1)
also have ... = (2 ^ l - 1) * 14 + l * 15 * 2 ^ (l-1) + 2 ^ l
apply(cases l > 0)
subgoal using group-cancel.add1 group-cancel.add2 less-numeral-extra(3) mult.assoc mult-eq-if by
auto[1]
using 3(3) by fastforce

```

By the previous proposition, we can conclude that T is indeed a suitable term for describing the running time

```

finally have T_FNNTT (x # y # numbers) = T (x # y # numbers)
using T-def(1)[of x#y#numbers l]

```


by (*metis 3.premis bits-1-div-2 diff-is-0-eq' evens-odds-length length-nums2 neq0-conv nums2-def power-0 zero-le-one zero-neq-one*)
thus *?case by simp*
qed (*auto simp add: assms*)

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

theorem *log-lin-time*:

assumes $length\ numbers = 2^l$

shows $T_{FNTT}\ numbers \leq 30 * l * length\ numbers + 1$

proof–

have *00*: $T_{FNTT}\ numbers = (2^l - 1) * 14 + l * 15 * 2^{(l-1)} + 2^l$

using *tight-bound*[*of* $\lambda\ xs.\ (length\ xs - 1) * 14 + (Discrete.log\ (length\ xs)) * 15 * 2^{(Discrete.log\ (length\ xs) - 1) + length\ xs\ numbers\ l}$]

assms by simp

have $l * 15 * 2^{(l-1)} \leq 15 * l * length\ numbers$ **using** *assms by simp*

moreover **have** $(2^l - 1) * 14 + 2^l \leq 15 * length\ numbers$

using *assms by linarith*

moreover **hence** $(2^l - 1) * 14 + 2^l \leq 15 * l * length\ numbers + 1$ **using** *assms*

apply(*cases l*)

subgoal **by** *simp*

by (*metis (no-types) add.commute le-add1 mult.assoc mult.commute*

mult-le-mono nat-mult-1 plus-1-eq-Suc trans-le-add2)

ultimately **have** $(2^l - 1) * 14 + l * 15 * 2^{(l-1)} + 2^l \leq 30 * l * length\ numbers + 1$

by *linarith*

then **show** *?thesis* **using** *00* **by** *simp*

qed

theorem *log-lin-time-explicitly*:

assumes $length\ numbers = 2^l$

shows $T_{FNTT}\ numbers \leq 30 * Discrete.log\ (length\ numbers) * length\ numbers + 1$

using *log-lin-time*[*of* $numbers\ l$] *assms by simp*

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](https://doi.org/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.