$(1-\delta)$-Correctness Proof of CRYSTALS-KYBER with Number Theoretic Transform

Katharina Kreuzer

April 14, 2026

Abstract

This article formalizes the specification and the algorithm of the cryptographic scheme CRYSTALS-KYBER with multiplication using the Number Theoretic Transform and verifies its $(1-\delta)$-correctness proof. CRYSTALS-KYBER is a key encapsulation mechanism in lattice-based post-quantum cryptography.

This entry formalizes the key generation, encryption and decryption algorithms and shows that the algorithm decodes correctly under a highly probable assumption ($(1-\delta)$-correctness). Moreover, the Number Theoretic Transform (NTT) in the case of Kyber and the convolution theorem thereon is formalized.

Contents

1 Introduction

CRYSTALS-KYBER is a cryptographic key encapsulation mechanism and one of the finalists of the third round in the NIST standardization project for post-quantum cryptography [1]. That is, even with feasible quantum computers, Kyber is thought to be hard to crack. It was introduced in [4] and its documentation can be found in [3].

Kyber is based on algebraic lattices and the module-LWE (Learning with Errors) Problem. Working over the quotient ring $R_q := \mathbb{Z}_q[x]/(x^{2^{n'}}+1)$ and vectors thereof, Kyber takes advantage of:

- properties both from polynomials and vectors
- cyclic properties of $\mathbb{Z}_q$ (where q is a prime)
- cyclic properties of the quotient ring
- the splitting of $x^{2^{n'}}+1$ as a reducible, cyclotomic polynomial over $\mathbb{Z}_q$

The algorithm in Kyber is quite simple:

1. Let Alice have a public key $A \in R_q^{k \times k}$ and a secret $s \in R_q^k$. Then she generates a second public key $t = Av + e$ using an error vector $e \in R_q^k$.
2. Bob (who wants to send a message to Alice) takes Alice's public keys A and t as well as his secret key $r \in R_q^k$, the message $m \in \{0,1\}^{256}$ and two random errors $e_1 \in R_q^k$ and $e_2 \in R_q$. He then computes the values $u = A^T r + e_1$ and $v = t^r + e_2 + \lceil q/2 \rfloor m$ and sends them to Alice.
3. Knowing her secret s, Alice can recover the message m from u and v By calculating $v - s^T u$. Any eavesdropper however cannot distinguish the encoded message from random samples.

The Number Theoretic Transform (NTT) is an analogue to the Discrete Fourier Transform in the setting of finite fields. As an extension to the AFP-entry "Number_Theoretic_Transform" [2], a special version of the NTT on R_q is formalized. The main difference is that the NTT used in Kyber has a "twiddle" factor, allowing for an easier implementation but requirng a $2n$-th root of unity instead of a n-th root of unity. Moreover, the structure of R_q is negacyclic, since $x^n \equiv -1 \mod x^n + 1$, instead of a cyclic convolution of the normal NTT. Additionally, the convolution theorem for the NTT in Kyber was formalized. It states $NTT(f \cdot g) = NTT(f) \cdot NTT(g)$.

In this work, we formalize the algorithms and verify the $(1 - \delta)$-correctness of Kyber and refine the algorithms to compute fast multiplication using the NTT.

```
theory Kyber_spec
imports Main "HOL-Computational_Algebra.Computational_Algebra"
  "HOL-Computational_Algebra.Polynomial_Factorial"
  "Berlekamp_Zassenhaus.Poly_Mod"
  "Berlekamp_Zassenhaus.Poly_Mod_Finite_Field"

begin
hide_type Matrix.vec
hide_const Matrix.vec_index
```

2 Type Class for Factorial Ring $\mathbb{Z}_q[x]/(x^n+1)$.

The Kyber algorithms work over the quotient ring $\mathbb{Z}_q[x]/(x^n+1)$ where q is a prime with $q \equiv 1 \mod 4$ and n is a power of 2. We encode this quotient ring as a type. In order to do so, we first look at the finite field $\mathbb{Z}_q$ implemented by `('a::prime_card) mod_ring`. Then we define polynomials using the constructor `poly`. For factoring out x^n+1, we define an equivalence relation on the polynomial ring $\mathbb{Z}_q[x]$ via the modulo operation with modulus x^n+1. Finally, we build the quotient of the equivalence relation using the construction `quotient_type`.

The module $\mathbb{Z}_q[x]/(x^n+1)$ was formalized with help from Manuel Eberl.

Modulo relation between two polynomials.

```
lemma of_int_mod_ring_eq_0_iff:
  "(of_int n :: ('n :: {finite, nontriv} mod_ring)) = 0 ⟷
    int (CARD('n)) dvd n"
  by transfer auto

lemma of_int_mod_ring_eq_of_int_iff:
  "(of_int n :: ('n :: {finite, nontriv} mod_ring)) = of_int m ⟷
    [n = m] (mod (int (CARD('n))))"
  by transfer (auto simp: cong_def)

definition mod_poly_rel :: "nat ⇒ int poly ⇒ int poly ⇒ bool" where
  "mod_poly_rel m p q ⟷
    (∀n. [poly.coeff p n = poly.coeff q n] (mod (int m)))"

lemma mod_poly_rel_altdef:
  "mod_poly_rel CARD('n :: nontriv) p q ⟷
    (of_int_poly p) = (of_int_poly q :: 'n mod_ring poly)"
  by (auto simp: poly_eq_iff mod_poly_rel_def
    of_int_mod_ring_eq_of_int_iff)

definition mod_poly_is_unit :: "nat ⇒ int poly ⇒ bool" where
  "mod_poly_is_unit m p ⟷ (∃r. mod_poly_rel m (p * r) 1)"

lemma mod_poly_is_unit_altdef:
```

```
  "mod_poly_is_unit CARD('n :: nontriv) p ⟷
    (of_int_poly p :: 'n mod_ring poly) dvd 1"
proof
  assume "mod_poly_is_unit CARD('n) p"
  thus "(of_int_poly p :: 'n mod_ring poly) dvd 1"
    by (auto simp: mod_poly_is_unit_def dvd_def mod_poly_rel_altdef
      of_int_poly_hom.hom_mult)
next
  assume "(of_int_poly p :: 'n mod_ring poly) dvd 1"
  then obtain q where q: "(of_int_poly p :: 'n mod_ring poly) * q = 1"
    by auto
  also have "q = of_int_poly (map_poly to_int_mod_ring q)"
    by (simp add: of_int_of_int_mod_ring poly_eqI)
  also have "of_int_poly p * ... =
      of_int_poly (p * map_poly to_int_mod_ring q)"
    by (simp add: of_int_poly_hom.hom_mult)
  finally show "mod_poly_is_unit CARD('n) p"
    by (auto simp: mod_poly_is_unit_def mod_poly_rel_altdef)
qed

definition mod_poly_irreducible :: "nat ⇒ int poly ⇒ bool" where
  "mod_poly_irreducible m Q ⟷
     ¬mod_poly_rel m Q 0 ∧
     ¬mod_poly_is_unit m Q ∧
        (∀a b. mod_poly_rel m Q (a * b) ⟶
               mod_poly_is_unit m a ∨ mod_poly_is_unit m b)"

lemma of_int_poly_to_int_poly: "of_int_poly (to_int_poly p) = p"
  by (simp add: of_int_of_int_mod_ring poly_eqI)

lemma mod_poly_irreducible_altdef:
  "mod_poly_irreducible CARD('n :: nontriv) p ⟷
    irreducible (of_int_poly p :: 'n mod_ring poly)"
proof
  assume "irreducible (of_int_poly p :: 'n mod_ring poly)"
  thus "mod_poly_irreducible CARD('n) p"
    by (auto simp: mod_poly_irreducible_def mod_poly_rel_altdef
    mod_poly_is_unit_altdef irreducible_def of_int_poly_hom.hom_mult)
next
  assume *: "mod_poly_irreducible CARD('n) p"
  show "irreducible (of_int_poly p :: 'n mod_ring poly)"
    unfolding irreducible_def
  proof (intro conjI impI allI)
    fix a b assume ab: "(of_int_poly p :: 'n mod_ring poly) = a * b"
    have "of_int_poly (map_poly to_int_mod_ring a *
      map_poly to_int_mod_ring b) =
      of_int_poly (map_poly to_int_mod_ring a) *
      (of_int_poly (map_poly to_int_mod_ring b) :: 'n mod_ring poly)"
      by (simp add: of_int_poly_hom.hom_mult)
```

```
    also have "... = a * b"
      by (simp add: of_int_poly_to_int_poly)
    also have "... = of_int_poly p"
      using ab by simp
    finally have "(of_int_poly p :: 'n mod_ring poly) =
      of_int_poly (to_int_poly a * to_int_poly b)" ..
    hence "of_int_poly (to_int_poly a) dvd (1 :: 'n mod_ring poly) ∨
           of_int_poly (to_int_poly b) dvd (1 :: 'n mod_ring poly)"
      using * unfolding mod_poly_irreducible_def mod_poly_rel_altdef
        mod_poly_is_unit_altdef by blast
    thus "(a dvd (1 :: 'n mod_ring poly)) ∨
      (b dvd (1 :: 'n mod_ring poly))"
      by (simp add: of_int_poly_to_int_poly)
  qed (use * in ‹auto simp: mod_poly_irreducible_def
    mod_poly_rel_altdef mod_poly_is_unit_altdef›)
qed
```

Type class for quotient ring $\mathbb{Z}_q[x]/(p)$. The polynomial p is represented as *qr_poly'* (an polynomial over the integers).

```
class qr_spec = prime_card +
  fixes qr_poly' :: "'a itself ⇒ int poly"
  assumes not_dvd_lead_coeff_qr_poly':
      "¬int CARD('a) dvd lead_coeff (qr_poly' TYPE('a))"
  and deg_qr'_pos : "degree (qr_poly' TYPE('a)) > 0"
```

qr_poly is the respective polynomial in $\mathbb{Z}_q[x]$.

```
definition qr_poly :: "'a :: qr_spec mod_ring poly" where
  "qr_poly = of_int_poly (qr_poly' TYPE('a))"
```

Functions to get the degree of the polynomials to be factored out.

```
definition (in qr_spec) deg_qr :: "'a itself ⇒ nat" where
  "deg_qr _ = degree (qr_poly' TYPE('a))"

lemma degree_qr_poly':
  "degree (qr_poly' TYPE('a :: qr_spec)) = deg_qr (TYPE('a))"
  by (simp add: deg_qr_def)

lemma degree_of_int_poly':
  assumes "of_int (lead_coeff p) ≠ (0 :: 'a :: ring_1)"
  shows "degree (of_int_poly p :: 'a poly) = degree p"
proof (intro antisym)
  show "degree (of_int_poly p) ≤ degree p"
    by (intro degree_le) (auto simp: coeff_eq_0)
  show "degree (of_int_poly p :: 'a poly) ≥ degree p"
    using assms by (intro le_degree) auto
qed

lemma degree_qr_poly:
  "degree (qr_poly :: 'a :: qr_spec mod_ring poly) = deg_qr (TYPE('a))"
```

```
  unfolding qr_poly_def
  using not_dvd_lead_coeff_qr_poly'[where ?'a = 'a]
  by (subst degree_of_int_poly')
     (auto simp: of_int_mod_ring_eq_0_iff degree_qr_poly')

lemma deg_qr_pos : "deg_qr TYPE('a :: qr_spec) > 0"
by (metis deg_qr'_pos degree_qr_poly')
```

The factor polynomial is non-zero.

```
lemma qr_poly_nz [simp]: "qr_poly ≠ 0"
  using deg_qr_pos[where ?'a = 'a] by (auto simp flip: degree_qr_poly)
```

Thus, when factoring out p, it has no effect on the neutral element 1.

```
lemma one_mod_qr_poly [simp]:
  "1 mod (qr_poly :: 'a :: qr_spec mod_ring poly) = 1"
proof -
  have "2 ^ 1 ≤ (2 ^ deg_qr TYPE('a) :: nat)"
    using deg_qr_pos[where ?'a = 'a]
    by (intro power_increasing) auto
  thus ?thesis by (metis degree_qr_poly deg_qr_pos degree_1 mod_poly_less)
qed
```

We define a modulo relation for polynomials modulo a polynomial $p =$`qr_poly`.

```
definition qr_rel :: "'a :: qr_spec mod_ring poly ⇒ 'a mod_ring poly ⇒
bool" where
  "qr_rel P Q ⟷ [P = Q] (mod qr_poly)"

lemma equivp_qr_rel: "equivp qr_rel"
  by (intro equivpI sympI reflpI transpI)
     (auto simp: qr_rel_def cong_sym intro: cong_trans)
```

Using this equivalence relation, we can define the quotient ring as a `quotient_type`.

```
quotient_type (overloaded) 'a qr = "'a :: qr_spec mod_ring poly" / qr_rel
  by (rule equivp_qr_rel)
```

Defining the conversion functions.

```
lift_definition to_qr :: "'a :: qr_spec mod_ring poly ⇒ 'a qr"
  is "λx. (x :: 'a mod_ring poly)" .

lift_definition of_qr :: "'a qr ⇒ 'a :: qr_spec mod_ring poly"
  is "λP::'a mod_ring poly. P mod qr_poly"
  by (simp add: qr_rel_def cong_def)
```

Simplification lemmas on conversion functions.

```
lemma of_qr_to_qr: "of_qr (to_qr (x)) = x mod qr_poly"
  apply (auto simp add: of_qr_def to_qr_def)
  by (metis of_qr.abs_eq of_qr.rep_eq)
```

```
lemma to_qr_of_qr: "to_qr (of_qr (x)) = x"
  apply (auto simp add: of_qr_def to_qr_def)
  by (metis (mono_tags, lifting) Quotient3_abs_rep Quotient3_qr
    Quotient3_rel cong_def qr_rel_def mod_mod_trivial)

lemma eq_to_qr: "x = y ⟹ to_qr x = to_qr y" by auto
```

Type class instantiation for *qr* (quotient ring).

```
instantiation qr :: (qr_spec) comm_ring_1
begin

lift_definition zero_qr :: "'a qr" is "0" .

lift_definition one_qr :: "'a qr" is "1" .

lift_definition plus_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(+)"
  unfolding qr_rel_def using cong_add by blast

lift_definition uminus_qr :: "'a qr ⇒ 'a qr"
  is "uminus"
  unfolding qr_rel_def  using cong_minus_minus_iff by blast

lift_definition minus_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(-)"
  unfolding qr_rel_def using cong_diff by blast

lift_definition times_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(*)"
  unfolding qr_rel_def using cong_mult by blast

instance
proof
  show "0 ≠ (1 :: 'a qr)"
    by transfer (simp add: qr_rel_def cong_def)
qed (transfer; simp add: qr_rel_def algebra_simps; fail)+

end

lemma of_qr_0 [simp]: "of_qr 0 = 0"
  and of_qr_1 [simp]: "of_qr 1 = 1"
  and of_qr_uminus [simp]: "of_qr (-p) = -of_qr p"
  and of_qr_add [simp]: "of_qr (p + q) = of_qr p + of_qr q"
  and of_qr_diff [simp]: "of_qr (p - q) = of_qr p - of_qr q"
  by (transfer; simp add: poly_mod_add_left poly_mod_diff_left; fail)+

lemma to_qr_0 [simp]: "to_qr 0 = 0"
  and to_qr_1 [simp]: "to_qr 1 = 1"
```

```
  and to_qr_uminus [simp]: "to_qr (-p) = -to_qr p"
  and to_qr_add [simp]: "to_qr (p + q) = to_qr p + to_qr q"
  and to_qr_diff [simp]: "to_qr (p - q) = to_qr p - to_qr q"
  and to_qr_mult [simp]: "to_qr (p * q) = to_qr p * to_qr q"
  by (transfer'; simp; fail)+

lemma to_qr_of_nat [simp]: "to_qr (of_nat n) = of_nat n"
  by (induction n) auto

lemma to_qr_of_int [simp]: "to_qr (of_int n) = of_int n"
  by (induction n) auto

lemma of_qr_of_nat [simp]: "of_qr (of_nat n) = of_nat n"
  by (induction n) auto

lemma of_qr_of_int [simp]: "of_qr (of_int n) = of_int n"
  by (induction n) auto

lemma of_qr_eq_0_iff [simp]: "of_qr p = 0 ⟷ p = 0"
  by transfer (simp add: qr_rel_def cong_def)

lemma to_qr_eq_0_iff:
  "to_qr p = 0 ⟷ qr_poly dvd p"
  by transfer (auto simp: qr_rel_def cong_def)
```

Some more lemmas that will probably be useful.

```
lemma to_qr_eq_iff [simp]:
  "to_qr P = (to_qr Q :: 'a :: qr_spec qr) ⟷ [P = Q] (mod qr_poly)"
  by transfer (auto simp: qr_rel_def)
```

Reduction modulo $x^n + 1$ is injective on polynomials of degree less than n in particular, this means that `card(QR(q^n)) = q^n`.

```
lemma inj_on_to_qr:
  "inj_on
     (to_qr :: 'a :: qr_spec mod_ring poly ⇒ 'a qr)
     {P. degree P < deg_qr TYPE('a)}"
  by (intro inj_onI) (auto simp: cong_def mod_poly_less
      simp flip: degree_qr_poly)
```

Characteristic of quotient ring is exactly q.

```
lemma of_int_qr_eq_0_iff [simp]:
  "of_int n = (0 :: 'a :: qr_spec qr) ⟷ int (CARD('a)) dvd n"
proof -
  have "of_int n = (0 :: 'a qr) ⟷ (of_int n :: 'a mod_ring poly) =
0"
    by (smt (z3) of_qr_eq_0_iff of_qr_of_int)
  also have "... ⟷ (of_int n :: 'a mod_ring) = 0"
    by (simp add: of_int_poly)
  also have "... ⟷ int (CARD('a)) dvd n"
```

```
    by (simp add: of_int_mod_ring_eq_0_iff)
  finally show ?thesis .
qed

lemma of_int_qr_eq_of_int_iff:
  "of_int n = (of_int m :: 'a :: qr_spec qr) ⟷
    [n = m] (mod (int (CARD('a))))"
  using of_int_qr_eq_0_iff[of "n - m", where ?'a = 'a]
  by (simp del: of_int_qr_eq_0_iff add: cong_iff_dvd_diff)

lemma of_nat_qr_eq_of_nat_iff:
  "of_nat n = (of_nat m :: 'a :: qr_spec qr) ⟷
    [n = m] (mod CARD('a))"
  using of_int_qr_eq_of_int_iff[of "int n" "int m"]
  by (simp add: cong_int_iff)

lemma of_nat_qr_eq_0_iff [simp]:
  "of_nat n = (0 :: 'a :: qr_spec qr) ⟷ CARD('a) dvd n"
  using of_int_qr_eq_0_iff[of "int n"] by simp
```

3 Specification of Kyber

```
definition to_module :: "int ⇒ 'a ::qr_spec qr" where
  "to_module x = to_qr (Poly [of_int_mod_ring x ::'a mod_ring])"
```

Properties in the ring *'a qr*. A good representative has degree up to n.

```
lemma deg_mod_qr_poly:
  assumes "degree x < deg_qr TYPE('a::qr_spec)"
  shows "x mod (qr_poly :: 'a mod_ring poly) = x"
using mod_poly_less[of x qr_poly] unfolding deg_qr_def
by (metis assms degree_qr_poly)

lemma of_qr_to_qr':
  assumes "degree x < deg_qr TYPE('a::qr_spec)"
  shows "of_qr (to_qr x) = (x ::'a mod_ring poly)"
using deg_mod_qr_poly[OF assms] of_qr_to_qr[of x] by simp

lemma deg_of_qr:
  "degree (of_qr (x ::'a qr)) < deg_qr TYPE('a::qr_spec)"
by (metis deg_qr_pos degree_0 degree_qr_poly degree_mod_less'
  qr_poly_nz of_qr.rep_eq)

lemma to_qr_smult_to_module:
  "to_qr (Polynomial.smult a p) = (to_qr (Poly [a])) * (to_qr p)"
by (metis Poly.simps(1) Poly.simps(2) mult.left_neutral
  mult_smult_left smult_one to_qr_mult)
```

```
lemma of_qr_to_qr_smult:
  "of_qr (to_qr (Polynomial.smult a p)) =
  Polynomial.smult a (of_qr (to_qr p))"
by (simp add: mod_smult_left of_qr_to_qr)
```

The following locale comprehends all variables used in crypto schemes over R_q like Kyber and Dilithium.

```
locale module_spec =
fixes "type_a" :: "('a :: qr_spec) itself"
  and "type_k" :: "('k ::finite) itself"
  and n q::int and k n'::nat
assumes
n_powr_2: "n = 2 ^ n'" and
n'_gr_0: "n' > 0" and
q_gr_two: "q > 2" and
q_prime : "prime q" and
CARD_a: "int (CARD('a :: qr_spec)) = q" and
CARD_k: "int (CARD('k :: finite)) = k" and
qr_poly'_eq: "qr_poly' TYPE('a) = Polynomial.monom 1 (nat n) + 1"

begin
```

Some properties of the modulus q.

```
lemma q_nonzero: "q ≠ 0"
using module_spec_axioms module_spec_def  by (smt (z3))

lemma q_gt_zero: "q>0"
using module_spec_axioms module_spec_def by (smt (z3))

lemma q_gt_two: "q>2"
using module_spec_axioms module_spec_def by (smt (z3))

lemma q_odd: "odd q"
using module_spec_axioms module_spec_def prime_odd_int by blast

lemma nat_q: "nat q = q"
using q_gt_zero by force
```

Some properties of the degree n.

```
lemma n_gt_1: "n > 1"
using module_spec_axioms module_spec_def
  by (simp add: n'_gr_0 n_powr_2)

lemma n_nonzero: "n ≠ 0"
using n_gt_1 by auto

lemma n_gt_zero: "n>0"
using n_gt_1 by auto
```

```
lemma nat_n: "nat n = n"
using n_gt_zero by force

lemma deg_qr_n:
  "deg_qr TYPE('a) = n"
unfolding deg_qr_def using qr_poly'_eq n_gt_1
by (simp add: degree_add_eq_left degree_monom_eq)

end
```

We now define a locale for the specification parameters of Kyber as in [4]. The specifications use the parameters:

$$\begin{aligned} n &= 256 = 2^{n'} \\ n' &= 8 \\ q &= 7681 \text{ or } 3329 \\ k &= 3 \end{aligned}$$

Additionally, we need that q is a prime with the property $q \equiv 1 \mod 4$.

```
locale kyber_spec = module_spec "TYPE ('a ::qr_spec)" "TYPE ('k::finite)"
+
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
assumes q_mod_4: "q mod 4 = 1"
begin
end

end
theory Mod_Plus_Minus

imports Kyber_spec

begin

lemma odd_half_floor:
  ‹⌊real_of_int x / 2⌋ = (x - 1) div 2› if ‹odd x›
  using that by (rule oddE) simp
```

4 Re-centered Modulo Operation

To define the compress and decompress functions, we need some special form of modulo. It returns the representation of the equivalence class in *(-q div 2, q div 2]*. Using these representatives, we ensure that the norm of the representative is as small as possible.

```
definition mod_plus_minus :: "int ⇒ int ⇒ int"
  (infixl ‹mod+-› 70) where
"m mod+- b =
```

```
  (if m mod b > ⌊b/2⌋ then m mod b - b else m mod b)"
```

Range of the (re-centered) modulo operation

```
lemma mod_range: "b>0 ⟹ (a::int) mod (b::int) ∈ {0..b-1}"
using range_mod by auto

lemma mod_rangeE:
  assumes "(a::int)∈{0..<b}"
  shows "a = a mod b"
using assms by auto


lemma half_mod_odd:
  assumes "b > 0" "odd b" "⌊real_of_int b / 2⌋ < y mod b"
  shows "- ⌊real_of_int b / 2⌋ ≤ y mod b - b"
    "y mod b - b ≤ ⌊real_of_int b / 2⌋"
proof -
  from odd_half_floor [of b]
  show "- ⌊real_of_int b / 2⌋ ≤ y mod b - b"
    using assms by linarith
  then have "y mod b ≤ b + ⌊real_of_int b / 2⌋"
    by (smt (verit) ‹b > 0› pos_mod_bound)
  then show "y mod b - b ≤ ⌊real_of_int b / 2⌋"
    by auto
qed

lemma half_mod:
assumes "b>0"
shows "- ⌊real_of_int b / 2⌋ ≤ y mod b"
using assms
by (smt (verit, best) floor_less_zero half_gt_zero mod_int_pos_iff of_int_pos)

lemma mod_plus_minus_range_odd:
  assumes "b>0" "odd b"
  shows "y mod+- b ∈ {-⌊b/2⌋..⌊b/2⌋}"
unfolding mod_plus_minus_def by (auto simp add: half_mod_odd[OF assms]
half_mod[OF assms(1)])

lemma odd_smaller_b:
  assumes "odd b"
  shows "⌊ real_of_int b / 2 ⌋ + ⌊ real_of_int b / 2 ⌋ < b"
using assms
by (smt (z3) floor_divide_of_int_eq odd_two_times_div_two_succ
  of_int_hom.hom_add of_int_hom.hom_one)


lemma mod_plus_minus_rangeE_neg:
  assumes "y ∈ {-⌊real_of_int b/2⌋..⌊real_of_int b/2⌋}"
          "odd b" "b > 0"
```

```
          "⌊real_of_int b / 2⌋ < y mod b"
  shows "y = y mod b - b"
proof -
  have "y ∈ {-⌊real_of_int b/2⌋..<0}" using assms
  by (meson atLeastAtMost_iff atLeastLessThan_iff linorder_not_le order_trans
zmod_le_nonneg_dividend)
  then have "y ∈ {-b..<0}" using assms(2-3)
  by (metis atLeastLessThan_iff floor_divide_of_int_eq int_div_less_self
linorder_linear
    linorder_not_le neg_le_iff_le numeral_code(1) numeral_le_iff of_int_numeral
order_trans
    semiring_norm(69))
  then have "y mod b = y + b"
  by (smt (verit) atLeastLessThan_iff mod_add_self2 mod_pos_pos_trivial)
  then show ?thesis by auto
qed

lemma mod_plus_minus_rangeE_pos:
  assumes "y ∈ {-⌊real_of_int b/2⌋..⌊real_of_int b/2⌋}"
          "odd b" "b > 0"
          "⌊real_of_int b / 2⌋ ≥ y mod b"
  shows "y = y mod b"
proof -
  have "y ∈ {0..⌊real_of_int b/2⌋}"
  proof (rule ccontr)
    assume "y ∉ {0..⌊real_of_int b / 2⌋} "
    then have "y ∈ {-⌊real_of_int b/2⌋..<0}" using assms(1) by auto
    then have "y ∈ {-b..<0}" using assms(2-3)
    by (metis atLeastLessThan_iff floor_divide_of_int_eq int_div_less_self
linorder_linear
      linorder_not_le neg_le_iff_le numeral_code(1) numeral_le_iff of_int_numeral
order_trans
      semiring_norm(69))
    then have "y mod b = y + b"
      by (smt (verit) atLeastLessThan_iff mod_add_self2 mod_pos_pos_trivial)
    then have "y mod b ≥ b - ⌊real_of_int b/2⌋" using assms(1) by auto
    then have "y mod b > ⌊real_of_int b/2⌋"
      using assms(2) odd_smaller_b by fastforce
    then show False using assms(4) by auto
  qed
  then have "y ∈ {0..<b}" using assms(2-3)
  by (metis atLeastAtMost_iff atLeastLessThan_empty atLeastLessThan_iff
floor_divide_of_int_eq
    int_div_less_self linorder_not_le numeral_One numeral_less_iff of_int_numeral
semiring_norm(76))
  then show ?thesis by auto
qed

lemma mod_plus_minus_rangeE:
```

```
  assumes "y ∈ {-⌊real_of_int b/2⌋..⌊real_of_int b/2⌋}"
          "odd b" "b > 0"
  shows "y = y mod+- b"
unfolding mod_plus_minus_def
using mod_plus_minus_rangeE_pos[OF assms] mod_plus_minus_rangeE_neg[OF
assms]
by auto
```

Image of 0.

```
lemma mod_plus_minus_zero:
  assumes "x mod+- b = 0"
  shows "x mod b = 0"
using assms unfolding mod_plus_minus_def
by (metis eq_iff_diff_eq_0 mod_mod_trivial mod_self)

lemma mod_plus_minus_zero':
  assumes "b>0" "odd b"
  shows "0 mod+- b = (0::int)"
using assms(1) mod_plus_minus_def by force
```

mod+- with negative values.

```
lemma neg_mod_plus_minus:
  assumes "odd b"
          "b>0"
  shows "(- x) mod+- b = - (x mod+- b)"
proof -
  obtain k :: int where k_def: "(-x) mod+- b = (-x)+ k* b"
  using mod_plus_minus_def
  proof -
    assume a1: "⋀k. - x mod+- b = - x + k * b ⟹ thesis"
    have "∃i. i mod b + - (x + i) = - x mod+- b"
    by (smt (verit, del_insts) mod_add_self1 mod_plus_minus_def)
    then show ?thesis
      using a1 by (metis (no_types) diff_add_cancel diff_diff_add
      diff_minus_eq_add minus_diff_eq minus_mult_div_eq_mod
      mult.commute mult_minus_left)
  qed
  then have "(-x) mod+- b = -(x - k*b)" using k_def by auto
  also have "... = - ((x-k*b) mod+- b)"
  proof -
    have range_xkb:"x - k * b ∈
      {- ⌊real_of_int b / 2⌋..⌊real_of_int b / 2⌋}"
      using k_def mod_plus_minus_range_odd[OF assms(2) assms(1)]
      by (smt (verit, ccfv_SIG) atLeastAtMost_iff)
    have "x - k*b = (x - k*b) mod+- b"
      using mod_plus_minus_rangeE[OF range_xkb assms] by auto
    then show ?thesis by auto
  qed
  also have "-((x - k*b) mod+- b) = -(x mod+- b)"
```

```
    unfolding mod_plus_minus_def
    by (smt (verit, best) mod_mult_self1)
  finally show ?thesis by auto
qed
```

Representative with `mod+-`

```
lemma mod_plus_minus_rep_ex:
"∃k. x = k*b + x mod+- b"
unfolding mod_plus_minus_def
by (split if_splits)(metis add.right_neutral add_diff_eq div_mod_decomp_int

  eq_iff_diff_eq_0 mod_add_self2)


lemma mod_plus_minus_rep:
  obtains k where "x = k*b + x mod+- b"
using mod_plus_minus_rep_ex by auto
```

Multiplication in `mod+-`

```
lemma mod_plus_minus_mult:
  "s*x mod+- q = (s mod+- q) * (x mod+- q) mod+- q"
unfolding mod_plus_minus_def
by (smt (verit, ccfv_threshold) minus_mod_self2 mod_mult_left_eq mod_mult_right_eq)
end
theory Abs_Qr

imports Mod_Plus_Minus
        Kyber_spec

begin
```

Auxiliary lemmas

```
lemma finite_range_plus:
  assumes "finite (range f)"
          "finite (range g)"
  shows "finite (range (λx. f x + g x))"
proof -
  have subs: "range (λx. (f x, g x)) ⊆ range f × range g" by auto
  have cart: "finite (range f × range g)" using assms by auto
  have finite: "finite (range (λx. (f x, g x)))"
    using rev_finite_subset[OF cart subs] .
  have "range (λx. f x + g x) = (λ(a,b). a+b) ‘ range (λx. (f x, g x))"
    using range_composition[of "(λ(a,b). a+b)" "(λx. (f x, g x))"]
    by auto
  then show ?thesis
    using finite finite_image_set[where f = "(λ(a,b). a+b)"]
    by auto
qed
```

```
lemma all_impl_Max:
  assumes "∀x. f x ≥ (a::int)"
          "finite (range f)"
  shows "(MAX x. f x) ≥ a"
by (simp add: Max_ge_iff assms(1) assms(2))

lemma Max_mono':
  assumes "∀x. f x ≤ g x"
          "finite (range f)"
          "finite (range g)"
  shows "(MAX x. f x) ≤ (MAX x. g x)"
using assms
by (metis (no_types, lifting) Max_ge_iff Max_in UNIV_not_empty
  image_is_empty rangeE rangeI)

lemma Max_mono_plus:
  assumes "finite (range (f::_⇒_::ordered_ab_semigroup_add))"
          "finite (range g)"
  shows "(MAX x. f x + g x) ≤ (MAX x. f x) + (MAX x. g x)"
proof -
  obtain xmax where xmax_def: "f xmax + g xmax = (MAX x. f x + g x)"

    using finite_range_plus[OF assms] Max_in by fastforce
  have "(MAX x. f x + g x) = f xmax + g xmax" using xmax_def by auto
  also have "... ≤ (MAX x. f x) + g xmax"
    using Max_ge[OF assms(1), of "f xmax"]
    by (auto simp add: add_right_mono[of "f xmax"])
  also have "... ≤ (MAX x. f x) + (MAX x. g x)"
    using Max_ge[OF assms(2), of "g xmax"]
    by (auto simp add: add_left_mono[of "g xmax"])
  finally show ?thesis by auto
qed
```

Lemmas for porting to *qr*.

```
lemma of_qr_mult:
  "of_qr (a * b) = of_qr a * of_qr b mod qr_poly"
by (metis of_qr_to_qr to_qr_mult to_qr_of_qr)

lemma of_qr_scale:
  "of_qr (to_module s * b) =
  Polynomial.smult (of_int_mod_ring s) (of_qr b)"
unfolding to_module_def
  by (auto simp add: of_qr_mult[of "to_qr [:of_int_mod_ring s:]" "b"]

  of_qr_to_qr) (simp add: mod_mult_left_eq mod_smult_left of_qr.rep_eq)

lemma to_module_mult:
  "poly.coeff (of_qr (to_module s * a)) x1 =
   of_int_mod_ring (s) * poly.coeff (of_qr a) x1"
```

```
using of_qr_scale[of s a] by simp
```

Lemmas on *round* and *floor*.

```
lemma odd_round_up:
  assumes "odd x"
  shows "round (real_of_int x / 2) = (x + 1) div 2"
proof -
  from assms have ‹odd (x + 2)›
    by simp
  then have ‹⌊real_of_int (x + 2) / 2⌋ = (x + 2 - 1) div 2›
    by (rule odd_half_floor)
  from this [symmetric] show ?thesis
    by (simp add: round_def ac_simps) linarith
qed

lemma floor_unique:
assumes "real_of_int a ≤ x" "x < a+1"
shows "floor x = a"
  using assms(1) assms(2) by linarith

lemma same_floor:
assumes "real_of_int a ≤ x" "real_of_int a ≤ y"
  "x < a+1" "y < a+1"
shows "floor x = floor y"
using assms floor_unique  by presburger

lemma one_mod_four_round:
assumes "x mod 4 = 1"
shows "round (real_of_int x / 4) = (x-1) div 4"
proof -
  have leq: "(x-1) div 4 ≤ real_of_int x / 4  + 1 / 2"
    using assms by linarith
  have gr: "real_of_int x / 4  + 1 / 2 < (x-1) div 4 + 1"
  proof -
    have "x+2 < 4 * ((x-1) div 4 + 1)"
    proof -
      have *:  "(x-1) div 4 + 1 = (x+3) div 4" by auto
      have "4 dvd x + 3" using assms by presburger
      then have "4 * ((x+3) div 4) = x+3"
        by (subst dvd_imp_mult_div_cancel_left, auto)
      then show ?thesis unfolding * by auto
    qed
    then show ?thesis by auto
  qed
  show "round (real_of_int x / 4) = (x-1) div 4"
    using floor_unique[OF leq gr] unfolding round_def by auto
qed
```

5 Re-centered "Norm" Function

```
context module_spec
begin
```

We want to show that `abs_infty_q` is a function induced by the Euclidean norm on the `mod_ring` using a re-centered representative via `mod+-`.

`abs_infty_poly` is the induced norm by `abs_infty_q` on polynomials over the polynomial ring over the `mod_ring`.

Unfortunately this is not a norm per se, as the homogeneity only holds in inequality, not equality. Still, it fulfils its purpose, since we only need the triangular inequality.

```
definition abs_infty_q :: "('a mod_ring) ⇒ int" where
  "abs_infty_q p = abs ((to_int_mod_ring p) mod+- q)"

definition abs_infty_poly :: "'a qr ⇒ int" where
  "abs_infty_poly p = Max (range (abs_infty_q ∘ poly.coeff (of_qr p)))"
```

Helping lemmas and properties of `Max`, `range` and `finite`.

```
lemma to_int_mod_ring_range:
  "range (to_int_mod_ring :: 'a mod_ring ⇒ int) = {0 ..< q}"
using CARD_a by (simp add: range_to_int_mod_ring)

lemma finite_Max:
  "finite (range (λxa. abs_infty_q (poly.coeff (of_qr x) xa)))"
proof -
  have finite_range: "finite (range (λxa. (poly.coeff (of_qr x) xa)))"

  using MOST_coeff_eq_0[of "of_qr x"] by auto
  have "range (λxa. |to_int_mod_ring (poly.coeff (of_qr x) xa) mod+- q|)
    = (λz. |to_int_mod_ring z mod+- q|) ` range (poly.coeff (of_qr x))"
  using range_composition[of "(λz. abs (to_int_mod_ring z mod+- q))"
    "poly.coeff (of_qr x)"] by auto
  then show ?thesis
    using finite_range finite_image_set[where
      f = "(λz. abs (to_int_mod_ring z) mod+- q)"]
    by (auto simp add: abs_infty_q_def)
qed

lemma finite_Max_scale:
  "finite (range (λxa. abs_infty_q (of_int_mod_ring s *
    poly.coeff (of_qr x) xa)))"
proof -
  have "of_int_mod_ring s * poly.coeff (of_qr x) xa =
    poly.coeff (of_qr (to_module s * x)) xa" for xa
  by (metis coeff_smult of_qr_to_qr_smult to_qr_of_qr
    to_qr_smult_to_module to_module_def)
  then show ?thesis
```

```
  using finite_Max by presburger
qed

lemma finite_Max_sum:
  "finite (range (λxa. abs_infty_q
     (poly.coeff (of_qr x) xa + poly.coeff (of_qr y) xa)))"
proof -
  have finite_range: "finite (range (λxa. (poly.coeff (of_qr x) xa +
    poly.coeff (of_qr y) xa)))"
  using MOST_coeff_eq_0[of "of_qr x"] by auto
  have "range (λxa. |to_int_mod_ring (poly.coeff (of_qr x) xa +
    poly.coeff (of_qr y) xa) mod+- q|) =
    (λz. |to_int_mod_ring z mod+- q|) `
    range (λxa. poly.coeff (of_qr x) xa + poly.coeff (of_qr y) xa)"
  using range_composition[of "(λz. abs (to_int_mod_ring z mod+- q))"
    "(λxa. poly.coeff (of_qr x) xa + poly.coeff (of_qr y) xa)"]
  by auto
  then show ?thesis
    using finite_range finite_image_set[where
      f = "(λz. abs (to_int_mod_ring z) mod+- q)" ]
    by (auto simp add: abs_infty_q_def)
qed


lemma finite_Max_sum':
  "finite (range
     (λxa. abs_infty_q (poly.coeff (of_qr x) xa) +
      abs_infty_q (poly.coeff (of_qr y) xa)))"
proof -
  have finite_range_x:
    "finite (range (λxa. abs_infty_q (poly.coeff (of_qr x) xa)))"
    using finite_Max[of x] by auto
  have finite_range_y:
    "finite (range (λxa. abs_infty_q (poly.coeff (of_qr y) xa)))"
    using finite_Max[of y] by auto
  show ?thesis
    using finite_range_plus[OF finite_range_x finite_range_y] by auto
qed



lemma Max_scale:
"(MAX xa. |s| * abs_infty_q (poly.coeff (of_qr x) xa)) =
    |s| * (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa))"
proof -
  have "(MAX xa. |s| * abs_infty_q (poly.coeff (of_qr x) xa)) =
    (Max (range (λxa. |s| * abs_infty_q (poly.coeff (of_qr x) xa))))"
    by auto
```

```
  moreover have "... = (Max ((λa. |s| * a) `
    (range (λxa. abs_infty_q (poly.coeff (of_qr x) xa)))))"
    by (metis range_composition)
  moreover have "... = |s| * (Max (range
    (λxa. abs_infty_q (poly.coeff (of_qr x) xa))))"
    by (subst mono_Max_commute[symmetric])
       (auto simp add: finite_Max Rings.mono_mult)
  moreover have "... =  |s| *
    (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa))"
    by auto
  ultimately show ?thesis by auto
qed
```

Show that abs_infty_q is definite, positive and fulfils the triangle inequality.

```
lemma abs_infty_q_definite:
  "abs_infty_q x = 0 ⟷ x = 0"
proof (auto simp add: abs_infty_q_def
  mod_plus_minus_zero'[OF q_gt_zero q_odd])
  assume "to_int_mod_ring x mod+- q = 0"
  then have "to_int_mod_ring x mod q = 0"
    using mod_plus_minus_zero[of "to_int_mod_ring x" q]
    by auto
  then have "to_int_mod_ring x = 0"
    using to_int_mod_ring_range CARD_a
    by (metis mod_rangeE range_eqI)
  then show "x = 0" by force
qed

lemma abs_infty_q_pos:
  "abs_infty_q x ≥ 0"
by (auto simp add: abs_infty_q_def)


lemma abs_infty_q_minus:
  "abs_infty_q (- x) = abs_infty_q x"
proof (cases "x=0")
case True
  then show ?thesis by auto
next
case False
  have minus_x: "to_int_mod_ring (-x) = q - to_int_mod_ring x"
  proof -
    have "to_int_mod_ring (-x) = to_int_mod_ring (-x) mod q"
      by (metis CARD_a Rep_mod_ring_mod to_int_mod_ring.rep_eq)
    also have "... = (- to_int_mod_ring x) mod q"
      by (metis (no_types, opaque_lifting) CARD_a diff_eq_eq
        mod_add_right_eq plus_mod_ring.rep_eq to_int_mod_ring.rep_eq
        uminus_add_conv_diff)
    also have "... = q - to_int_mod_ring x"
```

```
    proof -
      have "- to_int_mod_ring x ∈ {-q<..<0}"
      using CARD_a range_to_int_mod_ring False
        by (smt (verit, best) Rep_mod_ring_mod greaterThanLessThan_iff

          q_gt_zero to_int_mod_ring.rep_eq to_int_mod_ring_hom.eq_iff

          to_int_mod_ring_hom.hom_zero zmod_trivial_iff)
      then have "q-to_int_mod_ring x∈{0<..<q}" by auto
      then show ?thesis
        using minus_mod_self1 mod_rangeE
        by (simp add: to_int_mod_ring.rep_eq zmod_zminus1_eq_if)
    qed
    finally show ?thesis by auto
  qed
  then have "|to_int_mod_ring (- x) mod+- q| =
    |(q - (to_int_mod_ring x)) mod+- q|"
    by auto
  also have "... = | (- to_int_mod_ring x) mod+- q|"
    unfolding mod_plus_minus_def by (smt (z3) mod_add_self2)
  also have "... = | - (to_int_mod_ring x mod+- q)|"
    using neg_mod_plus_minus[OF q_odd q_gt_zero,
      of "to_int_mod_ring x"] by simp
  also have "... = |to_int_mod_ring x mod+- q|" by auto
  finally show ?thesis unfolding abs_infty_q_def by auto
qed
```

```
lemma to_int_mod_ring_mult:
  "to_int_mod_ring (a*b) =  to_int_mod_ring (a::'a mod_ring) *
    to_int_mod_ring (b::'a mod_ring) mod q"
by (metis (no_types, lifting) CARD_a of_int_hom.hom_mult
  of_int_mod_ring.rep_eq of_int_mod_ring_to_int_mod_ring
  of_int_of_int_mod_ring to_int_mod_ring.rep_eq)
```

Scaling only with inequality not equality! This causes a problem in proof of the Kyber scheme. Needed to add $q \equiv 1 \mod 4$ to change proof.

```
lemma mod_plus_minus_leq_mod:
  "|x mod+- q| ≤ |x|"
by (smt (verit, best) atLeastAtMost_iff mod_plus_minus_range_odd
  mod_plus_minus_rangeE q_gt_zero q_odd)

lemma abs_infty_q_scale_pos:
  assumes "s≥0"
  shows "abs_infty_q ((of_int_mod_ring s :: 'a mod_ring) * x) ≤
    |s| * (abs_infty_q x)"
proof -
  have "|to_int_mod_ring (of_int_mod_ring s * x) mod+- q| =
```

```
      |(to_int_mod_ring (of_int_mod_ring s ::'a mod_ring) *
        to_int_mod_ring x mod q) mod+- q|"
    using to_int_mod_ring_mult[of "of_int_mod_ring s" x] by simp
  also have "... = |(s mod q * to_int_mod_ring x) mod+- q|"
  by (simp add: CARD_a mod_plus_minus_def of_int_mod_ring.rep_eq to_int_mod_ring.rep_eq)
  also have "... ≤ |s mod q| * |to_int_mod_ring x mod+- q|"
  proof -
    have "|s mod q * to_int_mod_ring x mod+- q| =
          |(s mod q mod+- q) * (to_int_mod_ring x mod+- q) mod+- q|"
      using mod_plus_minus_mult by auto
    also have "... ≤ |(s mod q mod+- q) * (to_int_mod_ring x mod+- q)|"

      using mod_plus_minus_leq_mod by blast
    also have "... ≤ |s mod q mod+- q| * |(to_int_mod_ring x mod+- q)|"

      by (simp add: abs_mult)
    also have "... ≤ |s mod q| * |(to_int_mod_ring x mod+- q)|"
      using mod_plus_minus_leq_mod[of "s mod q"]
      by (simp add: mult_right_mono)
    finally show ?thesis by auto
  qed
  also have "... ≤ |s| * |to_int_mod_ring x mod+- q|" using assms
    by (simp add: mult_mono' q_gt_zero zmod_le_nonneg_dividend)
  finally show ?thesis unfolding abs_infty_q_def by auto
qed

lemma abs_infty_q_scale_neg:
  assumes "s<0"
  shows "abs_infty_q ((of_int_mod_ring s :: 'a mod_ring) * x) ≤
    |s| * (abs_infty_q x)"
using abs_infty_q_minus abs_infty_q_scale_pos
by (smt (verit, best) mult_minus_left of_int_minus of_int_of_int_mod_ring)

lemma abs_infty_q_scale:
  "abs_infty_q ((of_int_mod_ring s :: 'a mod_ring) * x) ≤
    |s| * (abs_infty_q x)"
apply (cases "s≥0")
using abs_infty_q_scale_pos apply presburger
using abs_infty_q_scale_neg by force
```

Triangle inequality for `abs_infty_q`.

```
lemma abs_infty_q_triangle_ineq:
  "abs_infty_q (x+y) ≤ abs_infty_q x + abs_infty_q y"
proof -
  have "to_int_mod_ring (x + y) mod+- q =
        (to_int_mod_ring x + to_int_mod_ring y) mod q mod+-q"
    by (simp add: to_int_mod_ring_def CARD_a plus_mod_ring.rep_eq)
  also have "... = (to_int_mod_ring x + to_int_mod_ring y) mod+-q"
    unfolding mod_plus_minus_def by auto
```

```
  also have "... = (to_int_mod_ring x mod+- q +
    to_int_mod_ring y mod+- q) mod+- q"
    unfolding mod_plus_minus_def
    by (smt (verit, ccfv_threshold) minus_mod_self2 mod_add_eq)
  finally have rewrite:"to_int_mod_ring (x + y) mod+- q =
    (to_int_mod_ring x mod+- q + to_int_mod_ring y mod+- q) mod+- q" .
  then have "|to_int_mod_ring (x + y) mod+- q|
    ≤ |to_int_mod_ring x mod+- q| + |to_int_mod_ring y mod+- q|"
    proof (cases
    "(to_int_mod_ring x mod+- q + to_int_mod_ring y mod+- q) ∈
      {-⌊real_of_int q/2⌋..<⌊real_of_int q/2⌋}")
    case True
      then have True': "to_int_mod_ring x mod+- q + to_int_mod_ring y
mod+- q
        ∈ {- ⌊real_of_int q / 2⌋..⌊real_of_int q / 2⌋}" by auto
      then have "(to_int_mod_ring x mod+- q +
        to_int_mod_ring y mod+- q) mod+- q
       = to_int_mod_ring x mod+- q + to_int_mod_ring y mod+- q"
        using mod_plus_minus_rangeE[OF True' q_odd q_gt_zero] by auto

      then show ?thesis by (simp add: rewrite)
    next
    case False
      then have "|(to_int_mod_ring x mod+- q +
        to_int_mod_ring y mod+- q)| ≥ ⌊real_of_int q /2⌋"
        by auto
      then have "|(to_int_mod_ring x mod+- q +
        to_int_mod_ring y mod+- q)| ≥ |(to_int_mod_ring x mod+- q +
        to_int_mod_ring y mod+- q) mod+- q|"
      using mod_plus_minus_range_odd[OF q_gt_zero q_odd,
        of "(to_int_mod_ring x mod+- q + to_int_mod_ring y mod+- q)"]
      by auto
      then show ?thesis by (simp add: rewrite)
    qed
  then show ?thesis
    by (auto simp add: abs_infty_q_def mod_plus_minus_def)
qed
```

Show that `abs_infty_poly` is definite, positive and fulfils the triangle inequality.

```
lemma abs_infty_poly_definite:
  "abs_infty_poly x = 0 ⟷ x = 0"
proof (auto simp add: abs_infty_poly_def abs_infty_q_definite)
  assume "(MAX xa. abs_infty_q (poly.coeff (of_qr x) xa)) = 0"
  then have abs_le_zero: "abs_infty_q (poly.coeff (of_qr x) xa) ≤ 0"
    for xa
    using Max_ge[OF finite_Max[of x],
      of "abs_infty_q (poly.coeff (of_qr x) xa)"]
    by (auto simp add: Max_ge[OF finite_Max])
```

```
  have "abs_infty_q (poly.coeff (of_qr x) xa) = 0" for xa
    using abs_infty_q_pos[of "poly.coeff (of_qr x) xa"]
    abs_le_zero[of xa] by auto
  then have "poly.coeff (of_qr x) xa = 0" for xa
    by (auto simp add: abs_infty_q_definite)
  then show "x = 0"
    using leading_coeff_0_iff of_qr_eq_0_iff by blast
qed
```

```
lemma abs_infty_poly_pos:
  "abs_infty_poly x ≥ 0"
proof (auto simp add: abs_infty_poly_def)
  have f_ge_zero: "∀xa. abs_infty_q (poly.coeff (of_qr x) xa) ≥ 0"
    by (auto simp add: abs_infty_q_pos)
  then show " 0 ≤ (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa))"
    using all_impl_Max[OF f_ge_zero finite_Max] by auto
qed
```

Again, homogeneity is only true for inequality not necessarily equality! Need to add $q \equiv 1 \mod 4$ such that proof of crypto scheme works out.

```
lemma abs_infty_poly_scale:
  "abs_infty_poly ((to_module s) * x) ≤ (abs s) * (abs_infty_poly x)"
proof -
  have fin1: "finite (range (λxa. abs_infty_q (of_int_mod_ring s *
    poly.coeff (of_qr x) xa)))"
    using finite_Max_scale by auto
  have fin2: "finite (range (λxa. |s| *
     abs_infty_q (poly.coeff (of_qr x) xa)))"
    by (metis finite_Max finite_imageI range_composition)
  have "abs_infty_poly (to_module s * x) =
        (MAX xa. abs_infty_q
         ((of_int_mod_ring s) * poly.coeff (of_qr x) xa))"
  using abs_infty_poly_def to_module_mult
    by (metis (mono_tags, lifting) comp_apply image_cong)
  also have "... ≤ (MAX xa. |s| * abs_infty_q (poly.coeff (of_qr x) xa))"
    using abs_infty_q_scale fin1 fin2 by (subst Max_mono', auto)
  also have "... = |s| * abs_infty_poly x"
    unfolding abs_infty_poly_def comp_def using Max_scale by auto
  finally show ?thesis by blast
qed
```

Triangle inequality for `abs_infty_poly`.

```
lemma abs_infty_poly_triangle_ineq:
  "abs_infty_poly (x+y) ≤ abs_infty_poly x + abs_infty_poly y"
proof -
  have "abs_infty_q (poly.coeff (of_qr x) xa +
    poly.coeff (of_qr y) xa) ≤
    abs_infty_q (poly.coeff (of_qr x) xa) +
```

```
    abs_infty_q (poly.coeff (of_qr y) xa)"
    for xa
    using abs_infty_q_triangle_ineq[of
      "poly.coeff (of_qr x) xa" "poly.coeff (of_qr y) xa"]
    by auto
  then have abs_q_triang: "∀xa.
    abs_infty_q (poly.coeff (of_qr x) xa + poly.coeff (of_qr y) xa) ≤
    abs_infty_q (poly.coeff (of_qr x) xa) +
    abs_infty_q (poly.coeff (of_qr y) xa)"
    by auto
  have "(MAX xa. abs_infty_q (poly.coeff (of_qr x) xa +
      poly.coeff (of_qr y) xa))
    ≤ (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa) +
      abs_infty_q (poly.coeff (of_qr y) xa))"
    using Max_mono'[OF abs_q_triang finite_Max_sum finite_Max_sum']
    by auto
  also have "... ≤ (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa)) +
       (MAX xb. abs_infty_q (poly.coeff (of_qr y) xb))"
    using Max_mono_plus[OF finite_Max[of x] finite_Max[of y]]
    by auto
  finally have "(MAX xa. abs_infty_q (poly.coeff (of_qr x) xa +
      poly.coeff (of_qr y) xa))
    ≤ (MAX xa. abs_infty_q (poly.coeff (of_qr x) xa)) +
       (MAX xb. abs_infty_q (poly.coeff (of_qr y) xb))"
    by auto
  then show ?thesis
    by (auto simp add: abs_infty_poly_def)
qed

end
```

Estimation inequality using message bit.

```
lemma(in kyber_spec) abs_infty_poly_ineq_pm_1:
assumes "∃x. poly.coeff (of_qr a) x ∈ {of_int_mod_ring (-1),1}"
shows "abs_infty_poly (to_module (round((real_of_int q)/2)) * a) ≥
              2 * round (real_of_int q / 4)"
proof -
  let ?x = "to_module (round((real_of_int q)/2)) * a"
  obtain x1 where x1_def:
    "poly.coeff (of_qr a) x1 ∈ {of_int_mod_ring(-1),1}"
    using assms by auto
  have "abs_infty_poly (to_module (round((real_of_int q)/2)) * a)
    ≥ abs_infty_q (poly.coeff (of_qr (to_module
      (round (real_of_int q / 2)) * a)) x1)"
    unfolding abs_infty_poly_def using x1_def
    by (simp add: finite_Max)
  also have "abs_infty_q (poly.coeff (of_qr (to_module
    (round (real_of_int q / 2)) * a)) x1)
  = abs_infty_q (of_int_mod_ring (round (real_of_int q / 2))
```

```
    * (poly.coeff (of_qr a) x1))"
    using to_module_mult[of "round (real_of_int q / 2)" a]
    by simp
  also have "... = abs_infty_q (of_int_mod_ring
    (round (real_of_int q / 2)))"
  proof -
    consider "poly.coeff (of_qr a) x1=1" |
      "poly.coeff (of_qr a) x1 = of_int_mod_ring (-1)"
      using x1_def by auto
    then show ?thesis
    proof (cases)
      case 2
      then show ?thesis
      by (metis abs_infty_q_minus mult.right_neutral mult_minus_right
          of_int_hom.hom_one of_int_minus of_int_of_int_mod_ring)
    qed (auto)
  qed
  also have "... = |round (real_of_int q / 2) mod+- q|"
    unfolding abs_infty_q_def
    using to_int_mod_ring_of_int_mod_ring
    by (simp add: CARD_a mod_add_left_eq mod_plus_minus_def
      of_int_mod_ring.rep_eq to_int_mod_ring.rep_eq)
  also have "... = |((q + 1) div 2) mod+- q|"
    using odd_round_up[OF q_odd] by auto
  also have "... = |((2 * q) div 2) mod q - (q - 1) div 2|"
  proof -
    have "(q + 1) div 2 mod q = (q + 1) div 2" using q_gt_two by auto
    moreover have "(q + 1) div 2 - q = - ((q - 1) div 2)" by (simp add:
q_odd)
    ultimately show ?thesis
    unfolding mod_plus_minus_def odd_half_floor[OF q_odd]
    by (split if_splits) simp
  qed
  also have "... = |(q-1) div 2|" using q_odd
    by (subst nonzero_mult_div_cancel_left[of 2 q], simp)
       (simp add: abs_div abs_minus_commute)
  also have "... = 2 * ((q-1) div 4)"
  proof -
    from q_gt_two have "(q-1) div 2 > 0" by simp
    then have "|(q-1) div 2| = (q-1) div 2" by auto
    also have "... = 2 * ((q-1) div 4)"
      by (subst div_mult_swap) (use q_mod_4 in
      ‹metis dvd_minus_mod›, force)
    finally show ?thesis by blast
  qed
  also have "... = 2 * round (real_of_int q / 4)"
    unfolding odd_round_up[OF q_odd] one_mod_four_round[OF q_mod_4]
    by (simp add: round_def)
  finally show ?thesis unfolding abs_infty_poly_def by simp
```

```
qed

end
theory Compress

imports Kyber_spec
        Mod_Plus_Minus
        Abs_Qr
        "HOL-Analysis.Finite_Cartesian_Product"

begin

lemma prime_half:
  assumes "prime (p::int)"
          "p > 2"
  shows "⌈p / 2⌉ > ⌊p / 2⌋"
proof -
  have "odd p" using prime_odd_int[OF assms] .
  then have "⌈p / 2⌉ > p/2"
  by (smt (verit, best) cos_npi_int cos_zero_iff_int
    le_of_int_ceiling mult.commute times_divide_eq_right)
  then have "⌊p / 2⌋ < p/2"
  by (meson floor_less_iff less_ceiling_iff)
  then show ?thesis using ‹⌈p / 2⌉ > p/2› by auto
qed

lemma ceiling_int:
  "⌈of_int a + b⌉ = a + ⌈b⌉"
unfolding ceiling_def by (simp add: add.commute)

lemma deg_Poly':
  assumes "Poly xs ≠ 0"
  shows "degree (Poly xs) ≤ length xs - 1"
proof (induct xs)
  case (Cons a xs)
  then show ?case
    by simp (metis Poly.simps(1) Suc_le_eq Suc_pred
    le_imp_less_Suc length_greater_0_conv)
qed simp


context kyber_spec begin
```

6 Compress and Decompress Functions

Properties of the *mod+-* function.

```
lemma two_mid_lt_q:
  "2 * ⌊real_of_int q/2⌋ < q"
```

```
using oddE[OF prime_odd_int[OF q_prime q_gt_two]] by fastforce

lemma mod_plus_minus_range_q:
  assumes "y ∈ {-⌊q/2⌋..⌊q/2⌋}"
  shows "y mod+- q = y"
using assms mod_plus_minus_rangeE q_gt_zero q_odd by presburger
```

Compression only works for $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where d is a positive integer with $d <\rceil \log_2(q)\lceil$. For compression we omit the least important bits. Decompression rescales to the modulus q.

```
definition compress :: "nat ⇒ int ⇒ int" where
  "compress d x =
  round (real_of_int (2^d * x) / real_of_int q) mod (2^d)"

definition decompress :: "nat ⇒ int ⇒ int" where
  "decompress d x =
  round (real_of_int q * real_of_int x / real_of_int 2^d)"

lemma compress_zero: "compress d 0 = 0"
unfolding compress_def by auto

lemma compress_less:
  ‹compress d x < 2 ^ d›
  by (simp add: compress_def)

lemma decompress_zero: "decompress d 0 = 0"
unfolding decompress_def by auto
```

Properties of the exponent d.

```
lemma d_lt_logq:
  assumes "of_nat d < ⌈(log 2 q)::real⌉"
  shows "d< log 2 q"
using assms by linarith

lemma twod_lt_q:
  assumes "of_nat d < ⌈(log 2 q)::real⌉"
  shows "2 powr (real d) < of_int q"
using assms less_log_iff[of 2 q d] d_lt_logq q_gt_zero
by auto

lemma break_point_gt_q_div_two:
  assumes "of_nat d < ⌈(log 2 q)::real⌉"
  shows "⌈q-(q/(2*2^d))⌉ > ⌊q/2⌋"
proof -
```

```
  have "1/((2::real)^d) ≤ (1::real)"
    using one_le_power[of 2 d] by simp
  have "⌈q-(q/(2*2^d))⌉ ≥ q-(q/2)* (1/(2^d))" by simp
  moreover have "q-(q/2)* (1/(2^d)) ≥ q - q/2"
    using ‹1/((2::real)^d) ≤ (1::real)›
    by (smt (z3) calculation divide_le_eq divide_nonneg_nonneg
      divide_self_if mult_left_mono of_int_nonneg
      times_divide_eq_right q_gt_zero)
  ultimately have "⌈q-(q/(2*2^d))⌉ ≥ ⌈q/2⌉ " by linarith
  moreover have "⌈q/2⌉ > ⌊q/2⌋"
    using prime_half[OF q_prime q_gt_two] .
  ultimately show ?thesis by auto
qed

lemma decompress_zero_unique:
  assumes "decompress d s = 0"
          "s ∈ {0..2^d - 1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "s = 0"
proof -
  let ?x = "real_of_int q * real_of_int s /
    real_of_int 2^d + 1/2"
  have "0 ≤ ?x ∧ ?x < 1" using assms(1)
     unfolding decompress_def round_def
    using floor_correct[of ?x] by auto
  then have "real_of_int q * real_of_int s /
    real_of_int 2^d < 1/2" by linarith
  moreover have "real_of_int q / real_of_int 2^d > 1"
    using twod_lt_q[OF assms(3)]
    by (simp add: powr_realpow)
  ultimately have "real_of_int s < 1/2"
  by (smt (verit, best) divide_less_eq_1_pos field_sum_of_halves
    pos_divide_less_eq times_divide_eq_left)
  then show ?thesis using assms(2) by auto
qed
```

Range of compress and decompress functions

```
lemma range_compress:
  assumes "x∈{0..q-1}" "of_nat d < ⌈(log 2 q)::real⌉"
  shows "compress d x ∈ {0..2^d - 1}"
using compress_def by auto

lemma range_decompress:
  assumes "x∈{0..2^d - 1}" "of_nat d < ⌈(log 2 q)::real⌉"
  shows "decompress d x ∈ {0..q-1}"
using decompress_def assms
proof (auto, goal_cases)
case 1
  then show ?case
```

```
  by (smt (verit, best) divide_eq_0_iff divide_numeral_1
    less_divide_eq_1_pos mult_of_int_commute
    nonzero_mult_div_cancel_right of_int_eq_0_iff
    of_int_less_1_iff powr_realpow q_gt_zero q_nonzero
    round_0 round_mono twod_lt_q zero_less_power)
next
case 2
  have "real_of_int q/2^d > 1" using twod_lt_q[OF assms(2)]
    by (simp add: powr_realpow)
  then have log: "real_of_int q - real_of_int q/2^d ≤ q-1" by simp
  have "x ≤ 2^d-1" using assms(1) by simp
  then have "real_of_int x ≤ 2^d - 1" by (simp add: int_less_real_le)
  then have "real_of_int q * real_of_int x / 2^d ≤
    real_of_int q * (2^d-1) / 2^d"
    by (smt (verit, best) divide_strict_right_mono
      mult_less_cancel_left_pos of_int_pos q_gt_zero
      zero_less_power)
  also have "... = real_of_int q * 2^d /2^d - real_of_int q/2^d"
    by (simp add: diff_divide_distrib right_diff_distrib)
  finally have "real_of_int q * real_of_int x / 2^d ≤
    real_of_int q - real_of_int q/2^d" by simp
  then have "round (real_of_int q * real_of_int x / 2^d) ≤
    round (real_of_int q - real_of_int q/2^d)"
    using round_mono by blast
  also have "... ≤ q - 1"
    using log by (metis round_mono round_of_int)
  finally show ?case by auto
qed
```

Compression is a function qrom $\mathbb{Z}/q\mathbb{Z}$ to $\mathbb{Z}/(2^d)\mathbb{Z}$.

```
lemma compress_in_range:
  assumes "x∈{0..⌈q-(q/(2*2^d))⌉-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "round (real_of_int (2^d * x) / real_of_int q) < 2^d "
proof -
  have "(2::int)^d ≠ 0" by simp
  have "real_of_int x < real_of_int q - real_of_int q / (2 * 2^d)"
    using assms(1) less_ceiling_iff by auto
  then have "2^d * real_of_int x / real_of_int q <
    2^d * (real_of_int q - real_of_int q / (2 * 2^d)) /
    real_of_int q"
    by (simp add: divide_strict_right_mono q_gt_zero)
  also have "... = 2^d * ((real_of_int q / real_of_int q) -
    (real_of_int q / real_of_int q) / (2 * 2^d))"
  by (simp add:algebra_simps diff_divide_distrib)
  also have "... = 2^d * (1 - 1/(2*2^d))"
    using q_nonzero by simp
  also have "... = 2^d - 1/2"
    using ‹2^d ≠ 0› by (simp add: right_diff_distrib')
```

```
  finally have "2^d * real_of_int x / real_of_int q <
    2^d - (1::real)/(2::real)"
    by auto
  then show ?thesis unfolding round_def
    using floor_less_iff by fastforce
qed
```

When does the modulo operation in the compression function change the output? Only when `x` ≥ ⌈`q-(q / (2*2^d))`⌉. Then we can determine that the compress function maps to zero. This is why we need the `mod+-` in the definition of Compression. Otherwise the error bound would not hold.

```
lemma compress_no_mod:
  assumes "x∈{0..⌈q-(q / (2*2^d))⌉-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "compress d x =
    round (real_of_int (2^d * x) / real_of_int q)"
unfolding compress_def
using compress_in_range[OF assms] assms(1) q_gt_zero
by (smt (z3) atLeastAtMost_iff divide_nonneg_nonneg
  mod_pos_pos_trivial mult_less_cancel_left_pos
  of_int_nonneg of_nat_0_less_iff right_diff_distrib'
  round_0 round_mono zero_less_power)


lemma compress_2d:
  assumes "x∈{⌈q-(q/(2*2^d))⌉..q-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "round (real_of_int (2^d * x) / real_of_int q) = 2^d "
using assms proof -
  have "(2::int)^d ≠ 0" by simp
  have "round (real_of_int (2^d * x) / real_of_int q) ≥ 2^d"
  proof -
    have "real_of_int x ≥ real_of_int q - real_of_int q / (2 * 2^d)"
      using assms(1) ceiling_le_iff by auto
    then have "2^d * real_of_int x / real_of_int q ≥
      2^d * (real_of_int q - real_of_int q / (2 * 2^d)) /
      real_of_int q"
      using q_gt_zero by (simp add: divide_right_mono)
    also have "2^d * (real_of_int q - real_of_int q /
      (2 * 2^d)) / real_of_int q
      = 2^d * ((real_of_int q / real_of_int q) -
      (real_of_int q / real_of_int q) / (2 * 2^d))"
      by (simp add:algebra_simps diff_divide_distrib)
    also have "... = 2^d * (1 - 1/(2*2^d))"
      using q_nonzero by simp
    also have "... = 2^d - 1/2"
      using ‹2^d ≠ 0› by (simp add: right_diff_distrib')
    finally have "2^d * real_of_int x / real_of_int q ≥
      2^d - (1::real)/(2::real)"
      by auto
```

```
    then show ?thesis unfolding round_def using le_floor_iff by force
  qed
  moreover have "round (real_of_int (2^d * x) / real_of_int q) ≤ 2^d"
  proof -
    have "d < log 2 q" using assms(2) by linarith
    then have "2 powr (real d) < of_int q"
      using less_log_iff[of 2 q d] q_gt_zero by auto
    have "x < q" using assms(1) by auto
    then have "real_of_int x/ real_of_int q < 1"
      by (simp add: q_gt_zero)
    then have "real_of_int (2^d * x) / real_of_int q <
      real_of_int (2^d)"
      by (auto) (smt (verit, best) divide_less_eq_1_pos
        nonzero_mult_div_cancel_left times_divide_eq_right
        zero_less_power)
    then show ?thesis unfolding round_def by linarith
  qed
  ultimately show ?thesis by auto
qed

lemma compress_mod:
  assumes "x∈{⌈q-(q/(2*2^d))⌉..q-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "compress d x = 0"
unfolding compress_def using compress_2d[OF assms] by simp
```

Error after compression and decompression of data. To prove the error bound, we distinguish the cases where the *mod+-* is relevant or not.

First let us look at the error bound for no *mod+-* reduction.

```
lemma decompress_compress_no_mod:
  assumes "x∈{0..⌈q-(q/(2*2^d))⌉-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "abs (decompress d (compress d x) - x) ≤
    round ( real_of_int q / real_of_int (2^(d+1)))"
proof -
  have "abs (decompress d (compress d x) - x) =
    abs (real_of_int (decompress d (compress d x)) -
    real_of_int q / real_of_int (2^d) *
    (real_of_int (2^d * x) / real_of_int q))"
    using q_gt_zero by force
  also have "... ≤ abs (real_of_int (decompress d (compress d x)) -
    real_of_int q / real_of_int (2^d) * real_of_int (compress d x)) +
    abs (real_of_int q / real_of_int (2^d) *
    real_of_int (compress d x) - real_of_int q / real_of_int (2^d) *
    real_of_int (2^d) / real_of_int q * x)"
    using abs_triangle_ineq[of
      "real_of_int (decompress d (compress d x)) -
       real_of_int q / real_of_int (2^d) * real_of_int (compress d x)"
```

```
    "real_of_int q / real_of_int (2^d) * real_of_int (compress d x)
     - real_of_int q / real_of_int (2^d) * real_of_int (2^d) /
     real_of_int q * real_of_int x"] by auto
also have "... ≤ 1/2 + abs (real_of_int q / real_of_int (2^d) *
  (real_of_int (compress d x) -
   real_of_int (2^d) / real_of_int q * real_of_int x))"
  proof -
    have part_one:
      "abs (real_of_int (decompress d (compress d x)) -
      real_of_int q / real_of_int (2^d) * real_of_int (compress d x))
      ≤ 1/2"
      unfolding decompress_def
      using of_int_round_abs_le[of "real_of_int q *
        real_of_int (compress d x) / real_of_int 2^d"] by simp
    have "real_of_int q * real_of_int (compress d x) / 2^d -
      real_of_int x =
      real_of_int q * (real_of_int (compress d x) -
      2^d * real_of_int x / real_of_int q) / 2^d"
      by (smt (verit, best) divide_cancel_right
      nonzero_mult_div_cancel_left of_int_eq_0_iff
      q_nonzero right_diff_distrib times_divide_eq_left
      zero_less_power)
    then have part_two:
      "abs (real_of_int q / real_of_int (2^d) *
      real_of_int (compress d x) -
      real_of_int q / real_of_int (2^d) * real_of_int (2^d) /
      real_of_int q * x) =
      abs (real_of_int q / real_of_int (2^d) *
      (real_of_int (compress d x) - real_of_int (2^d) /
      real_of_int q * x)) " by auto
    show ?thesis using part_one part_two by auto
 qed
also have "... = 1/2 + (real_of_int q / real_of_int (2^d)) *
    abs (real_of_int (compress d x) - real_of_int (2^d) /
    real_of_int q * real_of_int x)"
  by (subst abs_mult) (smt (verit, best) assms(2)
    less_divide_eq_1_pos of_int_add of_int_hom.hom_one
    of_int_power powr_realpow twod_lt_q zero_less_power)
also have "... ≤ 1/2 + (real_of_int q / real_of_int (2^d)) * (1/2) "
  using compress_no_mod[OF assms]
  using of_int_round_abs_le[of "real_of_int (2^d) *
    real_of_int x / real_of_int q"]
  by (smt (verit, ccfv_SIG) divide_nonneg_nonneg left_diff_distrib
    mult_less_cancel_left_pos of_int_mult of_int_nonneg q_gt_zero
    times_divide_eq_left zero_le_power)
finally have "real_of_int (abs (decompress d (compress d x) - x)) ≤
              real_of_int q / real_of_int (2*2^d) + 1/2"
  by simp
then show ?thesis unfolding round_def using le_floor_iff
```

```
    by fastforce
qed


lemma no_mod_plus_minus:
  assumes "abs y ≤ round ( real_of_int q / real_of_int (2^(d+1)))"
          "d>0"
  shows "abs y = abs (y mod+- q)"
proof -
  have "round (real_of_int q / real_of_int (2^(d+1))) ≤ ⌊q/2⌋"
  unfolding round_def
  proof -
    have "real_of_int q/real_of_int (2^d) ≤ real_of_int q/2"
    using ‹d>0›
    proof -
      have "1 / real_of_int (2^d) ≤ 1/2"
        using ‹d>0› inverse_of_nat_le[of 2 "2^d"]
        by (simp add: self_le_power)
      then show ?thesis using q_gt_zero
        by (smt (verit, best) frac_less2 of_int_le_0_iff zero_less_power)
    qed
    moreover have "real_of_int q/2 + 1 ≤ real_of_int q"
      using q_gt_two by auto
    ultimately have "real_of_int q / real_of_int (2^d) + 1 ≤
      real_of_int q" by linarith
    then have fact: "real_of_int q / real_of_int (2 ^ (d + 1)) +
      1/2 ≤ real_of_int q/2"
      by auto
    then show "⌊real_of_int q / real_of_int (2 ^ (d + 1)) + 1/2⌋ ≤
      ⌊real_of_int q/2⌋"
      using floor_mono[OF fact] by auto
  qed
  then have "abs y ≤ ⌊q/2⌋" using assms by auto
  then show ?thesis using mod_plus_minus_range_odd[OF q_gt_zero q_odd]

  by (smt (verit, del_insts) mod_plus_minus_def mod_pos_pos_trivial neg_mod_plus_minus

    q_odd two_mid_lt_q)
qed


lemma decompress_compress_no_mod_plus_minus:
  assumes "x∈{0..⌈q-(q/(2*2^d))⌉-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
          "d>0"
  shows "abs ((decompress d (compress d x) - x) mod+- q) ≤
          round ( real_of_int q / real_of_int (2^(d+1)))"
proof -
  have "abs ((decompress d (compress d x) - x) mod+- q) =
```

```
      abs ((decompress d (compress d x) - x)) "
    using no_mod_plus_minus[OF decompress_compress_no_mod
      [OF assms(1) assms(2)] assms(3)] by auto
  then show ?thesis using decompress_compress_no_mod
    [OF assms(1) assms(2)] by auto
qed
```

Now lets look at what happens when the `mod+-` reduction comes into action.

```
lemma decompress_compress_mod:
  assumes "x∈{⌈q-(q/(2*2^d))⌉..q-1}"
          "of_nat d < ⌈(log 2 q)::real⌉"
  shows "abs ((decompress d (compress d x) - x) mod+- q) ≤
         round ( real_of_int q / real_of_int (2^(d+1)))"
proof -
  have "(decompress d (compress d x) - x) = - x"
    using compress_mod[OF assms] unfolding decompress_def
    by auto
  moreover have "-x mod+- q = -x+q"
  proof -
    have range_x: "x ∈ {⌊real_of_int q / 2⌋<..q - 1}" using assms(1)

      break_point_gt_q_div_two[OF assms(2)] by auto
    then have *: "- x ∈ {-q + 1..< -⌊real_of_int q / 2⌋}" by auto
    have **: "-x + q ∈{0..<q-⌊real_of_int q / 2⌋}" using * by auto
    have "-x + q ∈{0..<q}"
    proof (subst atLeastLessThan_iff)
      have "q-⌊real_of_int q / 2⌋ ≤ q" using q_gt_zero by auto
      moreover have "0 ≤ - x + q ∧ - x + q < q-⌊real_of_int q / 2⌋"
using ** by auto
      ultimately show "0 ≤ - x + q ∧ - x + q < q" by linarith
    qed
    then have rew: "-x mod q = -x + q" using mod_rangeE by fastforce
    have "-x mod q < q - ⌊real_of_int q / 2⌋" using ** by (subst rew)(auto
simp add: * range_x)
    then have "⌊real_of_int q / 2⌋ ≥ - x mod q" by linarith
    then show ?thesis unfolding mod_plus_minus_def using rew by auto
  qed
  moreover have "abs (q - x) ≤ round ( real_of_int q /
    real_of_int (2^(d+1)))"
  proof -
    have "abs (q-x) = q-x"
      using assms(1) by auto
    also have "... ≤ q - ⌈q - q/(2*2^d)⌉"
      using assms(1) by simp
    also have "... = - ⌈- q/(2*2^d)⌉"
      using ceiling_int[of q "- q/(2*2^d)"] by auto
    also have "... = ⌊q/(2*2^d)⌋"
      by (simp add: ceiling_def)
    also have "... ≤ round (q/(2*2^d))"
```

```
    using floor_le_round by blast
  finally have "abs (q-x) ≤ round (q/(2^(d+1)))" by auto
  then show ?thesis by auto
 qed
 ultimately show ?thesis by auto
qed
```

Together, we can determine the general error bound on decompression of compression of the data. This error needs to be small enough not to disturb the encryption and decryption process.

```
lemma decompress_compress:
  assumes "x∈{0..<q}"
          "of_nat d < ⌈(log 2 q)::real⌉"
          "d>0"
  shows "let x' = decompress d (compress d x) in
         abs ((x' - x) mod+- q) ≤
          round ( real_of_int q / real_of_int (2^(d+1)) )"
proof (cases "x<⌈q-(q/(2*2^d))⌉")
case True
  then have range_x: "x∈{0..⌈q-(q/(2*2^d))⌉-1}"
    using assms(1) by auto
  show ?thesis unfolding Let_def
    using decompress_compress_no_mod_plus_minus[OF
    range_x assms(2) assms(3)] by auto
next
case False
  then have range_x: "x∈{⌈q-(q/(2*2^d))⌉..q-1}"
    using assms(1) by auto
  show ?thesis unfolding Let_def
    using decompress_compress_mod[OF range_x assms(2)]
    by auto
qed
```

We have now defined compression only on integers (ie *{0..<q}*, corresponding to ℤ_*q*). We need to extend this notion to the ring ℤ_*q[X]/(X^n+1)*. Here, a compressed polynomial is the compression on every coefficient.

How to channel through the types

- *to_qr :: 'a mod_ring poly ⇒ 'a qr*
- *Poly :: 'a mod_ring list ⇒ 'a mod_ring poly*
- *map of_int_mod_ring :: int list ⇒ 'a mod_ring list*
- *map compress :: int list ⇒ int list*
- *map to_int_mod_ring :: 'a mod_ring list ⇒ int list*
- *coeffs :: 'a mod_ring poly ⇒ 'a mod_ring list*

- *of_qr :: 'a qr ⇒ 'a mod_ring poly*

```
definition compress_poly :: "nat ⇒ 'a qr ⇒ 'a qr" where
  "compress_poly d =
        to_qr ∘
        Poly ∘
        (map of_int_mod_ring) ∘
        (map (compress d)) ∘
        (map to_int_mod_ring) ∘
        coeffs ∘
        of_qr"

definition decompress_poly :: "nat ⇒ 'a qr ⇒ 'a qr" where
  "decompress_poly d =
        to_qr ∘
        Poly ∘
        (map of_int_mod_ring) ∘
        (map (decompress d)) ∘
        (map to_int_mod_ring) ∘
        coeffs ∘
        of_qr"
```

Lemmas for compression error for polynomials. Lemma telescope to go qrom module level down to integer coefficients and back up again.

```
lemma of_int_mod_ring_eq_0:
  "((of_int_mod_ring x :: 'a mod_ring) = 0) ⟷
    (x mod q = 0)"
by (metis CARD_a mod_0 of_int_code(2)
  of_int_mod_ring.abs_eq of_int_mod_ring.rep_eq
  of_int_of_int_mod_ring)

lemma dropWhile_mod_ring:
  "dropWhile ((=)0) (map of_int_mod_ring xs :: 'a mod_ring list) =
   map of_int_mod_ring (dropWhile (λx. x mod q = 0) xs)"
proof (induct xs)
  case (Cons x xs)
  have "dropWhile ((=) 0) (map of_int_mod_ring (x # xs)) =
        dropWhile ((=) 0) ((of_int_mod_ring x :: 'a mod_ring) #
        (map of_int_mod_ring xs))"
    by auto
  also have "... = (if 0 = (of_int_mod_ring x :: 'a mod_ring)
    then dropWhile ((=) 0) (map of_int_mod_ring xs)
    else map of_int_mod_ring (x # xs))"
    unfolding dropWhile.simps(2)[of "((=) 0)"
      "of_int_mod_ring x :: 'a mod_ring" "map of_int_mod_ring xs"]
    by auto
  also have "... = (if x mod q = 0
    then map of_int_mod_ring (dropWhile (λx. x mod q = 0) xs)
    else map of_int_mod_ring (x # xs))"
```

```
    using of_int_mod_ring_eq_0 unfolding Cons.hyps by auto
  also have "... = map of_int_mod_ring (dropWhile (λx. x mod q = 0)
    (x # xs))"
    unfolding dropWhile.simps(2) by auto
  finally show ?case by blast
qed simp

lemma strip_while_mod_ring:
"(strip_while ((=) 0) (map of_int_mod_ring xs :: 'a mod_ring list)) =
  map of_int_mod_ring (strip_while (λx. x mod q = 0)  xs)"
unfolding strip_while_def comp_def rev_map dropWhile_mod_ring by auto


lemma of_qr_to_qr_Poly:
  assumes "length (xs :: int list) < Suc (nat n)"
          "xs ≠ []"
  shows "of_qr (to_qr
    (Poly (map (of_int_mod_ring :: int ⇒ 'a mod_ring) xs))) =
     Poly (map (of_int_mod_ring :: int ⇒ 'a mod_ring) xs)"
    (is "_ = ?Poly")
proof -
  have deg: "degree (?Poly) < n"
    using deg_Poly'[of "map of_int_mod_ring xs"] assms
    by (smt (verit, del_insts) One_nat_def Suc_pred degree_0
      length_greater_0_conv length_map less_Suc_eq_le
      order_less_le_trans zless_nat_eq_int_zless)
  then show ?thesis
    using of_qr_to_qr[of "?Poly"] deg_mod_qr_poly[of "?Poly"]
      deg_qr_n by (smt (verit, best) of_nat_less_imp_less)
qed

lemma telescope_stripped:
  assumes "length (xs :: int list) < Suc (nat n)"
    "strip_while (λx. x mod q = 0) xs = xs"
    "set xs ⊆ {0..<q}"
  shows "(map to_int_mod_ring)
    (coeffs (of_qr (to_qr (Poly
    (map (of_int_mod_ring :: int ⇒ 'a mod_ring) xs))))) = xs"
proof (cases "xs = []")
case False
  have ge_zero: "0≤x" and lt_q:"x < int CARD ('a)"
    if "x∈set xs" for x
    using assms(3) CARD_a atLeastLessThan_iff that by auto
  have to_int_of_int: "map (to_int_mod_ring ∘
    (of_int_mod_ring :: int ⇒ 'a mod_ring)) xs = xs"
    using to_int_mod_ring_of_int_mod_ring[OF ge_zero lt_q]
    by (simp add: map_idI)
  show ?thesis using assms(2)
    of_qr_to_qr_Poly[OF assms(1) False]
```

```
    by (auto simp add: to_int_of_int strip_while_mod_ring)
qed (simp)

lemma map_to_of_mod_ring:
  assumes "set xs ⊆ {0..<q}"
  shows "map (to_int_mod_ring ∘
    (of_int_mod_ring :: int ⇒ 'a mod_ring)) xs = xs"
using assms by (induct xs) (simp_all add: CARD_a)

lemma telescope:
  assumes "length (xs :: int list) < Suc (nat n)"
    "set xs ⊆ {0..<q}"
  shows "(map to_int_mod_ring)
    (coeffs (of_qr (to_qr (Poly
    (map (of_int_mod_ring :: int ⇒ 'a mod_ring) xs))))) =
    strip_while (λx. x mod q = 0) xs"
proof (cases "xs = strip_while (λx. x mod q = 0) xs")
case True
  then show ?thesis using telescope_stripped assms
    by auto
next
case False
  let ?of_int = "(map (of_int_mod_ring ::
    int ⇒ 'a mod_ring) xs)"
  have "xs ≠ []" using False by auto
  then have "(map to_int_mod_ring)
    (coeffs (of_qr (to_qr (Poly ?of_int)))) =
    (map to_int_mod_ring) (coeffs (Poly ?of_int))"
    using of_qr_to_qr_Poly[OF assms(1)] by auto
  also have "... = (map to_int_mod_ring)
    (strip_while ((=) 0) ?of_int)"
    by auto
  also have "... = map (to_int_mod_ring ∘
    (of_int_mod_ring :: int ⇒ 'a mod_ring))
    (strip_while (λx. x mod q = 0) xs)"
    using strip_while_mod_ring by auto
  also have "... = strip_while (λx. x mod q = 0) xs"
  using assms(2) proof (induct xs rule: rev_induct)
  case (snoc y ys)
    let ?to_of_mod_ring = "to_int_mod_ring ∘
      (of_int_mod_ring :: int ⇒ 'a mod_ring)"
    have "map ?to_of_mod_ring
      (strip_while (λx. x mod q = 0) (ys @ [y])) =
      (if y mod q = 0
       then map ?to_of_mod_ring (strip_while (λx. x mod q = 0) ys)
       else map ?to_of_mod_ring ys @ [?to_of_mod_ring y])"
      by (subst strip_while_snoc) auto
    also have "... = (if y mod q = 0
        then strip_while (λx. x mod q = 0) ys
```

```
        else map ?to_of_mod_ring ys @ [?to_of_mod_ring y])"
    using snoc by fastforce
    also have "... = (if y mod q = 0
        then strip_while (λx. x mod q = 0) ys
        else ys @ [y])"
      using map_to_of_mod_ring[OF snoc(2)] by auto
    also have "... = strip_while (λx. x mod q = 0) (ys @ [y])"
      by auto
    finally show ?case .
  qed simp
  finally show ?thesis by auto
qed

lemma length_coeffs_of_qr:
  "length (coeffs (of_qr (x ::'a qr))) < Suc (nat n)"
proof (cases "x=0")
case False
  then have "of_qr x ≠ 0" by simp
  then show ?thesis
    using length_coeffs_degree[of "of_qr x"] deg_of_qr[of x]
    using deg_qr_n by fastforce
qed  (auto simp add: n_gt_zero)
end

lemma strip_while_change:
  assumes "⋀x. P x ⟶ S x" "⋀x. (¬ P x) ⟶ (¬ S x)"
  shows "strip_while P xs = strip_while S xs"
proof (induct xs rule: rev_induct)
case (snoc x xs)
  have "P x = S x" using assms[of x] by blast
  then show ?case by (simp add: snoc.hyps)
qed simp

lemma strip_while_change_subset:
  assumes "set xs ⊆ s"
    "∀x∈s. P x ⟶ S x"
    "∀x∈s. (¬ P x) ⟶ (¬ S x)"
  shows "strip_while P xs = strip_while S xs"
using assms(1) proof (induct xs rule: rev_induct)
case (snoc x xs)
  have "x∈s" using snoc(2) by simp
  then have "P x ⟶ S x" and "(¬ P x) ⟶ (¬ S x)"
    using assms(2) assms(3) by auto
  then have "P x = S x" by blast
  then show ?case
  using snoc.hyps snoc.prems by auto
qed simp
```

Estimate for decompress compress for polynomials. Using the inequality for integers, chain it up to the level of polynomials.

```
context kyber_spec
begin
lemma decompress_compress_poly:
  assumes "of_nat d < ⌈(log 2 q)::real⌉"
          "d>0"
  shows "let x' = decompress_poly d (compress_poly d x) in
    abs_infty_poly (x - x') ≤
    round ( real_of_int q / real_of_int (2^(d+1)) )"
proof -
  let ?x' = "decompress_poly d (compress_poly d x)"
  have "abs_infty_q (poly.coeff (of_qr (x - ?x')) xa)
       ≤ round (real_of_int q / real_of_int (2 ^ (d + 1)))"
  for xa
  proof -
    let ?telescope = "(λxs. (map to_int_mod_ring)
      (coeffs (of_qr (to_qr (Poly
      (map (of_int_mod_ring :: int ⇒ 'a mod_ring) xs))))))"
    define compress_x where
      "compress_x =  map (compress d ∘ to_int_mod_ring)
      (coeffs (of_qr x))"
    let ?to_Poly = "(λa. Poly (map ((of_int_mod_ring ::
      int ⇒ 'a mod_ring) ∘ decompress d) a))"
    have "abs_infty_q (poly.coeff (of_qr x) xa -
      poly.coeff (of_qr (to_qr (?to_Poly
      (?telescope compress_x)))) xa ) =
      abs_infty_q (poly.coeff (of_qr x) xa -
      poly.coeff (of_qr (to_qr (?to_Poly
      (strip_while (λx. x = 0) compress_x)))) xa )"
    proof (cases "x = 0")
    case True
      then have "compress_x = []"
        unfolding compress_x_def by auto
      then show ?thesis by simp
    next
    case False
      then have nonempty:"compress_x ≠ []"
        unfolding compress_x_def by simp
      have "length compress_x < Suc (nat n)"
         by (auto simp add: compress_x_def length_coeffs_of_qr)
      moreover have "set compress_x ⊆ {0..<q}"
      proof -
        have to: "to_int_mod_ring (s::'a mod_ring) ∈
            {0..q - 1}" for s
          using to_int_mod_ring_range by auto
        have "compress d (to_int_mod_ring (s::'a mod_ring)) ∈
          {0..<q}" for s
          using range_compress[OF to assms(1), of s]
          twod_lt_q[OF assms(1)]
```

```
    by (simp add: powr_realpow)
  then show ?thesis unfolding compress_x_def by auto
qed
ultimately have "?telescope compress_x =
  strip_while (λx. x mod q = 0) compress_x"
  by (intro telescope[of "compress_x"]) simp
moreover have "strip_while (λx. x mod q = 0) compress_x =
  strip_while (λx. x = 0) compress_x"
proof -
  have ‹compress d s = 0› if ‹compress d s mod q = 0› for s
  proof -
    from ‹int d < ⌈log 2 (real_of_int q)⌉› twod_lt_q [of d]
    have ‹2 ^ d < q›
      by (simp add: powr_realpow)
    with compress_less [of d s] have ‹compress d s < q›
      by simp
    then have ‹compress d s = compress d s mod q›
      by (simp add: compress_def)
    with that show ?thesis
      by simp
  qed
  then have right: "⋀s. compress d s mod q = 0 ⟶
    compress d s = 0" by simp
  have  "¬ (compress d s = 0)"
    if "¬ (compress d s mod q = 0)" for s
    using twod_lt_q compress_def that by force
  then have left: "⋀s. ¬ (compress d s mod q = 0) ⟶
    ¬ (compress d s = 0)" by simp
  have "strip_while (λx. x mod q = 0) compress_x =
    strip_while (λx. x mod q = 0) (map (compress d)
    (map to_int_mod_ring (coeffs (of_qr x))))"
    (is "_ = strip_while (λx. x mod q = 0)
      (map (compress d) ?rest)")
    unfolding compress_x_def by simp
  also have "... = map (compress d)
    (strip_while ((λy. y mod q = 0) ∘ compress d)
    (map to_int_mod_ring (coeffs (of_qr x))))"
    using strip_while_map[of "λy. y mod q = 0" "compress d"]
    by blast
  also have "... = map (compress d)
    (strip_while ((λy. y = 0) ∘ compress d)
    (map to_int_mod_ring (coeffs (of_qr x))))"
    by (smt (verit, best) comp_eq_dest_lhs left right
      strip_while_change)
  also have "... = strip_while (λx. x = 0)
    (map (compress d) ?rest)"
    using strip_while_map[of "λy. y = 0"
      "compress d", symmetric] by blast
  finally show ?thesis
```

```
      unfolding compress_x_def by auto
  qed
  ultimately show ?thesis by auto
qed
also have "... = abs_infty_q (poly.coeff (of_qr x) xa -
  poly.coeff (?to_Poly (strip_while (λx. x = 0) compress_x)) xa)"

proof (cases "?to_Poly (strip_while (λx. x = 0) compress_x) = 0")
case False
  have "degree (?to_Poly (strip_while (λx. x = 0) compress_x)) ≤
    length (map ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘
    decompress d) (strip_while (λx. x = 0) compress_x)) - 1"
    using deg_Poly'[OF False] .
  moreover have "length (map (of_int_mod_ring ∘ decompress d)
      (strip_while (λx. x = 0) compress_x)) ≤
      length (coeffs (of_qr x))"
    unfolding compress_x_def
    by (metis length_map length_strip_while_le)
  moreover have "length (coeffs (of_qr x)) - 1 < deg_qr TYPE('a)"
    using deg_of_qr degree_eq_length_coeffs by metis
  ultimately have deg:
    "degree (?to_Poly (strip_while (λx. x = 0) compress_x)) <
    deg_qr TYPE('a)" by auto
  show ?thesis using of_qr_to_qr'
    by (simp add: of_qr_to_qr'[OF deg])
qed simp
also have "... = abs_infty_q (poly.coeff (of_qr x) xa -
  poly.coeff (Poly (map of_int_mod_ring (strip_while (λx. x = 0)
    (map (decompress d) compress_x)))) xa )"
proof -
  have  "s = 0" if "decompress d s = 0" "s ∈ {0..2^d - 1}" for s
    using decompress_zero_unique[OF that assms(1)] .
  then have right: "∀s ∈ {0..2^d-1}. decompress d s = 0 ⟶
    s = 0" by simp
  have left: "∀ s ∈ {0..2^d-1}. decompress d s ≠ 0 ⟶ s ≠ 0"
    using decompress_zero[of d] by auto
  have compress_x_range: "set compress_x ⊆ {0..2^d - 1}"
    unfolding compress_x_def compress_def by auto
  have "map (decompress d) (strip_while (λx. x = 0) compress_x) =
    map (decompress d) (strip_while (λx. decompress d x = 0)
      compress_x)"
  using strip_while_change_subset[OF compress_x_range right left]

    by auto
  also have "... = strip_while (λx. x = 0)
    (map (decompress d) compress_x)"
  proof -
    have "(λx. x = 0) ∘ decompress d = (λx. decompress d x = 0)"
      using comp_def by blast
```

```
    then show ?thesis
      using strip_while_map[symmetric, of "decompress d"
        "λx. x=0" compress_x] by auto
  qed
  finally have "map (decompress d) (strip_while (λx. x = 0)
    compress_x) =  strip_while (λx. x = 0) (map (decompress d)
    compress_x)" by auto
  then show ?thesis by (metis map_map)
qed
also have "... = abs_infty_q (poly.coeff (of_qr x) xa -
  poly.coeff (Poly (map of_int_mod_ring (strip_while
  (λx. x mod q = 0) (map (decompress d) compress_x)))) xa )"
proof -
  have range: "set (map (decompress d) compress_x) ⊆ {0..<q}"
    unfolding compress_x_def compress_def
    using range_decompress[OF _ assms(1)] by auto
  have right: " ∀x∈{0..<q}. x = 0 ⟶ x mod q = 0" by auto
  have left: "∀x∈{0..<q}. ¬ x = 0 ⟶ ¬ x mod q = 0" by auto
  have "strip_while (λx. x = 0) (map (decompress d) compress_x) =
    strip_while (λx. x mod q = 0) (map (decompress d) compress_x)"
  using strip_while_change_subset[OF range right left] by auto
  then show ?thesis by auto
qed
also have "... = abs_infty_q (poly.coeff (of_qr x) xa -
  poly.coeff (Poly (map of_int_mod_ring
  (map (decompress d) compress_x))) xa )"
  by (metis Poly_coeffs coeffs_Poly strip_while_mod_ring)
also have "... = abs_infty_q (poly.coeff (of_qr x) xa -
  ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘ decompress d ∘
    compress d ∘ to_int_mod_ring) (poly.coeff (of_qr x) xa))"
using coeffs_Poly
proof (cases "xa < length (coeffs (?to_Poly  compress_x))")
case True
  have "poly.coeff (?to_Poly compress_x) xa =
        coeffs (?to_Poly compress_x) ! xa"
  using nth_coeffs_coeff[OF True] by simp
  also have "... = strip_while ((=) 0) (map (
    (of_int_mod_ring :: int ⇒ 'a mod_ring) ∘ decompress d)
    compress_x) ! xa"
    using coeffs_Poly by auto
  also have "... = (map ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘

    decompress d) compress_x) ! xa"
    using True by (metis coeffs_Poly nth_strip_while)
  also have "... = ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘
    decompress d ∘ compress d ∘ to_int_mod_ring)
    (coeffs (of_qr x) ! xa)"
    unfolding compress_x_def
    by (smt (z3) True coeffs_Poly compress_x_def length_map
```

```
        length_strip_while_le map_map not_less nth_map order_trans)
    also have "... = ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘
      decompress d ∘ compress d ∘ to_int_mod_ring)
      (poly.coeff (of_qr x) xa)"
      by (metis (no_types, lifting) True coeffs_Poly compress_x_def

        length_map length_strip_while_le not_less nth_coeffs_coeff
        order.trans)
    finally have no_coeff: "poly.coeff (?to_Poly compress_x) xa =
        ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘ decompress d ∘
        compress d ∘ to_int_mod_ring) (poly.coeff (of_qr x) xa)"
      by auto
    show ?thesis unfolding compress_x_def
    by (metis compress_x_def map_map no_coeff)
  next
  case False
    then have "poly.coeff (?to_Poly compress_x) xa = 0"
      by (metis Poly_coeffs coeff_Poly_eq nth_default_def)
    moreover have "((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘
      decompress d ∘ compress d ∘ to_int_mod_ring)
      (poly.coeff (of_qr x) xa) = 0"
    proof (cases "poly.coeff (of_qr x) xa = 0")
    case True
      then show ?thesis using compress_zero decompress_zero
        by auto
    next
    case False
      then show ?thesis
      proof (subst ccontr, goal_cases)
      case 1
        then have "poly.coeff (?to_Poly compress_x) xa ≠ 0"
          by (subst coeff_Poly) (metis (no_types, lifting) comp_apply

            compress_x_def compress_zero decompress_zero map_map
            nth_default_coeffs_eq nth_default_map_eq
            of_int_mod_ring_hom.hom_zero to_int_mod_ring_hom.hom_zero)
        then show ?case using ‹poly.coeff (?to_Poly compress_x) xa
= 0›
          by auto
      qed auto
    qed
    ultimately show ?thesis by auto
  qed
  also have "... = abs_infty_q (
    ((of_int_mod_ring :: int ⇒ 'a mod_ring) ∘ decompress d ∘
      compress d ∘ to_int_mod_ring) (poly.coeff (of_qr x) xa) -
      poly.coeff (of_qr x) xa)"
    using abs_infty_q_minus by (metis minus_diff_eq)
  also have "... = |((decompress d ∘ compress d ∘ to_int_mod_ring)
```

```
      (poly.coeff (of_qr x) xa) -
       to_int_mod_ring (poly.coeff (of_qr x) xa)) mod+- q|"
      unfolding abs_infty_q_def
      using to_int_mod_ring_of_int_mod_ring
      by (smt (verit, best) CARD_a comp_apply mod_plus_minus_def
        of_int_diff of_int_mod_ring.rep_eq
        of_int_mod_ring_to_int_mod_ring of_int_of_int_mod_ring)
    also have "... ≤ round (real_of_int q / real_of_int (2 ^ (d + 1)))"

    proof -
      have range_to_int_mod_ring:
        "to_int_mod_ring (poly.coeff (of_qr x) xa) ∈ {0..<q}"
        using to_int_mod_ring_range by auto
      then show ?thesis
        unfolding abs_infty_q_def Let_def
        using decompress_compress[OF range_to_int_mod_ring assms]
        by simp
    qed
    finally have "abs_infty_q (poly.coeff (of_qr x) xa - poly.coeff
      (of_qr (to_qr (?to_Poly (?telescope compress_x)))) xa )
      ≤ round (real_of_int q / real_of_int (2 ^ (d + 1)))" by auto
    then show ?thesis unfolding compress_x_def decompress_poly_def
      compress_poly_def by (auto simp add: o_assoc)
  qed
  moreover have finite:
    "finite (range (abs_infty_q ∘ poly.coeff (of_qr (x - ?x'))))"
    by (metis finite_Max image_comp image_image)
  ultimately show ?thesis unfolding abs_infty_poly_def
    using Max_le_iff[OF finite] by auto
qed
```

More properties of compress and decompress, used for returning message at the end.

```
lemma compress_1:
  shows "compress 1 x ∈ {0,1}"
unfolding compress_def by auto

lemma compress_poly_1:
  shows "∀i. poly.coeff (of_qr (compress_poly 1 x)) i ∈ {0,1}"
proof -
  have "poly.coeff (of_qr (compress_poly 1 x)) i ∈ {0,1}"
    for i
  proof -
    have "set (map (compress 1)
      ((map to_int_mod_ring ∘ coeffs ∘ of_qr) x)) ⊆ {0,1}"
      using compress_1 by auto
    then have "set ((map (compress 1) ∘ map to_int_mod_ring ∘
      coeffs ∘ of_qr) x) ⊆ {0,1}"
      (is "set (?compressed_1) ⊆ _")
```

```
    by auto
  then have "set (map (of_int_mod_ring :: int ⇒ 'a mod_ring)
    ?compressed_1) ⊆ {0,1}"
    (is "set (?of_int_compressed_1)⊆_")
    by (smt (verit, best) imageE insert_iff of_int_mod_ring_hom.hom_zero

      of_int_mod_ring_to_int_mod_ring set_map singletonD subsetD subsetI

      to_int_mod_ring_hom.hom_one)
  then have "nth_default 0 (?of_int_compressed_1) i
     ∈ {0,1}"
     by (smt (verit, best) comp_apply compress_1 compress_zero
      insert_iff nth_default_map_eq of_int_mod_ring_hom.hom_zero
      of_int_mod_ring_to_int_mod_ring singleton_iff
      to_int_mod_ring_hom.hom_one)
  moreover have "Poly (?of_int_compressed_1)
    = Poly (?of_int_compressed_1) mod qr_poly"
  proof -
    have "degree (Poly (?of_int_compressed_1)) < deg_qr TYPE('a)"
    proof (cases "Poly ?of_int_compressed_1 ≠ 0")
    case True
      have "degree (Poly ?of_int_compressed_1) ≤
        length (map (of_int_mod_ring :: int ⇒ 'a mod_ring)
        ?compressed_1) - 1"
        using deg_Poly'[OF True] by simp
      also have "... = length ((coeffs ∘ of_qr) x) - 1"
        by simp
      also have "... < n" unfolding comp_def
        using length_coeffs_of_qr
        by (metis deg_qr_n deg_of_qr degree_eq_length_coeffs
        nat_int zless_nat_conj)
      finally have "degree (Poly ?of_int_compressed_1) < n"
      using True ‹int (length ((coeffs ∘ of_qr) x) - 1) < n›
        deg_Poly' by fastforce
      then show ?thesis using deg_qr_n by simp
    next
    case False
      then show ?thesis
      using deg_qr_pos by auto
    qed
    then show ?thesis
      using deg_mod_qr_poly[of "Poly (?of_int_compressed_1)",
        symmetric] by auto
  qed
  ultimately show ?thesis unfolding compress_poly_def comp_def
    using of_qr_to_qr[of "Poly (?of_int_compressed_1)"]
    by auto
qed
then show ?thesis by auto
```

```
qed
end

lemma of_int_mod_ring_mult:
  "of_int_mod_ring (a*b) = of_int_mod_ring a * of_int_mod_ring b"
unfolding of_int_mod_ring_def
by (metis (mono_tags, lifting) Rep_mod_ring_inverse mod_mult_eq
  of_int_mod_ring.rep_eq of_int_mod_ring_def times_mod_ring.rep_eq)

context kyber_spec
begin
lemma decompress_1:
  assumes "a∈{0,1}"
  shows "decompress 1 a = round(real_of_int q/2) * a"
unfolding decompress_def using assms by auto

lemma decompress_poly_1:
  assumes "∀i. poly.coeff (of_qr x) i ∈ {0,1}"
  shows "decompress_poly 1 x =
    to_module (round((real_of_int q)/2)) * x"
proof -
  have "poly.coeff (of_qr (decompress_poly 1 x)) i =
   poly.coeff (of_qr (to_module (round((real_of_int q)/2)) * x)) i"
  for i
  proof -
    have "set (map to_int_mod_ring (coeffs (of_qr x))) ⊆ {0,1}"
    (is "set (?int_coeffs) ⊆ _")
    proof -
      have "set (coeffs (of_qr x)) ⊆ {0,1}" using assms
      by (meson forall_coeffs_conv insert_iff subset_code(1))
      then show ?thesis by auto
    qed
    then have "map (decompress 1) (?int_coeffs) =
      map ((*) (round (real_of_int q/2))) (?int_coeffs)"
    proof (induct "?int_coeffs")
    case (Cons a xa)
      then show ?case using decompress_1
      by (meson map_eq_conv subsetD)
    qed simp
    then have "poly.coeff (of_qr (decompress_poly 1 x)) i =
      poly.coeff (of_qr (to_qr (Poly (map of_int_mod_ring
        (map (λa. round(real_of_int q/2) * a)
        (?int_coeffs)))))) i"
      unfolding decompress_poly_def comp_def by presburger
    also have "... = poly.coeff (of_qr (to_qr (Poly
        (map (λa. of_int_mod_ring ((round(real_of_int q/2)) * a))
        (?int_coeffs))))) i"
      using map_map[of of_int_mod_ring "((*) (round (real_of_int q/2)))"]
      by (smt (z3) map_eq_conv o_apply)
```

```
    also have "... = poly.coeff (of_qr (to_qr (Poly
        (map (λa. of_int_mod_ring (round(real_of_int q/2)) *
        of_int_mod_ring a) (?int_coeffs))))) i"
      by (simp add: of_int_mod_ring_mult[of "(round(real_of_int q/2))"])
    also have "... = poly.coeff (of_qr (to_qr (Poly
        (map (λa. of_int_mod_ring (round(real_of_int q/2)) * a)
        (map of_int_mod_ring (?int_coeffs)))))) i"
      using map_map[symmetric, of
        "(λa. of_int_mod_ring (round (real_of_int q/2)) * a ::'a mod_ring)"
        "of_int_mod_ring"] unfolding comp_def by presburger
    also have "... = poly.coeff (of_qr (to_qr
        (Polynomial.smult (of_int_mod_ring (round(real_of_int q/2)))
        (Poly (map of_int_mod_ring (?int_coeffs)))))) i"
      using smult_Poly[symmetric, of
        "(of_int_mod_ring (round (real_of_int q/2)))"]
      by metis
    also have "... = poly.coeff (of_qr ((to_module
      (round (real_of_int q/2)) *
      to_qr (Poly (map of_int_mod_ring (?int_coeffs)))))) i"
      unfolding to_module_def
      using to_qr_smult_to_module
        [of "of_int_mod_ring (round (real_of_int q/2))"]
      by metis
    also have "... = poly.coeff (of_qr
       (to_module (round (real_of_int q/2)) *
        to_qr (Poly (coeffs (of_qr x)))))i"
      by (subst map_map[of of_int_mod_ring to_int_mod_ring],
        unfold comp_def)(subst of_int_mod_ring_to_int_mod_ring, auto)
    also have "... = poly.coeff (of_qr
       (to_module (round (real_of_int q/2)) * x))i"
      by (subst Poly_coeffs) (subst to_qr_of_qr, simp)
    finally show ?thesis by auto
  qed
  then have eq: "of_qr (decompress_poly 1 x) =
    of_qr (to_module (round((real_of_int q)/2)) * x)"
    by (simp add: poly_eq_iff)
  show ?thesis using arg_cong[OF eq, of "to_qr"]
    to_qr_of_qr[of "decompress_poly 1 x"]
    to_qr_of_qr[of "to_module (round (real_of_int q/2)) * x"]
    by auto
qed
end
```

Compression and decompression for vectors.

```
definition map_vector ::
  "('b ⇒ 'c) ⇒ ('b, 'n) vec ⇒ ('c, 'n::finite) vec" where
  "map_vector f v = (χ i. f (vec_nth v i))"

context kyber_spec
```

begin

Compression and decompression of vectors in `ℤ_q[X]/(X^n+1)`.

```
definition compress_vec ::
  "nat ⇒ ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
  "compress_vec d = map_vector (compress_poly d)"

definition decompress_vec ::
  "nat ⇒ ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
  "decompress_vec d = map_vector (decompress_poly d)"

end

end
theory Crypto_Scheme

imports Kyber_spec
        Compress
        Abs_Qr

begin
```

7 $(1 - \delta)$-Correctness Proof of the Kyber Crypto Scheme

```
context kyber_spec
begin
```

In the following the key generation, encryption and decryption algorithms of Kyber are stated. Here, the variables have the meaning:

- A: matrix, part of Alices public key
- s: vector, Alices secret key
- t: is the key generated by Alice qrom A and s in `key_gen`
- r: Bobs "secret" key, randomly picked vector
- m: message bits, $m \in \{0,1\}^{256}$
- (u, v): encrypted message
- dt, du, dv: the compression parameters for t, u and v respectively. Notice that `0 < d < ⌈log_2 q⌉`. The d values are public knowledge.
- e, $e1$ and $e2$: error parameters to obscure the message. We need to make certain that an eavesdropper cannot distinguish the encrypted message qrom uniformly random input. Notice that e and $e1$ are vectors while $e2$ is a mere element in `ℤ_q[X]/(X^n+1)`.

definition *key_gen ::*
```
  "nat ⇒ (('a qr, 'k) vec, 'k) vec ⇒ ('a qr, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
"key_gen dt A s e = compress_vec dt (A *v s + e)"
```

definition *encrypt ::*
```
  "('a qr, 'k) vec ⇒ (('a qr, 'k) vec, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec ⇒ ('a qr) ⇒
    nat ⇒ nat ⇒ nat ⇒ 'a qr ⇒
   (('a qr, 'k) vec) * ('a qr)" where
"encrypt t A r e1 e2 dt du dv m =
  (compress_vec du ((transpose A) *v r + e1),
   compress_poly dv (scalar_product (decompress_vec dt t) r +
    e2 + to_module (round((real_of_int q)/2)) * m)) "
```

definition *decrypt ::*
```
  "('a qr, 'k) vec ⇒ ('a qr) ⇒ ('a qr, 'k) vec ⇒
  nat ⇒ nat ⇒ 'a qr" where
"decrypt u v s du dv = compress_poly 1 ((decompress_poly dv v) -
  scalar_product s (decompress_vec du u))"
```

Lifting a function to the quotient ring

fun *f_int_to_poly :: "(int ⇒ int) ⇒ ('a qr) ⇒ ('a qr)"* **where**
```
  "f_int_to_poly f =
        to_qr ∘
        Poly ∘
        (map of_int_mod_ring) ∘
        (map f) ∘
        (map to_int_mod_ring) ∘
        coeffs ∘
        of_qr"
```

Error of compression and decompression.

definition *compress_error_poly ::*
```
  "nat ⇒ 'a qr ⇒ 'a qr" where
"compress_error_poly d y =
  decompress_poly d (compress_poly d y) - y"
```

definition *compress_error_vec ::*
```
  "nat ⇒ ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
"compress_error_vec d y =
  decompress_vec d (compress_vec d y) - y"
```

Lemmas for scalar product

lemma *scalar_product_linear_left:*
```
  "scalar_product (a+b) c =
   scalar_product a c + scalar_product b (c :: ('a qr, 'k) vec)"
```
unfolding *scalar_product_def*

```
by auto (metis (no_types, lifting) distrib_right sum.cong sum.distrib)

lemma scalar_product_linear_right:
  "scalar_product a (b+c) =
   scalar_product a b + scalar_product a (c :: ('a qr, 'k) vec)"
unfolding scalar_product_def
by auto (metis (no_types, lifting) distrib_left sum.cong sum.distrib)

lemma scalar_product_assoc:
  "scalar_product (A *v s) (r :: ('a qr, 'k) vec ) =
   scalar_product s (r v* A)"
unfolding scalar_product_def matrix_vector_mult_def
  vector_matrix_mult_def
proof auto
  have "(∑i∈UNIV. (∑j∈UNIV. (vec_nth (vec_nth A i) j) *
      (vec_nth s j)) * (vec_nth r i)) =
    (∑i∈UNIV. (∑j∈UNIV. (vec_nth (vec_nth A i) j) *
      (vec_nth s j) * (vec_nth r i)))"
    by (simp add: sum_distrib_right)
  also have "... = (∑j∈UNIV. (∑i∈UNIV. (vec_nth (vec_nth A i) j) *

    (vec_nth s j) * (vec_nth r i)))"
    using sum.swap .
  also have "... = (∑j∈UNIV. (∑i∈UNIV. (vec_nth s j) *
    (vec_nth (vec_nth A i) j) * (vec_nth r i)))"
    by (metis (no_types, lifting) mult_commute_abs sum.cong)
  also have "... = (∑j∈UNIV. (vec_nth s j) *
    (∑i∈UNIV. (vec_nth (vec_nth A i) j) * (vec_nth r i)))"
    by (metis (no_types, lifting) mult.assoc sum.cong sum_distrib_left)
  finally show "(∑i∈UNIV. (∑j∈UNIV. (vec_nth (vec_nth A i) j) *
    (vec_nth s j)) * (vec_nth r i)) = (∑j∈UNIV. (vec_nth s j) *
    (∑i∈UNIV. (vec_nth r i) *  (vec_nth (vec_nth A i) j)))"
    by (simp add: mult.commute)
qed
```

Lemma about coeff Poly

```
lemma coeffs_in_coeff:
  assumes "∀i. poly.coeff x i ∈ A"
  shows "set (coeffs x) ⊆ A"
by (simp add: assms coeffs_def image_subsetI)

lemma set_coeff_Poly: "set ((coeffs ∘ Poly) xs) ⊆ set xs"
proof -
  have "x ∈ set (strip_while ((=) 0) xs) ⟹ x ∈ set xs"
    for x
    by (metis append.assoc append_Cons in_set_conv_decomp
      split_strip_while_append)
  then show ?thesis by auto
qed
```

We now want to show the deterministic correctness of the algorithm. That means, after choosing the variables correctly, generating the public key, encrypting and decrypting, we get back the original message.

```
lemma kyber_correct:
  fixes A s r e e1 e2 dt du dv ct cu cv t u v
  assumes
      t_def:   "t = key_gen dt A s e"
  and u_v_def: "(u,v) = encrypt t A r e1 e2 dt du dv m"
  and ct_def:  "ct = compress_error_vec dt (A *v s + e)"
  and cu_def:  "cu = compress_error_vec du
                ((transpose A) *v r + e1)"
  and cv_def:  "cv = compress_error_poly dv
               (scalar_product (decompress_vec dt t) r + e2 +
                to_module (round((real_of_int q)/2)) * m)"
  and delta:   "abs_infty_poly (scalar_product e r + e2 + cv -
               scalar_product s e1 + scalar_product ct r -
               scalar_product s cu) < round (real_of_int q / 4)"
  and m01:     "set ((coeffs ∘ of_qr) m) ⊆ {0,1}"
  shows "decrypt u v s du dv = m"
proof -
```

First, show that the calculations are performed correctly.

```
  have t_correct: "decompress_vec dt t = A *v s + e + ct "
    using t_def ct_def unfolding compress_error_vec_def
    key_gen_def by simp
  have u_correct: "decompress_vec du u =
    (transpose A) *v r + e1 + cu"
    using u_v_def cu_def unfolding encrypt_def
    compress_error_vec_def by simp
  have v_correct: "decompress_poly dv v =
    scalar_product (decompress_vec dt t) r + e2 +
    to_module (round((real_of_int q)/2)) * m + cv"
    using u_v_def cv_def unfolding encrypt_def
    compress_error_poly_def by simp
  have v_correct': "decompress_poly dv v =
    scalar_product (A *v s + e) r + e2 +
    to_module (round((real_of_int q)/2)) * m + cv +
    scalar_product ct r"
   using t_correct v_correct
    by (auto simp add: scalar_product_linear_left)
  let ?t = "decompress_vec dt t"
  let ?u = "decompress_vec du u"
  let ?v = "decompress_poly dv v"
```

Define w as the error term of the message encoding. Have $\|w\|_{\infty,q} < \lceil q/4 \rfloor$

```
  define w where "w = scalar_product e r + e2 + cv -
    scalar_product s e1 + scalar_product ct r -
    scalar_product s cu"
```

```
have w_length: "abs_infty_poly w < round (real_of_int q / 4)"
  unfolding w_def using delta by auto
moreover have "abs_infty_poly w = abs_infty_poly (-w)"
  unfolding abs_infty_poly_def
  using neg_mod_plus_minus[OF q_odd q_gt_zero]
  using abs_infty_q_def abs_infty_q_minus by auto
ultimately have minus_w_length:
  "abs_infty_poly (-w) < round (real_of_int q / 4)"
  by auto
have vsu: "?v - scalar_product s ?u =
      w + to_module (round((real_of_int q)/2)) * m"
  unfolding w_def by (auto simp add: u_correct v_correct'
  scalar_product_linear_left scalar_product_linear_right
  scalar_product_assoc)
```

Set m' as the actual result of the decryption. It remains to show that $m' = m$.

```
define m' where "m' = decrypt u v s du dv"
have coeffs_m': "∀i. poly.coeff (of_qr m') i ∈ {0,1}"
  unfolding m'_def decrypt_def using compress_poly_1 by auto
```

Show $\|v - s^T u - \lceil q/2 \rfloor m'\|_{\infty,q} \leq \lceil q/4 \rfloor$

```
have "abs_infty_poly (?v - scalar_product s ?u -
  to_module (round((real_of_int q)/2)) * m')
= abs_infty_poly (?v - scalar_product s ?u -
  decompress_poly 1 (compress_poly 1 (?v - scalar_product s ?u)))"
  by (auto simp flip: decompress_poly_1[of m', OF coeffs_m'])
     (simp add:m'_def decrypt_def)
also have "... ≤ round (real_of_int q / 4)"
  using decompress_compress_poly[of 1 "?v - scalar_product s ?u"]
    q_gt_two by fastforce
finally have "abs_infty_poly (?v - scalar_product s ?u -
  to_module (round((real_of_int q)/2)) * m') ≤
  round (real_of_int q / 4)"
  by auto
```

Show $\|\lceil q/2 \rfloor (m - m')\|_{\infty,q} < 2\lceil q/4 \rfloor$

```
then have "abs_infty_poly (w + to_module
  (round((real_of_int q)/2)) * m - to_module
  (round((real_of_int q)/2)) * m') ≤ round (real_of_int q / 4)"
  using vsu by auto
then have w_mm': "abs_infty_poly (w +
  to_module (round((real_of_int q)/2)) * (m - m'))
  ≤ round (real_of_int q / 4)"
  by (smt (verit) add_uminus_conv_diff is_num_normalize(1)
    right_diff_distrib')
have "abs_infty_poly (to_module
    (round((real_of_int q)/2)) * (m - m')) =
  abs_infty_poly (w + to_module
```

```
      (round((real_of_int q)/2)) * (m - m') - w)"
    by auto
  also have "... ≤ abs_infty_poly
    (w + to_module (round((real_of_int q)/2)) * (m - m'))
     + abs_infty_poly (- w)"
    using abs_infty_poly_triangle_ineq[of
      "w+to_module (round((real_of_int q)/2)) * (m - m')" "-w"]
    by auto
  also have "... < 2 * round (real_of_int q / 4)"
    using w_mm' minus_w_length by auto
  finally have error_lt: "abs_infty_poly (to_module (round((real_of_int
q)/2)) * (m - m')) <
    2 * round (real_of_int q / 4)"
    by auto
```

Finally show that $m - m'$ is small enough, ie that it is an integer smaller than one. Here, we need that $q \cong 1 \mod 4$.

```
  have coeffs_m':"set ((coeffs ∘ of_qr) m') ⊆ {0,1}"
  proof -
    have "compress 1 a ∈ {0,1}" for a
    unfolding compress_def by auto
    then have "poly.coeff (of_qr (compress_poly 1 a)) i ∈ {0,1}"
      for a i
      using compress_poly_1 by presburger
    then have "set (coeffs (of_qr (compress_poly 1 a))) ⊆ {0,1}"
      for a
      using coeffs_in_coeff[of "of_qr (compress_poly 1 a)" "{0,1}"]
      by simp
    then show ?thesis unfolding m'_def decrypt_def by simp
  qed
  have coeff_0pm1: "set ((coeffs ∘ of_qr) (m-m')) ⊆
    {of_int_mod_ring (-1),0,1}"
  proof -
    have "poly.coeff (of_qr m) i ∈ {0,1}"
      for i using m01 coeff_in_coeffs
      by (metis comp_def insertCI le_degree subset_iff
        zero_poly.rep_eq)
    moreover have "poly.coeff (of_qr m') i ∈ {0,1}" for i
      using coeffs_m' coeff_in_coeffs
      by (metis comp_def insertCI le_degree subset_iff zero_poly.rep_eq)
    ultimately have "poly.coeff (of_qr m - of_qr m') i ∈  {of_int_mod_ring
(- 1), 0, 1}" for i
      by (metis (no_types, lifting) coeff_diff diff_zero
        eq_iff_diff_eq_0 insert_iff of_int_hom.hom_one of_int_minus
        of_int_of_int_mod_ring singleton_iff verit_minus_simplify(3))
    then have "set (coeffs (of_qr m - of_qr m')) ⊆  {of_int_mod_ring
(- 1), 0, 1}"
      by (simp add: coeffs_in_coeff)
    then show ?thesis using m01 of_qr_diff[of m m'] by simp
```

```
  qed
  have "set ((coeffs ∘ of_qr) (m-m')) ⊆ {0}"
  proof (rule ccontr)
    assume "¬set ((coeffs ∘ of_qr) (m-m')) ⊆ {0}"
    then have "∃i. poly.coeff (of_qr (m-m')) i ∈
      {of_int_mod_ring (-1),1}"
      using coeff_0pm1
      by (smt (z3) coeff_in_coeffs comp_apply insert_iff
        leading_coeff_0_iff order_refl
        set_coeffs_subset_singleton_0_iff subsetD)
    then have error_ge: "abs_infty_poly (to_module
      (round((real_of_int q)/2)) * (m-m')) ≥
      2 * round (real_of_int q / 4)"
      using abs_infty_poly_ineq_pm_1 by simp
    show False using error_lt error_ge by simp
  qed
  then show ?thesis by (simp flip: m'_def) (metis to_qr_of_qr)
qed


end

end
theory Kyber_Values
  imports
    Crypto_Scheme


begin
```

8 Specification for Kyber

```
typedef fin7681 = "{0..<7681::int}"
  morphisms fin7681_rep fin7681_abs
  by (rule_tac x = 0 in exI, simp)

setup_lifting type_definition_fin7681


lemma CARD_fin7681 [simp]: "CARD (fin7681) = 7681"
  unfolding type_definition.card [OF type_definition_fin7681]
  by simp

lemma fin7681_nontriv [simp]: "1 < CARD(fin7681)"
  unfolding CARD_fin7681 by auto

lemma prime_7681: "prime (7681::nat)" by eval

instantiation fin7681 :: comm_ring_1
```

```
begin

lift_definition zero_fin7681 :: "fin7681" is "0" by simp

lift_definition one_fin7681 :: "fin7681" is "1" by simp

lift_definition plus_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"
  is "(λx y. (x+y) mod 7681)"
  by auto

lift_definition uminus_fin7681 :: "fin7681 ⇒ fin7681"
  is "(λx. (uminus x) mod 7681)"
  by auto

lift_definition minus_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"
  is "(λx y. (x-y) mod 7681)"
  by auto

lift_definition times_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"
  is "(λx y. (x*y) mod 7681)"
  by auto


instance
proof
  fix a b c ::fin7681
  show "a * b * c = a * (b * c)"
    by (transfer, simp add: algebra_simps mod_mult_left_eq mod_mult_right_eq)
  show "a + b + c = a + (b + c)"
    by (transfer, simp add: algebra_simps mod_add_left_eq mod_add_right_eq)
  show "(a + b) * c = a * c + b * c"
    by (transfer, simp add: algebra_simps mod_add_right_eq  mod_mult_right_eq)
qed (transfer; simp add: algebra_simps mod_add_right_eq; fail)+

end


instantiation fin7681 :: finite
begin
instance
  by standard (auto simp flip: type_definition.Abs_image [OF type_definition_fin7681])
end


instantiation fin7681 :: equal
begin
lift_definition equal_fin7681 :: "fin7681 ⇒ fin7681 ⇒ bool" is "(=)" .
instance by (intro_classes, transfer, auto)
end
```

```
instantiation fin7681 :: nontriv
begin
instance
proof
  show "1 < CARD(fin7681)" unfolding CARD_fin7681 by auto
qed
end

instantiation fin7681 :: prime_card
begin
instance
proof
  show "prime CARD(fin7681)" unfolding CARD_fin7681 using prime_7681
    by blast
qed
end

instantiation fin7681 :: qr_spec
begin

definition qr_poly'_fin7681:: "fin7681 itself ⇒ int poly" where
  "qr_poly'_fin7681 ≡ (λ_. Polynomial.monom (1::int) 256 + 1)"

instance proof
  have "lead_coeff (qr_poly' TYPE(fin7681)) = 1" unfolding qr_poly'_fin7681_def

    by (simp add: degree_add_eq_left degree_monom_eq)
  then show "¬ int CARD(fin7681) dvd
      lead_coeff (qr_poly' TYPE(fin7681))"
    unfolding CARD_fin7681 by auto
next
  have "degree (qr_poly' TYPE(fin7681)) = 256" unfolding qr_poly'_fin7681_def
    by (simp add: degree_add_eq_left degree_monom_eq)
  then show "0 < degree (qr_poly' TYPE(fin7681))"  by auto
qed
end

lift_definition to_int_fin7681 :: "fin7681 ⇒ int" is "λx. x" .

lift_definition of_int_fin7681 :: "int ⇒ fin7681" is "λx. (x mod 7681)"
  by simp

interpretation to_int_fin7681_hom: inj_zero_hom to_int_fin7681
  by (unfold_locales; transfer, auto)

interpretation of_int_fin7681_hom: zero_hom of_int_fin7681
  by (unfold_locales, transfer, auto)
```

```
lemma to_int_fin7681_of_int_fin7681 [simp]:
  "to_int_fin7681 (of_int_fin7681 x) = x mod 7681"
  using of_int_fin7681.rep_eq to_int_fin7681.rep_eq by presburger

lemma of_int_fin7681_to_int_fin7681 [simp]:
  "of_int_fin7681 (to_int_fin7681 x) = x"
  using fin7681_rep to_int_fin7681.rep_eq to_int_fin7681_hom.injectivity

    to_int_fin7681_of_int_fin7681 by force

lemma of_int_mod_ring_eq_iff [simp]:
  "(of_int_fin7681 a = of_int_fin7681 b) ⟷
   ((a mod 7681) = (b mod 7681))"
  by (metis of_int_fin7681.abs_eq of_int_fin7681.rep_eq)

interpretation kyber7681: kyber_spec 256 7681 3 8 "TYPE(fin7681)" "TYPE(3)"
proof (unfold_locales, goal_cases)
  case 4
  then show ?case using prime_7681 prime_int_numeral_eq by blast
next
  case 5
  then show ?case using CARD_fin7681 by auto
next
  case 7
  then show ?case unfolding qr_poly'_fin7681_def by auto
qed auto

end
theory Mod_Ring_Numeral
imports
  "Berlekamp_Zassenhaus.Poly_Mod"
  "Berlekamp_Zassenhaus.Poly_Mod_Finite_Field"
  "HOL-Library.Numeral_Type"

begin
```

9 Lemmas for Simplification of Modulo Equivalences

```
lemma to_int_mod_ring_of_int [simp]:
  "to_int_mod_ring (of_int n :: 'a :: nontriv mod_ring) = n mod int CARD('a)"
  by transfer auto

lemma to_int_mod_ring_of_nat [simp]:
  "to_int_mod_ring (of_nat n :: 'a :: nontriv mod_ring) = n mod CARD('a)"
  by transfer (auto simp: of_nat_mod)

lemma to_int_mod_ring_numeral [simp]:
```

"to_int_mod_ring (numeral n :: 'a :: nontriv mod_ring) = numeral n mod CARD('a)"
by *(metis of_nat_numeral to_int_mod_ring_of_nat)*

lemma *of_int_mod_ring_eq_iff [simp]:*
"((of_int a :: 'a :: nontriv mod_ring) = of_int b) ⟷ ((a mod CARD('a)) = (b mod CARD('a)))"
by *(metis to_int_mod_ring_hom.eq_iff to_int_mod_ring_of_int)*

lemma *of_nat_mod_ring_eq_iff [simp]:*
"((of_nat a :: 'a :: nontriv mod_ring) = of_nat b) ⟷ ((a mod CARD('a)) = (b mod CARD('a)))"
by *(metis of_nat_eq_iff to_int_mod_ring_hom.eq_iff to_int_mod_ring_of_nat)*

lemma *one_eq_numeral_mod_ring_iff [simp]:*
"(1 :: 'a :: nontriv mod_ring) = numeral a ⟷ (1 mod CARD('a)) = (numeral a mod CARD('a))"
using *of_nat_mod_ring_eq_iff[of 1 "numeral a",* **where** *?'a = 'a]*
by *(simp del: of_nat_mod_ring_eq_iff)*

lemma *numeral_eq_one_mod_ring_iff [simp]:*
"numeral a = (1 :: 'a :: nontriv mod_ring) ⟷ (numeral a mod CARD('a)) = (1 mod CARD('a))"
using *of_nat_mod_ring_eq_iff[of "numeral a" 1,* **where** *?'a = 'a]*
by *(simp del: of_nat_mod_ring_eq_iff)*

lemma *zero_eq_numeral_mod_ring_iff [simp]:*
"(0 :: 'a :: nontriv mod_ring) = numeral a ⟷ 0 = (numeral a mod CARD('a))"
using *of_nat_mod_ring_eq_iff[of 0 "numeral a",* **where** *?'a = 'a]*
by *(simp del: of_nat_mod_ring_eq_iff)*

lemma *numeral_eq_zero_mod_ring_iff [simp]:*
"numeral a = (0 :: 'a :: nontriv mod_ring) ⟷ (numeral a mod CARD('a)) = 0"
using *of_nat_mod_ring_eq_iff[of "numeral a" 0,* **where** *?'a = 'a]*
by *(simp del: of_nat_mod_ring_eq_iff)*

lemma *numeral_mod_ring_eq_iff [simp]:*
"((numeral a :: 'a :: nontriv mod_ring) = numeral b) ⟷ ((numeral a mod CARD('a)) = (numeral b mod CARD('a)))"
using *of_nat_mod_ring_eq_iff[of "numeral a" "numeral b",* **where** *?'a = 'a]*
by *(simp del: of_nat_mod_ring_eq_iff)*

instantiation *bit1 :: (finite) nontriv*
begin
instance proof
show *"1 < CARD('a bit1)"* **by** *simp*

```
qed
end
```

```
end
theory NTT_Scheme

imports Crypto_Scheme
  Mod_Ring_Numeral
  "Number_Theoretic_Transform.NTT"

begin
```

10 Number Theoretic Transform for Kyber

```
lemma Poly_strip_while:
"Poly (strip_while ((=) 0) x) = Poly x"
by (metis Poly_coeffs coeffs_Poly)
```

```
locale kyber_ntt = kyber_spec _ _ _ _"TYPE('a :: qr_spec)" "TYPE('k::finite)"
+
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
  and ω :: "('a::qr_spec) mod_ring"
  and μ :: "'a mod_ring"
  and ψ :: "'a mod_ring"
  and ψinv :: "'a mod_ring"
  and ninv :: "'a mod_ring"
  and mult_factor :: int
assumes
      omega_properties: "ω^n = 1" "ω ≠ 1" "(∀ m. ω^m = 1 ∧ m≠0 ⟶
m ≥ n)"
  and mu_properties: "μ * ω = 1" "μ ≠ 1"
  and psi_properties: "ψ^2 = ω" "ψ^n = -1"
  and psi_psiinv: "ψ * ψinv = 1"
  and n_ninv: "(of_int_mod_ring n) * ninv = 1"
  and q_split: "q = mult_factor * n + 1"
begin
```

Some properties of the roots ω and ψ and their inverses μ and $\psi_i nv$.

```
lemma mu_prop:
  "(∀ m. μ^m = 1 ∧ m≠0 ⟶ m ≥ n)"
by (metis mu_properties(1) mult.commute mult.right_neutral
  omega_properties(3) power_mult_distrib power_one)

lemma mu_prop':
```

```
assumes "μ^m' = 1" "m'≠0" shows "m' ≥ n"
using mu_prop  assms by blast

lemma omega_prop':
assumes "ω^m' = 1" "m'≠0" shows "m' ≥ n"
using omega_properties(3)  assms by blast

lemma psi_props:
shows "ψ^(2*n) = 1"
      "ψ^(n*(2*a+1)) = -1"
      "ψ≠1"
proof -
  show "ψ^(2*n) = 1"
  by (simp add: omega_properties(1) power_mult psi_properties)
  show "ψ^(n*(2*a+1)) = -1"
  by (metis (no_types, lifting) mult.commute mult_1 power_add
    power_minus1_even power_mult psi_properties(2))
  show "ψ≠1"
  using omega_properties(2) one_power2 psi_properties(1) by blast
qed

lemma psi_inv_exp:
"ψ^i * ψinv ^i = 1"
using left_right_inverse_power psi_psiinv by blast

lemma inv_psi_exp:
"ψinv^i * ψ ^i = 1"
by (simp add: mult.commute psi_inv_exp)


lemma negative_psi:
assumes "i<j"
shows "ψ^j * ψinv ^i = ψ^(j-i)"
proof -
  have j: "ψ^j = ψ^(j-i) * ψ^i" using assms
  by (metis add.commute le_add_diff_inverse nat_less_le power_add)
  show ?thesis unfolding j
  by (simp add: left_right_inverse_power psi_psiinv)
qed

lemma negative_psi':
assumes "i≤j"
shows "ψinv^i * ψ ^j = ψ^(j-i)"
proof -
  have j: "ψ^j = ψ^i * ψ^(j-i)" using assms
  by (metis le_add_diff_inverse power_add)
  show ?thesis unfolding j mult.assoc[symmetric] using inv_psi_exp[of
i] by simp
qed
```

```
lemma psiinv_prop:
shows "ψinv^2 = μ"
proof -
  show "ψinv^2 = μ"
  by (metis (mono_tags, lifting) mu_properties(1) mult.commute
    mult_cancel_right mult_cancel_right2 power_mult_distrib psi_properties(1)
psi_psiinv)
qed

lemma n_ninv':
"ninv * (of_int_mod_ring n) = 1"
using n_ninv
by (simp add: mult.commute)
```

The *map2* function for polynomials.

```
definition map2_poly :: "('a mod_ring ⇒ 'a mod_ring ⇒ 'a mod_ring) ⇒

    'a mod_ring poly ⇒ 'a mod_ring poly ⇒ 'a mod_ring poly" where
"map2_poly f p1 p2 =
  Poly (map2 f (map (poly.coeff p1) [0..<nat n]) (map (poly.coeff p2)
[0..<nat n]))"
```

Additional lemmas on polynomials.

```
lemma Poly_map_coeff:
assumes "degree f < num"
shows "Poly (map (poly.coeff (f)) [0..<num]) = f"
proof (subst poly_eq_iff, safe)
  fix j
  show "poly.coeff (Poly (map (poly.coeff f) [0..<num])) j = poly.coeff
f j"
  proof (cases "j<num")
    case True
    then show ?thesis
    unfolding coeff_Poly by (subst nth_default_nth, auto)
  next
    case False
    then have "j>degree f" using assms by auto
    then show ?thesis unfolding coeff_Poly using False
    by (simp add: coeff_eq_0 nth_default_beyond)
  qed
qed

lemma map_upto_n_mod:
"(Poly (map f [0..<n]) mod qr_poly) = (Poly (map f [0..<n]) :: 'a mod_ring
poly)"
proof -
  have "degree (Poly (map f [0::nat..<n])) < n"
  by (metis Suc_pred' deg_Poly' deg_qr_n deg_qr_pos degree_0 diff_zero
```

```
le_imp_less_Suc
    length_map length_upt nat_int)
  then show ?thesis
  by (subst deg_mod_qr_poly, use deg_qr_n in ‹auto›)
qed
```

```
lemma coeff_of_qr_zero:
assumes "i≥n"
shows "poly.coeff (of_qr (f :: 'a qr)) i = 0"
proof -
  have "degree (of_qr f) < i"
    using deg_of_qr deg_qr_n assms order_less_le_trans by auto
  then show ?thesis by (subst coeff_eq_0, auto)
qed
```

Definition of NTT on polynomials. In contrast to the ordinary NTT, we use a different exponent on the root of unity ψ.

```
definition ntt_coeff_poly :: "'a qr ⇒ nat ⇒ 'a mod_ring" where
  "ntt_coeff_poly g i = (∑j∈{0..<n}. (poly.coeff (of_qr g) j) * ψ^(j
* (2*i+1)))"
```

```
definition ntt_coeffs :: "'a qr ⇒ 'a mod_ring list" where
  "ntt_coeffs g = map (ntt_coeff_poly g) [0..<n]"
```

```
definition ntt_poly :: "'a qr ⇒ 'a qr" where
"ntt_poly g = to_qr (Poly (ntt_coeffs g))"
```

Definition of inverse NTT on polynomials. The inverse transformed is already scaled such that it is the true inverse of the NTT.

```
definition inv_ntt_coeff_poly :: "'a qr ⇒ nat ⇒ 'a mod_ring" where
  "inv_ntt_coeff_poly g' i = ninv *
    (∑j∈{0..<n}. (poly.coeff (of_qr g') j) * ψinv^(i*(2*j+1)))"
```

```
definition inv_ntt_coeffs :: "'a qr ⇒ 'a mod_ring list" where
  "inv_ntt_coeffs g' = map (inv_ntt_coeff_poly g') [0..<n]"
```

```
definition inv_ntt_poly :: "'a qr ⇒ 'a qr" where
  "inv_ntt_poly g = to_qr (Poly (inv_ntt_coeffs g))"
```

Kyber is indeed in the NTT-domain with root of unity ω. Note, that our ntt on polynomials uses a slightly different exponent. The root of unity ω defines an alternative NTT in Kyber.

Have $7681 = 30 * 256 + 1$ and $3329 = 13 * 256 + 1$.

```
interpretation kyber_ntt: ntt "nat q" "nat n" "nat mult_factor" ω μ
proof (unfold_locales, goal_cases)
  case 2
```

```
  then show ?case  using q_gt_two by linarith
next
  case 3
  then show ?case
    by (smt (verit, del_insts) int_nat_eq mult.commute nat_int_add
    nat_mult_distrib of_nat_1 q_gt_two q_split zadd_int_left)
next
  case 4
  then show ?case using n_gt_1 by linarith
qed (use CARD_a nat_int in ‹auto simp add: omega_properties mu_properties›)
```

Multiplication in of polynomials in R_q is a negacyclic convolution (because we factored by $x^n + 1$, thus $x^n \equiv -1 \mod x^n + 1$). This is the reason why we needed to adapt the exponent in the NTT.

```
definition qr_mult_coeffs :: "'a qr ⇒ 'a qr ⇒ 'a qr" (infixl ‹⋆› 70) where
  "qr_mult_coeffs f g = to_qr (map2_poly (*) (of_qr f) (of_qr g))"
```

The definition of the exponentiation ^ only allows for natural exponents, thus we need to cheat a bit by introducing `conv_sign x`$\equiv (-1)^x$.

```
definition conv_sign :: "int ⇒ 'a mod_ring" where
"conv_sign x = (if x mod 2 = 0 then 1 else -1)"
```

The definition of the negacyclic convolution.

```
definition negacycl_conv :: "'a qr ⇒ 'a qr ⇒ 'a qr" where
"negacycl_conv f g =
  to_qr (Poly (map
  (λi. ∑j<n. conv_sign ((int i - int j) div n) *
    poly.coeff (of_qr f) j * poly.coeff (of_qr g) (nat ((int i - int j)
mod n)))
  [0..<n]))"
```

```
lemma negacycl_conv_mod_qr_poly:
"of_qr (negacycl_conv f g) mod qr_poly = of_qr (negacycl_conv f g)"
unfolding negacycl_conv_def of_qr_to_qr by auto
```

Representation of f modulo `qr_poly`.

```
lemma mod_div_qr_poly:
"(f :: 'a mod_ring poly) = (f mod qr_poly) + qr_poly * (f div qr_poly)"
by simp
```

`take_deg` returns the first n coefficients of a polynomial.

```
definition take_deg :: "nat ⇒ ('b::zero) poly ⇒ 'b poly"  where
"take_deg = (λn. λf. Poly (take n (coeffs f)))"
```

`drop_deg` returns the coefficients of a polynomial strarting from the n-th coefficient.

```
definition drop_deg :: "nat ⇒ ('b::zero) poly ⇒ 'b poly"  where
```

```
"drop_deg = (λn. λf. Poly (drop n (coeffs f)))"
```

take_deg and *drop_deg* return the modulo and divisor representants.

```
lemma take_deg_monom_drop_deg:
assumes "degree f ≥ n"
shows "(f :: 'a mod_ring poly) = take_deg n f + (Polynomial.monom 1 n)
* drop_deg n f"
proof -
  have "min (length (coeffs f)) n = n" using assms
  by (metis bot_nat_0.not_eq_extremum degree_0 le_imp_less_Suc
    length_coeffs_degree min.absorb1 min.absorb4)
  then show ?thesis
    unfolding take_deg_def drop_deg_def
    apply (subst Poly_coeffs[of f,symmetric])
    apply (subst append_take_drop_id[of n "coeffs f", symmetric])
    apply (subst Poly_append)
    by (auto)
qed

lemma split_mod_qr_poly:
assumes "degree f ≥ n"
shows "(f :: 'a mod_ring poly) = take_deg n f - drop_deg n f + qr_poly
* drop_deg n f"
proof -
  have "(Polynomial.monom 1 n + 1) * drop_deg n f =
    Polynomial.monom 1 n *  drop_deg n f + drop_deg n f"
    by (simp add: mult_poly_add_left)
  then show ?thesis
    apply (subst take_deg_monom_drop_deg[OF assms])
    apply (unfold qr_poly_def qr_poly'_eq of_int_hom.map_poly_hom_add)

    by auto
qed
```

Lemmas on the degrees of *take_deg* and *drop_deg*.

```
lemma degree_drop_n:
"degree (drop_deg n f) = degree f - n"
unfolding drop_deg_def
by (simp add: degree_eq_length_coeffs)

lemma degree_drop_2n:
assumes "degree f < 2*n"
shows "degree (drop_deg n f) < n"
using assms unfolding degree_drop_n by auto

lemma degree_take_n:
"degree (take_deg n f) < n"
unfolding take_deg_def
by (metis coeff_Poly_eq deg_qr_n deg_qr_pos degree_0 leading_coeff_0_iff
```

```
  nth_default_take of_nat_eq_iff)

lemma deg_mult_of_qr:
"degree (of_qr (f ::'a qr) * of_qr g) < 2 * n"
by (metis add_less_mono deg_of_qr deg_qr_n degree_0 degree_mult_eq
  mult_2 mult_eq_0_iff nat_int_comparison(1))
```

Representation of a polynomial modulo *qr_poly* using *take_deg* and *drop_deg*.

```
lemma mod_qr_poly:
assumes "degree f ≥ n" "degree f < 2*n"
shows "(f :: 'a mod_ring poly) mod qr_poly = take_deg n f - drop_deg n
f "
proof -
  have "degree (take_deg n f - drop_deg n f) < deg_qr TYPE('a)"
    using degree_diff_le_max[of "take_deg n f" "drop_deg n f"]
     degree_drop_2n[OF assms(2)]  degree_take_n
     by (metis deg_qr_n degree_diff_less nat_int)
  then have "(take_deg n f - drop_deg n f) mod qr_poly =
    take_deg n f - drop_deg n f" by (subst deg_mod_qr_poly, auto)
  then show ?thesis
    by (subst split_mod_qr_poly[OF assms(1)], auto)
qed
```

Coefficients of *take_deg*, *drop_deg* and the modulo representant.

```
lemma coeff_take_deg:
assumes "i<n"
shows "poly.coeff (take_deg n f) i = poly.coeff (f::'a mod_ring poly)
i"
using assms unfolding take_deg_def
by (simp add: nth_default_coeffs_eq nth_default_take)

lemma coeff_drop_deg:
assumes "i<n"
shows "poly.coeff (drop_deg n f) i = poly.coeff (f::'a mod_ring poly)
(i+n)"
using assms unfolding drop_deg_def
by (simp add: nth_default_coeffs_eq nth_default_drop)

lemma coeff_mod_qr_poly:
assumes "degree (f::'a mod_ring poly) ≥ n" "degree f < 2*n" "i<n"
shows "poly.coeff (f mod qr_poly) i = poly.coeff f i - poly.coeff f (i+n)"
apply (subst mod_qr_poly[OF assms(1) assms(2)])
apply (subst coeff_diff)
apply (unfold coeff_take_deg[OF assms(3)] coeff_drop_deg[OF assms(3)])
by auto
```

More lemmas on the splitting of sums.

```
lemma sum_leq_split:
```

```
"(∑ia≤i+n. f ia) = (∑ia<n. f ia) + (∑ia∈{n..i+n}. f ia)"
proof -
  have *: "{..i + n} - {..<n} = {n..i + n}"
  by (metis atLeastLessThanSuc_atLeastAtMost lessThan_Suc_atMost lessThan_minus_lessThan)

  show ?thesis
  by (subst sum.subset_diff[of "{..<n}" "{..i+n}" f]) (auto simp add:
* add.commute)
qed

lemma less_diff:
assumes "l1<l2"
shows "{..<l2} - {..l1} = {l1<..<l2::nat}"
by (metis atLeastSucLessThan_greaterThanLessThan lessThan_Suc_atMost lessThan_minus_lessTha

lemma sum_less_split:
assumes "l1<(l2::nat)"
shows "sum f {..<l2} = sum f {..l1} + sum f {l1<..<l2}"
by (subst sum.subset_diff[of "{..l1}" "{..<l2}" f])
   (auto simp add: assms add.commute order_le_less_trans less_diff[OF
assms])

lemma div_minus_1:
assumes "(x::int) ∈ {-b..<0}"
shows "x div b = -1"
using assms
by (smt (verit, ccfv_SIG) atLeastLessThan_iff div_minus_minus div_pos_neg_trivial)
```

A coefficient of polynomial multiplication is a coefficient of the negacyclic convolution.

```
lemma coeff_conv:
fixes f :: "'a qr"
assumes "i<n"
shows "poly.coeff ((of_qr f) * (of_qr g) mod qr_poly) i =
    (∑j<n. conv_sign ((int i - int j) div n) *
      poly.coeff (of_qr f) j * poly.coeff (of_qr g) (nat ((int i - int
j) mod n)))"
proof (cases "degree (of_qr f) + degree (of_qr g)<n")
  case True
  then have True':"degree ((of_qr f) * (of_qr g)) <n" using degree_mult_le

  using order_le_less_trans by blast
  have "poly.coeff ((of_qr f) * (of_qr g) mod qr_poly) i =
    poly.coeff ((of_qr f) * (of_qr g)) i" using True'
  by (metis deg_qr_n degree_qr_poly mod_poly_less nat_int)
  also have "... = (∑ia≤i. poly.coeff (of_qr f) ia * poly.coeff (of_qr
g) (i - ia))"
    unfolding coeff_mult by auto
  also have "... = (∑ia≤i. conv_sign ((int i - int ia) div int n) *
```

```
    poly.coeff (of_qr f) ia *
    poly.coeff (of_qr g) (nat ((int i - int ia) mod n)))"
  proof -
    have "i-ia = nat ((int i - int ia) mod n)" if "ia ≤ i" for ia
    using assms that by force
    moreover have "conv_sign ((int i - int ia) div int n) = 1"
      if "ia ≤ i" for ia unfolding conv_sign_def
      using assms that by force
    ultimately show ?thesis by auto
  qed
  also have "... = (∑ia<n. conv_sign ((int i - int ia) div int n) *
    poly.coeff (of_qr f) ia *
    poly.coeff (of_qr g) (nat ((int i - int ia) mod n)))"
  proof -
    have "poly.coeff (of_qr f) ia *
       poly.coeff (of_qr g) (nat ((int i - int ia) mod int n)) = 0"
      if "ia ∈ {i<..<n}" for ia
    proof (subst mult_eq_0_iff, safe, goal_cases)
      case 1
      have deg_g: "nat ((int i - int ia) mod int n) ≤ degree (of_qr g)"

        using le_degree[OF 1] by auto
      have "ia > degree (of_qr f)"
      proof (rule ccontr)
        assume "¬ degree (of_qr f) < ia"
        then have as: "ia ≤ degree (of_qr f)" by auto
        then have ni: "nat ((int i - int ia) mod int n) + ia = n + i"

          using that  by (smt (verit, ccfv_threshold) True deg_g
          greaterThanLessThan_iff int_nat_eq less_imp_of_nat_less
          mod_add_self1 mod_pos_pos_trivial of_nat_0_le_iff of_nat_add

          of_nat_mono)
        have "n + i ≤ degree (of_qr f) + degree (of_qr g)"
          unfolding ni[symmetric] using as deg_g by auto
        then show False using True by auto
      qed
      then show ?case
      using coeff_eq_0 by blast
    qed
    then show ?thesis
      by (subst sum_less_split[OF ‹i<n›]) (simp add: sum.neutral)
  qed
  finally show ?thesis by blast
next
  case False
  then have *: "degree (of_qr f * of_qr g) ≥ n"
  by (metis add.right_neutral add_0 deg_of_qr deg_qr_n degree_0 degree_mult_eq
```

```
      linorder_not_le nat_int)
  have "poly.coeff (of_qr f * of_qr g) (i + n) = (∑ia<n.
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (i + n - ia))"

    unfolding coeff_mult using coeff_of_qr_zero
    by (subst sum_leq_split[of _ i]) (auto)
  also have "... = (∑ia∈{i<..<n}.
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (i + n - ia))"

    using coeff_of_qr_zero by (subst sum_less_split[OF ‹i<n›]) auto
  also have "... = (∑ia∈{i<..<n}.
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (nat ((int i -
ia) mod n)))"
  proof -
    have "int i - int ia + int n ∈{0..<n}" if "ia ∈ {i<..<n}" for ia
using assms that by auto
    then have "int i + n-ia = (int i-ia) mod n" if "ia∈{i<..<n}" for ia
     using ‹i<n› that by (smt (verit, best) mod_add_self1 mod_rangeE)
    then have "i+n-ia = nat ((int i - ia) mod n)" if "ia∈{i<..<n}" for
ia
    by (metis int_minus nat_int of_nat_add that)
    then show ?thesis by fastforce
  qed
  also have "... = - (∑ia∈{i<..<n}. conv_sign ((int i - int ia) div n)
*
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (nat ((int i -
ia) mod n)))"
  proof -
    have negative:"(int i - int ia) ∈ {-n..<0}" if "ia ∈{i<..<n}" for
ia
      using that by auto
    have "(int i - int ia) div n = -1" if "ia ∈{i<..<n}" for ia
      using div_minus_1[OF negative[OF that]] .
    then have "conv_sign ((int i - int ia) div n) = -1" if "ia ∈{i<..<n}"
for ia
      unfolding conv_sign_def using that by auto
    then have *: "(∑ia∈{i<..<n}. foo ia) =
      (∑x∈{i<..<n}. - (conv_sign ((int i - int x) div int n) * foo x))"

    for foo by auto
    show ?thesis
    by (subst sum_negf[symmetric], subst *) (simp add: mult.assoc)
  qed
  finally have i_n: "poly.coeff (of_qr f * of_qr g) (i + n) =
    - (∑ia∈{i<..<n}. conv_sign ((int i - int ia) div n) *
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (nat ((int i -
ia) mod n)))"
    by blast
  have i_n': "poly.coeff (of_qr f * of_qr g) i =
```

```
      (∑ia≤i. conv_sign ((int i - int ia) div n) *
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (nat ((int i -
ia) mod n)))"
  proof -
    have "conv_sign ((int i - int ia) div n) = 1" if "ia ≤i" for ia
      using that assms conv_sign_def by force
    moreover have "i-ia ∈{0..<n}" if "ia ≤i" for ia using that assms
by auto
    then have "i-ia = (nat ((int i - ia) mod n))" if "ia≤i" for ia
      using assms that by force
    ultimately show ?thesis unfolding coeff_mult
    using assms less_imp_diff_less mod_less by auto
  qed
  have calc: "poly.coeff (of_qr f * of_qr g) i - poly.coeff (of_qr f *
of_qr g) (i + n) =
    (∑ia<n. conv_sign ((int i - int ia) div n) *
        poly.coeff (of_qr f) ia * poly.coeff (of_qr g) (nat ((int i -
ia) mod n)))"
    by (subst i_n, subst i_n')
       (metis (no_types, lifting) assms diff_minus_eq_add sum_less_split)
  show ?thesis unfolding coeff_mod_qr_poly[OF * deg_mult_of_qr assms]
calc
    by auto
qed
```

Polynomial multiplication in R_q is the negacyclic convolution.

```
lemma mult_negacycl:
"f * g = negacycl_conv f g"
proof -
  have f_times_g: "f * g = to_qr ((of_qr f) * (of_qr g) mod qr_poly)"
    by (metis of_qr_mult to_qr_of_qr)
  have conv: "poly.coeff ((of_qr f) * (of_qr g) mod qr_poly) i =
    (∑j<n. conv_sign ((int i - int j) div n) *
    poly.coeff (of_qr f) j * poly.coeff (of_qr g) (nat ((int i - j) mod
n)))"
  if "i<n" for i using coeff_conv[OF that] by auto
  have "poly.coeff (of_qr (f*g)) i =
    poly.coeff (of_qr (negacycl_conv f g)) i" for i
  proof (cases "i<n")
    case True
    then show ?thesis unfolding negacycl_conv_def f_times_g of_qr_to_qr

      map_upto_n_mod mod_mod_trivial coeff_Poly_eq
      using conv[OF True] by (subst nth_default_nth[of i], auto)
  next
    case False
    then show ?thesis using coeff_of_qr_zero[of i "f*g"]
      coeff_of_qr_zero[of i "negacycl_conv f g"] by auto
  qed
```

```
  then show ?thesis
    using poly_eq_iff [of "of_qr (f * g)" "of_qr (negacycl_conv f g)"]
    by (metis to_qr_of_qr)
qed
```

Additional lemmas on `ntt_coeffs`.

```
lemma length_ntt_coeffs:
"length (ntt_coeffs f) ≤ n"
unfolding ntt_coeffs_def by auto

lemma degree_Poly_ntt_coeffs:
"degree (Poly (ntt_coeffs f)) < n"
using length_ntt_coeffs
by (smt (verit) deg_Poly' degree_0 degree_take_n diff_diff_cancel
  diff_is_0_eq le_neq_implies_less less_nat_zero_code nat_le_linear
  order.strict_trans1 power_0_left power_eq_0_iff)

lemma Poly_ntt_coeffs_mod_qr_poly:
  "Poly (ntt_coeffs f) mod qr_poly = Poly (ntt_coeffs f)"
using map_upto_n_mod ntt_coeffs_def by presburger


lemma nth_default_map:
assumes "i<na"
shows "nth_default x (map f [0..<na]) i = f i"
using assms
by (simp add: nth_default_nth)


lemma nth_coeffs_negacycl:
assumes "j<n"
shows "poly.coeff (of_qr (negacycl_conv f g)) j =
  (∑i<n. conv_sign ((int j - int i) div int n) * poly.coeff (of_qr f)
i *
   poly.coeff (of_qr g) (nat ((int j - int i) mod int n)))"
unfolding negacycl_conv_def of_qr_to_qr map_upto_n_mod coeff_Poly_eq
 nth_default_map[OF assms] by auto
```

Writing the convolution sign as a conditional if statement.

```
lemma conv_sign_if:
assumes "x<n" "y<n"
shows "conv_sign ((int x - int y) div int n) = (if int x - int y < 0 then
-1 else 1)"
unfolding conv_sign_def
proof (split if_splits, safe, goal_cases)
  case 1
  then have "int x - int y ∈ {-n..<0}" using assms by simp
  then have "(int x - int y) div int n mod 2 = 1"
    using div_minus_1 by presburger
```

```
  then show ?case by auto
next
  case 2
  then have "(int x - int y) div int n mod 2 = 0"
  using assms(1) by force
  then show ?case by auto
qed
```

The convolution theorem on coefficients.

```
lemma ntt_coeff_poly_mult:
assumes "l<n"
shows "ntt_coeff_poly (f*g) l = ntt_coeff_poly f l * ntt_coeff_poly g
l"
proof -
  define f1 where "f1 = (λx. λ y.
        conv_sign ((int x - int y) div int n) *
        poly.coeff (of_qr f) y *
        poly.coeff (of_qr g) (nat ((int x - int y) mod int n)))"
  have "ntt_coeff_poly (f*g) l = (∑j = 0..<n. poly.coeff (of_qr (negacycl_conv
f g)) j *
        ψ^(j*(2*l+1)))" unfolding ntt_coeff_poly_def mult_negacycl by
auto
  also have "... = (∑j=0..<n. (∑i<n. f1 j i * ψ^(j*(2*l+1))))"
  proof (subst sum.cong[of "{0..<n}" "{0..<n}"
      "(λj. poly.coeff (of_qr (negacycl_conv f g)) j * ψ^(j*(2*l+1)))"
      "(λj. (∑i<n. f1 j i * ψ^(j*(2*l+1))))"],
      goal_cases)
    case (2 j)
    then have "j<n" by auto
    have "poly.coeff (of_qr (negacycl_conv f g)) j * ψ ^ (j * (2 * l
+ 1)) =
      (∑na<n. (conv_sign ((int j - int na) div int n) *
        poly.coeff (of_qr f) na * poly.coeff (of_qr g) (nat ((int j -
int na) mod int n))) *
        ψ ^ (j * (2 * l + 1)))"
      apply (subst nth_coeffs_negacycl[OF ‹j<n›])
      apply (subst sum_distrib_right)
      by auto
    also have "... = (∑na<n. f1 j na * ψ ^ (j * (2 * l + 1)))"
      unfolding f1_def by auto
    finally show ?case by blast
  qed auto
  also have "... = (∑i<n. ∑j<n. f1 j i * ψ ^ (j * (2 * l + 1))) "
    by (subst atLeast0LessThan, subst sum.swap, auto)
  also have "... = (∑i<n. poly.coeff (of_qr f) i * ψ ^ (i * (2 * l +
1)) *
    (∑j<n. poly.coeff (of_qr g) (nat ((int j - int i) mod int n)) *
        (if int j - int i < 0 then -1 else 1) *
         ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1))))"
```

```
  proof (subst sum.cong[of "{..<n}" "{..<n}" "(λi. (∑j<n. f1 j i * ψ
^ (j * (2 * l + 1))))"
    "(λi. poly.coeff (of_qr f) i * ψ ^ (i * (2 * l + 1)) *
        (∑j<n. poly.coeff (of_qr g) (nat ((int j - int i) mod int n))
*
        (if int j - int i < 0 then -1 else 1) *
         ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1))))"], goal_cases)
    case (2 i)
    then show ?case
    proof (subst sum_distrib_left, subst sum.cong[of "{..<n}" "{..<n}"

      "(λj. f1 j i * ψ ^ (j * (2 * l + 1)))"
      "(λj. poly.coeff (of_qr f) i * ψ ^ (i * (2 * l + 1)) *
        (poly.coeff (of_qr g) (nat ((int j - int i) mod int n)) *
        (if int j - int i < 0 then - 1 else 1) *
        ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1))))"], goal_cases)
      case (2 j)
      then have *: "conv_sign ((int j - int i) div int n) =
        (if int j - int i < 0 then - 1 else 1)" using conv_sign_if by
auto
      have "f1 j i * ψ ^ (j * (2 * l + 1)) =
        ψ ^ (i * (2 * l + 1)) * f1 j i * ψinv ^ (i * (2 * l + 1)) * ψ
^ (j * (2 * l + 1))"
        using psi_psiinv
        by (simp add: left_right_inverse_power)
      also have "... = poly.coeff (of_qr f) i * ψ ^ (i * (2 * l + 1))
*
      (poly.coeff (of_qr g) (nat ((int j - int i) mod int n)) *
      (if int j - int i < 0 then - 1 else 1) * ψinv ^ (i * (2 * l + 1))
* ψ ^ (j * (2 * l + 1)))"
        unfolding f1_def mult.assoc
        by (simp add: "*" mult.left_commute)
      finally show ?case  by blast
    qed auto
  qed auto
  also have "... = (∑i<n. poly.coeff (of_qr f) i * ψ ^ (i * (2 * l +
1)) *
    (∑x<n. poly.coeff (of_qr g) x * ψ ^ (x * (2 * l + 1))))"
  proof -
    define x' where "x' = (λj i. nat ((int j - int i) mod int n))"
    let ?if_inv = "(λi j. (if int j - int i < 0 then - 1 else 1) *
      ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1)))"
    have rewrite: "(if int j - int i < 0 then - 1 else 1) *
      ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1)) =
      ψ ^ ((x' j i) * (2 * l + 1))"  if "i<n" "j<n" for i j
    proof (cases "int j - int i <0")
      case True
      have lt: "i * (2 * l + 1) < n * (2 * l + 1)" using ‹i<n›
      by (metis One_nat_def add_gr_0 lessI mult_less_mono1)
```

```
      have "?if_inv i j = (-1) * ψinv ^ (i * (2 * l + 1)) * ψ ^ (j *
(2 * l + 1))"
        using True by (auto split: if_splits)
      also have "... = ψ^((n-i+j)* (2 * l + 1))" unfolding psi_props(2)[of
l, symmetric]
         negative_psi[OF lt]
         by (metis comm_semiring_class.distrib diff_mult_distrib power_add)
      also have "... = ψ ^ ((x' j i) * (2 * l + 1))" unfolding x'_def
      by (smt (verit, best) True mod_add_self2 mod_pos_pos_trivial nat_int_add

        nat_less_le of_nat_0_le_iff of_nat_diff that(1))
      finally show ?thesis by blast
    next
      case False
      then have "i≤j" by auto
      have lt: "i * (2 * l + 1) ≤ j * (2 * l + 1)" using ‹i≤j›
      using add_gr_0 less_one mult_less_mono1
      using mult_le_cancel2 by presburger
      have "?if_inv i j = ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l
+ 1))"
        using False by (auto split: if_splits)
      also have "... = ψ^((j-i)* (2 * l + 1))"
        using negative_psi'[OF lt]  diff_mult_distrib by presburger
      also have "... = ψ ^ ((x' j i) * (2 * l + 1))" unfolding x'_def
        by (metis ‹i ≤ j› less_imp_diff_less mod_pos_pos_trivial nat_int

          of_nat_0_le_iff of_nat_diff of_nat_less_iff that(2))
      finally show ?thesis by blast
    qed
    then have "(∑j<n. poly.coeff (of_qr g) (nat ((int j - int i) mod
int n)) *
      (if int j - int i < 0 then - 1 else 1) *
      ψinv ^ (i * (2 * l + 1)) * ψ ^ (j * (2 * l + 1))) =
      (∑x<n. poly.coeff (of_qr g) x * ψ ^ (x * (2 * l + 1)))"
      (is "(∑j<n. ?left j i) = _") if "i<n" for i
    proof -
      have *: "(∑j<n. ?left j i) =
        (∑j<n. poly.coeff (of_qr g) (x' j i) * ψ ^ ((x' j i) * (2 *
l + 1)))"
        using rewrite[OF that] x'_def
        by (smt (verit, ccfv_SIG) lessThan_iff mult.assoc sum.cong)
      have eq: "(λj. x' j i) ` {..<n} = {..<n}" unfolding x'_def
      proof (safe, goal_cases)
        case (1 _ j)
        with n_gt_zero show ?case
          by (simp add: nat_less_iff)
      next
        case (2 x)
        define j where "j = (x+i) mod n"
```

```
        have "j∈{..<n}"
        by (metis j_def lessThan_iff mod_less_divisor n_gt_zero of_nat_0_less_iff)
        moreover have "x = nat ((int j - int i) mod int n)" unfolding
j_def
        by (simp add: "2" mod_diff_cong zmod_int)
        ultimately show ?case by auto
      qed
      have inj: "inj_on (λj. x' j i) {..<n}" unfolding x'_def inj_on_def

      proof (safe, goal_cases)
        case (1 x y)
        then have "((int x - int i) mod int n) = ((int y - int i) mod
int n)"
          by (meson eq_nat_nat_iff mod_int_pos_iff n_gt_zero)
        then have "int x mod int n = int y mod int n"
          by (smt (z3) mod_diff_cong)
        then show ?case using 1 by auto
      qed
      show ?thesis unfolding * by (subst sum.reindex_cong[OF inj eq[symmetric],

        of "(λx. poly.coeff (of_qr g) x * ψ ^ (x * (2 * l + 1)))"
        "(λj. poly.coeff (of_qr g) (x' j i) * ψ ^ (x' j i * (2 * l + 1)))"],
auto)
    qed
    then show ?thesis by force
  qed
  also have "... = (∑i<n. poly.coeff (of_qr f) i * ψ ^ (i * (2 * l +
1))) *
    (∑x'<n. poly.coeff (of_qr g) x' * ψ ^ (x' * (2 * l + 1)))"
    unfolding sum_distrib_right by auto
  also have "... = ntt_coeff_poly f l * ntt_coeff_poly g l"
    unfolding ntt_coeff_poly_def atLeast0LessThan by auto
  finally show ?thesis by blast
qed


lemma ntt_coeffs_mult:
assumes "i<n"
shows "ntt_coeffs (f*g) !i = ntt_coeffs f! i * ntt_coeffs g ! i"
unfolding ntt_coeffs_def using ntt_coeff_poly_mult[OF assms]
by (simp add: assms)
```

Steps towards the convolution theorem.

```
lemma nth_default_ntt_coeff_mult:
"nth_default 0 (ntt_coeffs (f * g)) i =
 nth_default 0 (map2 (*)
  (map (poly.coeff (Poly (ntt_coeffs f))) [0..<nat (int n)])
  (map (poly.coeff (Poly (ntt_coeffs g))) [0..<nat (int n)])) i"
(is "?left i = ?right i")
```

```
proof (cases "i∈{0..<n}")
  case True
  then have l: "?left i = ntt_coeffs (f * g) ! i"
    by (simp add: nth_default_nth ntt_coeffs_def)
  have *: "?right i = (poly.coeff (Poly (ntt_coeffs f)) i) * (poly.coeff
(Poly (ntt_coeffs g)) i)"
    using True
    by (metis (no_types, lifting) coeff_Poly_eq diff_zero length_map length_upt

      map_nth_default mult_hom.hom_zero nat_int nth_default_map2 ntt_coeffs_def)
  then have r: "?right i = (ntt_coeffs f) ! i * (ntt_coeffs g) ! i"
    unfolding * unfolding coeff_Poly using nth_default_nth
    by (metis True atLeastLessThan_iff diff_zero length_map length_upt
ntt_coeffs_def)
  show ?thesis unfolding l r using ntt_coeffs_mult True by auto
next
  case False
  then have "?left i = 0" unfolding ntt_coeffs_def
    by (simp add: nth_default_beyond)
  moreover have "?right i = 0" using False
  by (simp add: nth_default_def)
  ultimately show ?thesis by presburger
qed

lemma Poly_ntt_coeffs_mult:
"Poly (ntt_coeffs (f * g)) = Poly (map2 (*)
  (map (poly.coeff (Poly (ntt_coeffs f))) [0..<nat (int n)])
  (map (poly.coeff (Poly (ntt_coeffs g))) [0..<nat (int n)]))"
apply (intro poly_eqI) apply (unfold coeff_Poly)
using nth_default_ntt_coeff_mult[of f g] by auto
```

Convolution theorem for NTT

```
lemma ntt_mult:
"ntt_poly (f * g) = qr_mult_coeffs (ntt_poly f) (ntt_poly g)"
proof -
  have "Poly (ntt_coeffs (f*g)) mod qr_poly =
    Poly (ntt_coeffs (f*g))"
    using Poly_ntt_coeffs_mod_qr_poly by force
  also have "... = Poly (coeffs (map2_poly (*) (Poly (ntt_coeffs f)) (Poly
(ntt_coeffs g))))"
    unfolding map2_poly_def coeffs_Poly Poly_strip_while
    using Poly_ntt_coeffs_mult by auto
  also have "... = (map2_poly (*) (of_qr (to_qr (Poly (ntt_coeffs f))))
        (of_qr (to_qr (Poly (ntt_coeffs g))))) mod qr_poly"
  unfolding of_qr_to_qr map_poly_def Poly_ntt_coeffs_mod_qr_poly
  by (metis Poly_coeffs Poly_ntt_coeffs_mod_qr_poly calculation)
  finally have "[Poly (ntt_coeffs (f * g)) =
      (map2_poly (*) (of_qr (to_qr (Poly (ntt_coeffs f))))
        (of_qr (to_qr (Poly (ntt_coeffs g)))))] (mod qr_poly)"
```

```
  using cong_def by blast
  then have "to_qr (Poly (ntt_coeffs (f * g))) =
    to_qr (map2_poly (*) (of_qr (to_qr (Poly (ntt_coeffs f))))
         (of_qr (to_qr (Poly (ntt_coeffs g)))))"
  using of_qr_to_qr by auto
  then show ?thesis
    unfolding ntt_poly_def qr_mult_coeffs_def
    by auto
qed
```

Correctness of NTT on polynomials.

```
lemma inv_ntt_poly_correct:
"inv_ntt_poly (ntt_poly f) = f"
proof -
  have rew_sum: "(∑j = 0..<n. nth_default 0
     (map (λi. ∑j = 0..<n. poly.coeff (of_qr f) j * ψ ^ (j * (2 * i +
1))) [0..<n])
      j * ψinv ^ (i * (2 * j + 1))) =
    (∑j = 0..<n. (∑j' = 0..<n. poly.coeff (of_qr f) j' * ψ ^ (j' *
(2 * j + 1)))
     * ψinv ^ (i * (2 * j + 1)))"
    (is "(∑j = 0..<n. ?left j) = (∑j = 0..<n. ?right j)") for i
  proof (subst sum.cong[of "{0..<n}" "{0..<n}" ?left ?right], goal_cases)
    case (2 x)
    then show ?case by (subst nth_default_map[of x n], auto)
  qed auto
  have  "(∑j = 0..<n. ∑j' = 0..<n. poly.coeff (of_qr f) j' *
    ψ ^ (j' * (2 * j + 1)) * ψinv ^ (i * (2 * j + 1))) =
    (of_int_mod_ring n) * poly.coeff (of_qr f) i" if "i<n" for i
  proof -
    have rew_psi: "ψ ^ (j * (2 * j' + 1)) * ψinv ^ (i * (2 * j' + 1))
=
      ψ ^ j * ψinv ^ i * (ψ ^ (j * 2) * ψinv ^ (i * 2)) ^ j'"
      if "j'<n" "j<n" for j' j
    by (smt (verit, ccfv_threshold) kyber_ntt.exp_rule mult.commute
      power_add power_mult power_one_right)
    have "(∑j = 0..<n. ∑j' = 0..<n. poly.coeff (of_qr f) j' *
        ψ ^ (j' * (2 * j + 1)) * ψinv ^ (i * (2 * j + 1))) =
      (∑j' = 0..<n. poly.coeff (of_qr f) j' * ψ^j' * ψinv^i *
        (∑j = 0..<n. (ψ ^ (j' * 2) * ψinv ^ (i * 2))^j))"
    apply (subst sum_distrib_left, subst sum.swap)
     proof (subst sum.cong[of "{0..<n}" "{0..<n}"
      "(λj. ∑ia = 0..<n. poly.coeff (of_qr f) j * ψ ^ (j * (2 * ia +
1)) *
          ψinv ^ (i * (2 * ia + 1)))"
      "(λj. ∑ia = 0..<n. poly.coeff (of_qr f) j * ψ ^ j * ψinv ^ i *
          (ψ ^ (j * 2) * ψinv ^ (i * 2)) ^ ia)"], goal_cases)
      case (2 j)
      then show ?case proof (subst sum.cong[of "{0..<n}" "{0..<n}"
```

```
      "(λia. poly.coeff (of_qr f) j * ψ ^ (j * (2 * ia + 1)) *
        ψinv ^ (i * (2 * ia + 1)))"
      "(λia. poly.coeff (of_qr f) j * ψ ^ j * ψinv ^ i *
        (ψ ^ (j * 2) * ψinv ^ (i * 2)) ^ ia)"], goal_cases)
        case (2 j')
        then show ?case using rew_psi[of j' j] by simp
      qed auto
    qed auto
    also have "... = (∑j' = 0..<n.
      (if j' = i then poly.coeff (of_qr f) j' * ψ^j' * ψinv^i * (of_int_mod_ring
n) else 0))"
    proof (subst sum.cong[of "{0..<n}" "{0..<n}"
      "(λj'. poly.coeff (of_qr f) j' * ψ ^ j' * ψinv ^ i *
        (∑j=0..<n. (ψ ^ (j' * 2) * ψinv ^ (i * 2))^j))"
      "(λj'. (if j' = i then poly.coeff (of_qr f) j' * ψ^j' * ψinv^i
*
        (of_int_mod_ring n) else 0))"], goal_cases)
      case (2 j')
      then show ?case proof (cases "j' = i")
        case True
        then have "(∑j=0..<n. (ψ ^ (j' * 2) * ψinv ^ (i * 2))^j) = of_int_mod_ring
n"
          unfolding True psi_inv_exp
          by (metis kyber_ntt.sum_rules(5) mult.right_neutral power_one
sum.cong)
        then show ?thesis using True by auto
      next
        case False
        have not1: "ψ ^ (j' * 2) * ψinv ^ (i * 2) ≠ 1"
        proof -
          have "ω^j' * μ ^i ≠ 1"
          proof (cases "j'<i")
            case True
            have *: "ω^j' * μ ^i = μ^(i-j')" using True
            by (metis (no_types, lifting) le_add_diff_inverse less_or_eq_imp_le

              mult.assoc mult_cancel_right2 power_add power_mult psi_inv_exp

              psi_properties(1) psiinv_prop)
            show ?thesis proof (unfold *, rule ccontr)
              assume "¬ μ ^ (i - j') ≠ 1"
              then have 1: "μ ^ (i - j') = 1" by auto
              show False using mu_prop'[OF 1] ‹j'≠i›
              using True less_imp_diff_less that diff_is_0_eq leD by blast
            qed
          next
            case False
            have 2: "ω^j' * μ ^i = ω^(j'-i)" using False
            by (smt (verit) Nat.add_diff_assoc ab_semigroup_mult_class.mult_ac(1)
```

```
            add_diff_cancel_left' left_right_inverse_power linorder_not_less

            mu_properties(1) mult.commute mult_numeral_1_right numeral_One
power_add)
          show ?thesis proof (unfold 2, rule ccontr)
            assume "¬ ω ^ (j' - i) ≠ 1"
            then have 1: "ω ^ (j' - i) = 1" by auto
            have "n > j' - i" using ‹j' ∈ {0..<n}› by auto
            then show False using omega_prop'[OF 1] ‹j'≠i›
            using False
            by (meson diff_is_0_eq leD order_le_imp_less_or_eq)
          qed
        qed
        then show ?thesis
        by (metis mult.commute power_mult psi_properties(1) psiinv_prop)

      qed
      have "(1 - ψ ^ (j' * 2) * ψinv ^ (i * 2)) *
        (∑j=0..<n. (ψ ^ (j' * 2) * ψinv ^ (i * 2))^j) = 0"
      proof (subst kyber_ntt.geo_sum, goal_cases)
        case 1
        then show ?case using not1 by auto
      next
        case 2
        then show ?case
        by (metis (no_types, opaque_lifting) cancel_comm_monoid_add_class.diff_cancel

          mu_properties(1) mult.commute omega_properties(1) power_mult
power_mult_distrib
          power_one psi_properties(1) psiinv_prop)
      qed
      then have "(∑j=0..<n. (ψ ^ (j' * 2) * ψinv ^ (i * 2))^j) = 0"

        using not1 by auto
      then show ?thesis using False by auto
    qed
  qed auto
  also have "... = poly.coeff (of_qr f) i * ψ^i * ψinv^i * (of_int_mod_ring
n)"
    by (subst sum.delta[of "{0..<n}" i], use ‹i<n› in auto)
  also have "... = (of_int_mod_ring n) * poly.coeff (of_qr f) i"
    by (simp add: psi_inv_exp)
  finally show ?thesis by blast
qed
then have rew_coeff: "(map (λi. ninv * (∑j = 0..<n. ∑n = 0..<n.
  poly.coeff (of_qr f) n * ψ ^ (n * (2 * j + 1)) * ψinv ^ (i * (2 *
j + 1)))) [0..<n]) =
  map (λi. ninv * (of_int_mod_ring (int n) * poly.coeff (of_qr f) i))
```

```
[0..<n]"
  unfolding map_eq_conv by auto
  show ?thesis unfolding inv_ntt_poly_def ntt_poly_def inv_ntt_coeffs_def
ntt_coeffs_def
    inv_ntt_coeff_poly_def ntt_coeff_poly_def of_qr_to_qr map_upto_n_mod
coeff_Poly
    apply (subst rew_sum)
    apply (subst sum_distrib_right)
    apply (subst rew_coeff)
    apply (subst mult.assoc[symmetric])
    apply (subst n_ninv')
    apply (subst mult_1)
    apply (subst Poly_map_coeff)
    subgoal using deg_of_qr deg_qr_n by fastforce
    subgoal unfolding to_qr_of_qr by auto
  done
qed


lemma ntt_inv_poly_correct:
"ntt_poly (inv_ntt_poly f) = f"
proof -
  have rew_sum: "(∑j = 0..<n. nth_default 0 (map (λi. ninv *
    (∑j' = 0..<n. poly.coeff (of_qr f) j' * ψinv ^ (i * (2 * j' + 1))))
[0..<n]) j *
      ψ ^ (j * (2 * i + 1))) =
    (∑j = 0..<n. ninv * (∑j' = 0..<n. poly.coeff (of_qr f) j' * ψinv
^ (j * (2 * j' + 1)))
     * ψ ^ (j * (2 * i + 1)))"
    (is "(∑j = 0..<n. ?left j) = (∑j = 0..<n. ?right j)") for i
  proof (subst sum.cong[of "{0..<n}" "{0..<n}" ?left ?right], goal_cases)
    case (2 x)
    then show ?case by (subst nth_default_map[of x n], auto)
  qed auto
  have  "(∑j = 0..<n. ∑n = 0..<n. ninv * (poly.coeff (of_qr f) n *
    ψinv ^ (j * (2 * n + 1))) * ψ ^ (j * (2 * i + 1))) =
    ninv * (of_int_mod_ring (int n) * poly.coeff (of_qr f) i)" if "i<n"
for i
  proof -
    have rew_psi: "ψinv ^ (j' * (2 * j + 1)) * ψ ^ (j' * (2 * i + 1))
=
      (ψinv ^ (j * 2) * ψ ^ (i * 2)) ^ j'"
      if "j'<n" "j<n" for j' j
    proof -
      have "ψinv ^ (j' * (2 * j + 1)) * ψ ^ (j' * (2 * i + 1)) =
        ψinv ^ (j' * (2 * j)) * ψ ^ (j' * (2 * i)) * ψinv ^ j'  * ψ ^
j' "
      by (simp add: power_add)
      also have "... = (ψinv ^ (2 * j) * ψ ^ (2 * i))^ j'"
```

```
    by (smt (verit, best) inv_psi_exp kyber_ntt.exp_rule mult.assoc

      mult.commute mult.right_neutral power_mult)
    also have "... = (ψinv ^ (j * 2) * ψ ^ (i * 2)) ^ j'"
    by (simp add: mult.commute)
    finally show ?thesis by blast
  qed
  have "(∑j = 0..<n. ∑j' = 0..<n. ninv * (poly.coeff (of_qr f) j'
*
  ψinv ^ (j * (2 * j' + 1))) * ψ ^ (j * (2 * i + 1))) =
    (∑j' = 0..<n. ninv * poly.coeff (of_qr f) j' *
      (∑j = 0..<n. (ψinv ^ (j' * 2) * ψ ^ (i * 2))^j))"
  apply (subst sum_distrib_left, subst sum.swap, unfold mult.assoc[symmetric])
  proof (subst sum.cong[of "{0..<n}" "{0..<n}"
    "(λj. ∑ia = 0..<n. ninv * poly.coeff (of_qr f) j * ψinv ^ (ia
* (2 * j + 1)) *
        ψ ^ (ia * (2 * i + 1)))"
    "(λj. ∑n = 0..<n. ninv * poly.coeff (of_qr f) j *
        (ψinv ^ (j * 2) * ψ ^ (i * 2)) ^ n)"], goal_cases)
    case (2 j)
    then show ?case
    proof (subst sum.cong[of "{0..<n}" "{0..<n}"
    "(λia. ninv * poly.coeff (of_qr f) j * ψinv ^ (ia * (2 * j + 1))
*
      ψ ^ (ia * (2 * i + 1)))"
    "(λia. ninv * poly.coeff (of_qr f) j *
      (ψinv ^ (j * 2) * ψ ^ (i * 2)) ^ ia)"], goal_cases)
      case (2 j')
      then show ?case using rew_psi[of j' j] by simp
    qed auto
  qed auto
  also have "... = (∑j' = 0..<n.
    (if j' = i then ninv * poly.coeff (of_qr f) j' *
      ψinv^j' * ψ^i * (of_int_mod_ring n) else 0))"
  (is "(∑j' = 0..<n. ?right j') = (∑j' = 0..<n. ?left j')")
  proof (subst sum.cong[of "{0..<n}" "{0..<n}" "?right" "?left"], goal_cases)
    case (2 j')
    then show ?case proof (cases "j' = i")
      case True
      then have "(∑j=0..<n. (ψinv ^ (j' * 2) * ψ ^ (i * 2))^j) = of_int_mod_ring
n"
        unfolding True psi_inv_exp
        by (metis kyber_ntt.sum_const mult.commute mult.right_neutral

        power_one psi_inv_exp sum.cong)
      then show ?thesis using True
      by (simp add: inv_psi_exp)
    next
      case False
```

```
have not1: "ψinv ^ (j' * 2) * ψ ^ (i * 2) ≠ 1"
proof -
  have "μ^j' * ω ^i ≠ 1"
  proof (cases "j'<i")
    case True
    have *: "μ^j' * ω ^i = ω^(i-j')" using True
    by (smt (verit, best) add.commute kyber_ntt.omega_properties(1)

      le_add_diff_inverse left_right_inverse_power less_or_eq_imp_le

      mu_properties(1) mult.left_commute mult_cancel_right1 power_add)
    show ?thesis proof (unfold *, rule ccontr)
      assume "¬ ω ^ (i - j') ≠ 1"
      then have 1: "ω ^ (i - j') = 1" by auto
      show False using omega_prop'[OF 1] ‹j'≠i›
      using True less_imp_diff_less that diff_is_0_eq leD by blast
    qed
  next
    case False
    have 2: "μ^j' * ω ^i = μ^(j'-i)" using False
    by (smt (verit) Nat.add_diff_assoc ab_semigroup_mult_class.mult_ac(1)

      add_diff_cancel_left' left_right_inverse_power linorder_not_less

      mu_properties(1) mult.commute mult_numeral_1_right numeral_One
power_add)
    show ?thesis proof (unfold 2, rule ccontr)
      assume "¬ μ ^ (j' - i) ≠ 1"
      then have 1: "μ ^ (j' - i) = 1" by auto
      have "n > j' - i" using ‹j' ∈ {0..<n}› by auto
      then show False using mu_prop'[OF 1] ‹j'≠i›
      using False
      by (meson diff_is_0_eq leD order_le_imp_less_or_eq)
    qed
  qed
  then show ?thesis
  by (metis mult.commute power_mult psi_properties(1) psiinv_prop)

qed
have "(1 - ψinv ^ (j' * 2) * ψ ^ (i * 2)) *
  (∑j=0..<n. (ψinv ^ (j' * 2) * ψ ^ (i * 2))^j) = 0"
proof (subst kyber_ntt.geo_sum, goal_cases)
  case 1
  then show ?case using not1 by auto
next
  case 2
  then show ?case
  by (metis (no_types, opaque_lifting) cancel_comm_monoid_add_class.diff_cancel
```

```
            mu_properties(1) mult.commute omega_properties(1) power_mult
power_mult_distrib
            power_one psi_properties(1) psiinv_prop)
        qed
        then have "(∑j=0..<n. (ψinv ^ (j' * 2) * ψ ^ (i * 2))^j) = 0"

          using not1 by auto
        then show ?thesis using False by auto
      qed
    qed auto
    also have "... = ninv * poly.coeff (of_qr f) i * ψinv^i * ψ^i * (of_int_mod_ring
n)"
      by (subst sum.delta[of "{0..<n}" i], use ‹i<n› in auto)
    also have "... = ninv * ((of_int_mod_ring n) * poly.coeff (of_qr f)
i)"
      by (simp add: psi_inv_exp mult.commute)
    finally show ?thesis by blast
  qed
  then have rew_coeff: "(map (λi. ∑j = 0..<n. ∑n = 0..<n. ninv * (poly.coeff
(of_qr f) n *
    ψinv ^ (j * (2 * n + 1))) * ψ ^ (j * (2 * i + 1))) [0..<n]) =
    map (λi. ninv * (of_int_mod_ring (int n) * poly.coeff (of_qr f) i))
[0..<n]"
  unfolding map_eq_conv by auto
  show ?thesis unfolding inv_ntt_poly_def ntt_poly_def inv_ntt_coeffs_def
ntt_coeffs_def
    inv_ntt_coeff_poly_def ntt_coeff_poly_def of_qr_to_qr map_upto_n_mod
coeff_Poly
    apply (subst rew_sum)
    apply (subst sum_distrib_left)
    apply (subst sum_distrib_right)
    apply (subst rew_coeff)
    apply (subst mult.assoc[symmetric])
    apply (subst n_ninv')
    apply (subst mult_1)
    apply (subst Poly_map_coeff)
    subgoal using deg_of_qr deg_qr_n by fastforce
    subgoal unfolding to_qr_of_qr by auto
  done
qed
```

The multiplication of two polynomials can be computed by the NTT.

```
lemma convolution_thm_ntt_poly:
  "f*g = inv_ntt_poly (qr_mult_coeffs (ntt_poly f) (ntt_poly g))"
unfolding ntt_mult[symmetric] inv_ntt_poly_correct by auto



end
```

end
theory *Crypto_Scheme_NTT*

imports *Crypto_Scheme*
NTT_Scheme

begin

11 Kyber Algorithm using NTT for Fast Multiplication

hide_type *Matrix.vec*

context *kyber_ntt*
begin

```
definition mult_ntt:: "'a qr ⇒ 'a qr ⇒ 'a qr" (infixl ‹*ntt› 70) where
  "mult_ntt f g = inv_ntt_poly (ntt_poly f ⋆ ntt_poly g)"

lemma mult_ntt:
  "f*g = f *ntt g"
  unfolding mult_ntt_def using convolution_thm_ntt_poly by auto

definition scalar_prod_ntt::
  "('a qr, 'k) vec ⇒ ('a qr, 'k) vec ⇒ 'a qr" (infixl ‹·ntt› 70) where
  "scalar_prod_ntt v w =
  (∑ i∈(UNIV::'k set). (vec_nth v i) *ntt (vec_nth w i))"

lemma scalar_prod_ntt:
  "scalar_product v w = scalar_prod_ntt v w"
  unfolding scalar_product_def scalar_prod_ntt_def using mult_ntt by
auto

definition mat_vec_mult_ntt::
  "(('a qr, 'k) vec, 'k) vec ⇒ ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" (infixl
‹·ntt› 70) where
  "mat_vec_mult_ntt A v = vec_lambda (λi.
  (∑ j∈UNIV. (vec_nth (vec_nth A i) j) *ntt (vec_nth v j)))"


lemma mat_vec_mult_ntt:
  "A *v v = mat_vec_mult_ntt A v"
  unfolding matrix_vector_mult_def mat_vec_mult_ntt_def using mult_ntt
by auto
```

Refined algorithm using NTT for multiplications

```
definition key_gen_ntt ::
```

```
  "nat ⇒ (('a qr, 'k) vec, 'k) vec ⇒ ('a qr, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
  "key_gen_ntt dt A s e = compress_vec dt (A ·ntt s + e)"

lemma key_gen_ntt:
  "key_gen_ntt dt A s e = key_gen dt A s e"
  unfolding key_gen_ntt_def key_gen_def mat_vec_mult_ntt by auto

definition encrypt_ntt ::
  "('a qr, 'k) vec ⇒ (('a qr, 'k) vec, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec ⇒ ('a qr) ⇒
    nat ⇒ nat ⇒ nat ⇒ 'a qr ⇒
   (('a qr, 'k) vec) * ('a qr)" where
  "encrypt_ntt t A r e1 e2 dt du dv m =
  (compress_vec du ((transpose A) ·ntt r + e1),
   compress_poly dv ((decompress_vec dt t) ·ntt r +
    e2 + to_module (round((real_of_int q)/2)) *ntt m)) "

lemma encrypt_ntt:
  "encrypt_ntt t A r e1 e2 dt du dv m = encrypt t A r e1 e2 dt du dv m"
  unfolding encrypt_ntt_def encrypt_def mat_vec_mult_ntt scalar_prod_ntt
mult_ntt by auto

definition decrypt_ntt ::
  "('a qr, 'k) vec ⇒ ('a qr) ⇒ ('a qr, 'k) vec ⇒
  nat ⇒ nat ⇒ 'a qr" where
  "decrypt_ntt u v s du dv = compress_poly 1 ((decompress_poly dv v) -

  s ·ntt (decompress_vec du u))"

lemma decrypt_ntt:
  "decrypt_ntt u v s du dv = decrypt u v s du dv"
  unfolding decrypt_ntt_def decrypt_def scalar_prod_ntt by auto
```

$(1-\delta)$-correctness for the refined algorithm

```
lemma kyber_correct_ntt:
  fixes A s r e e1 e2 dt du dv ct cu cv t u v
  assumes
      t_def:   "t = key_gen_ntt dt A s e"
  and u_v_def: "(u,v) = encrypt_ntt t A r e1 e2 dt du dv m"
  and ct_def:  "ct = compress_error_vec dt (A ·ntt s + e)"
  and cu_def:  "cu = compress_error_vec du
                ((transpose A) ·ntt r + e1)"
  and cv_def:  "cv = compress_error_poly dv
                ((decompress_vec dt t) ·ntt r + e2 +
                 to_module (round((real_of_int q)/2)) *ntt m)"
  and delta:   "abs_infty_poly (e ·ntt r + e2 + cv -
               s ·ntt e1 + ct ·ntt r -
               s ·ntt cu) < round (real_of_int q / 4)"
```

```
  and m01:     "set ((coeffs ∘ of_qr) m) ⊆ {0,1}"
  shows "decrypt_ntt u v s du dv = m"
using assms unfolding key_gen_ntt encrypt_ntt decrypt_ntt mat_vec_mult_ntt[symmetric]

scalar_prod_ntt[symmetric] mult_ntt[symmetric] using kyber_correct by
auto

end
end
theory Powers3844

imports Main Kyber_Values

begin
```

12 Checking Powers of Root of Unity

In order to check, that 3844 is indeed a root of unity, we need to calculate all powers and show that they are not equal to one.

```
fun fast_exp_7681 ::" int ⇒ nat ⇒ int" where
"fast_exp_7681 x 0 = 1" |
"fast_exp_7681 x (Suc e) = (x * (fast_exp_7681 x e)) mod 7681"

lemma list_all_fast_exp_7681:
  "list_all (λl. fast_exp_7681 (3844::int) l ≠ 1) [1..<256]"
  by eval

lemma fast_exp_7681_to_mod_ring:
"fast_exp_7681 x e = to_int_mod_ring ((of_int_mod_ring x :: fin7681 mod_ring)^e)"
proof (induct e arbitrary: x rule: fast_exp_7681.induct)
  case (2 x e)
  then show ?case
  by (metis (no_types, lifting) Suc_inject fast_exp_7681.elims kyber7681.module_spec_axioms

    module_spec.CARD_a nat.simps(3) of_int_mod_ring.rep_eq of_int_mod_ring_mult

    of_int_mod_ring_to_int_mod_ring power_Suc to_int_mod_ring.rep_eq)
qed auto

lemma fast_exp_7681_less256:
assumes "0<l" "l<256"
shows "fast_exp_7681 3844 l ≠ 1"
using list_all_fast_exp_7681 assms
by (smt (verit, ccfv_threshold) Ball_set One_nat_def atLeastLessThan_iff

  bot_nat_0.not_eq_extremum fast_exp_7681.elims less_Suc_numeral less_nat_zero_code

  not_less numeral_One numeral_less_iff set_upt)
```

```
lemma powr_less256:
assumes "0<l" "l<256"
shows "(3844::fin7681 mod_ring)^l ≠ 1"
using fast_exp_7681_less256[OF assms] unfolding fast_exp_7681_to_mod_ring
by (metis of_int_numeral of_int_of_int_mod_ring to_int_mod_ring_hom.hom_one)


end
theory Kyber_NTT_Values

imports Kyber_Values
  NTT_Scheme
  Powers3844

begin
```

13 Specification of Kyber with NTT

Calculations for NTT specifications

```
lemma "3844 * 6584 = (1 :: fin7681 mod_ring)"
  by simp

lemma "62 * 1115 = (1 :: fin7681 mod_ring)"
  by simp

lemma "256 * 7651 = (1:: fin7681 mod_ring)"
  by simp

lemma "7681 = 30 * 256 + (1::int)" by simp


lemma powr256:  "3844 ^ 256 = (1::fin7681 mod_ring)"
proof -
  have calc1: "3844^16 = (7154::fin7681 mod_ring)" by simp
  have calc2: "7154^16 = (1::fin7681 mod_ring)" by simp
  have "(3844::fin7681 mod_ring)^256 = (3844^16)^16"
    by (metis (mono_tags, opaque_lifting) num_double numeral_times_numeral
power_mult)
  also have "... = 1" unfolding calc1 calc2 by auto
  finally show ?thesis by blast
qed



lemma powr256':
  "62 ^ 256 = (- 1::fin7681 mod_ring)"
proof -
```

```
  have calc1: "62^16 = (1366::fin7681 mod_ring)" by simp
  have calc2: "1366^16 = (-1::fin7681 mod_ring)" by simp
  have "(62::fin7681 mod_ring)^256 = (62^16)^16"
    by (metis (mono_tags, opaque_lifting) num_double numeral_times_numeral
power_mult)
  also have "... = -1" unfolding calc1 calc2 by auto
  finally show ?thesis by blast
qed
```

```
interpretation kyber7681_ntt: kyber_ntt 256 7681 3 8
  "TYPE(fin7681)" "TYPE(3)" 3844 6584 62 1115 7651 30
proof (unfold_locales, goal_cases)
  case 4
  then show ?case using kyber7681.q_prime by fastforce
next
  case 6
  then show ?case using kyber7681.CARD_k by blast
next
  case 7
  then show ?case by (simp add: qr_poly'_fin7681_def)
next
  case 9
  then show ?case using powr256 by blast
next
  case 11
  then show ?case proof (safe, goal_cases)
    case (1 m)
    then show ?case using powr_less256[OF 1(2)]
      using linorder_not_less by blast
  qed
next
  case 15
  then show ?case using powr256' by blast
next
  case 17
  have mult: "256 * 7651 = (1::fin7681 mod_ring)" by simp
  have of_int: "of_int_mod_ring (int 256) = 256"
    by (metis o_def of_nat_numeral of_nat_of_int_mod_ring)
  show ?case unfolding of_int mult by simp
qed (auto)

end
```

References

[1] G. Alagic, D. A. Cooper, Q. Dang, T. Dang, J. M. Kelsey, J. Lichtinger, Y.-K. Liu, C. A. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and D. Apon. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process, 2022-07-05 04:07:00 2022.

[2] T. Ammer and K. Kreuzer. Number theoretic transform. *Archive of Formal Proofs*, August 2022. https://isa-afp.org/entries/Number_Theoretic_Transform.html, Formal proof development.

[3] R. M. Avanzi, J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation. 2017.

[4] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS — Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367, 2018.