

$(1 - \delta)$ -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	3
2	Type Class for Factorial Ring $\mathbb{Z}_q[x]/(x^n + 1)$.	4
3	Specification of Kyber	8
4	Re-centered Modulo Operation	11
5	Re-centered "Norm" Function	14
6	Compress and Decompress Functions	17
7	$(1 - \delta)$-Correctness Proof of the Kyber Crypto Scheme	24
8	Specification for Kyber	27
9	Lemmas for Simplification of Modulo Equivalences	29
10	Number Theoretic Transform for Kyber	30
11	Kyber Algorithm using NTT for Fast Multiplication	38
12	Checking Powers of Root of Unity	40
13	Specification of Kyber with NTT	41

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 $v = t^r + e_2 + \lceil q/2 \rceil m$ 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 requiring a $2n$ -th root of unity instead of a n -th root of unity. Moreover, the structure of R_q is negacyclic, since $x^n \equiv -1 \pmod{x^n + 1}$, 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 $q \equiv 1 \pmod{4}$ 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  $\longleftrightarrow$ 
   int (CARD('n)) dvd n"
  <proof>

```

```

lemma of_int_mod_ring_eq_of_int_iff:
  "(of_int n :: ('n :: {finite, nontriv} mod_ring)) = of_int m  $\longleftrightarrow$ 
   [n = m] (mod (int (CARD('n))))"
  <proof>

```

```

definition mod_poly_rel :: "nat  $\Rightarrow$  int poly  $\Rightarrow$  int poly  $\Rightarrow$  bool" where
  "mod_poly_rel m p q  $\longleftrightarrow$ 
   ( $\forall n$ . [poly.coeff p n = poly.coeff q n] (mod (int m)))"

```

```

lemma mod_poly_rel_altdef:
  "mod_poly_rel (CARD('n :: nontriv)) p q  $\longleftrightarrow$ 
   (of_int_poly p) = (of_int_poly q :: 'n mod_ring poly)"
  <proof>

```

```

definition mod_poly_is_unit :: "nat  $\Rightarrow$  int poly  $\Rightarrow$  bool" where
  "mod_poly_is_unit m p  $\longleftrightarrow$  ( $\exists r$ . mod_poly_rel m (p * r) 1)"

```

```

lemma mod_poly_is_unit_altdef:
  "mod_poly_is_unit (CARD('n :: nontriv)) p  $\longleftrightarrow$ 

```

```

    (of_int_poly p :: 'n mod_ring poly) dvd 1"
  <proof>

```

```

definition mod_poly_irreducible :: "nat  $\Rightarrow$  int poly  $\Rightarrow$  bool" where
  "mod_poly_irreducible m Q  $\longleftrightarrow$ 
     $\neg$ mod_poly_rel m Q 0  $\wedge$ 
     $\neg$ mod_poly_is_unit m Q  $\wedge$ 
    ( $\forall$  a b. mod_poly_rel m Q (a * b)  $\longrightarrow$ 
      mod_poly_is_unit m a  $\vee$  mod_poly_is_unit m b)"

```

```

lemma of_int_poly_to_int_poly: "of_int_poly (to_int_poly p) = p"
  <proof>

```

```

lemma mod_poly_irreducible_altdef:
  "mod_poly_irreducible CARD('n :: nontriv) p  $\longleftrightarrow$ 
    irreducible (of_int_poly p :: 'n mod_ring poly)"
  <proof>

```

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  $\Rightarrow$  int poly"
  assumes not_dvd_lead_coeff_qr_poly':
    " $\neg$ 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  $\Rightarrow$  nat" where
  "deg_qr _ = degree (qr_poly' TYPE('a))"

```

```

lemma degree_qr_poly':
  "degree (qr_poly' TYPE('a :: qr_spec)) = deg_qr (TYPE('a))"
  <proof>

```

```

lemma degree_of_int_poly':
  assumes "of_int (lead_coeff p)  $\neq$  (0 :: 'a :: ring_1)"
  shows "degree (of_int_poly p :: 'a poly) = degree p"
  <proof>

```

```

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

```

```

lemma deg_qr_pos : "deg_qr TYPE('a :: qr_spec) > 0"
  <proof>

```

The factor polynomial is non-zero.

```
lemma qr_poly_nz [simp]: "qr_poly ≠ 0"
  ⟨proof⟩
```

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⟩
```

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"
  ⟨proof⟩
```

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
  ⟨proof⟩
```

Defining the conversion functions.

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

```
lift_definition of_qr :: "'a qr ⇒ 'a :: qr_spec mod_ring poly"
  is "λP::'a mod_ring poly. P mod qr_poly"
  ⟨proof⟩
```

Simplification lemmas on conversion functions.

```
lemma of_qr_to_qr: "of_qr (to_qr (x)) = x mod qr_poly"
  ⟨proof⟩
```

```
lemma to_qr_of_qr: "to_qr (of_qr (x)) = x"
  ⟨proof⟩
```

```
lemma eq_to_qr: "x = y ⇒ to_qr x = to_qr y" ⟨proof⟩
```

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

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

```
lift_definition zero_qr :: "'a qr" is "0" ⟨proof⟩
```

```
lift_definition one_qr :: "'a qr" is "1" ⟨proof⟩
```

```

lift_definition plus_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(+)"
  ⟨proof⟩

lift_definition uminus_qr :: "'a qr ⇒ 'a qr"
  is "uminus"
  ⟨proof⟩

lift_definition minus_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(-)"
  ⟨proof⟩

lift_definition times_qr :: "'a qr ⇒ 'a qr ⇒ 'a qr"
  is "(*)"
  ⟨proof⟩

instance
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma to_qr_of_nat [simp]: "to_qr (of_nat n) = of_nat n"
  ⟨proof⟩

lemma to_qr_of_int [simp]: "to_qr (of_int n) = of_int n"
  ⟨proof⟩

lemma of_qr_of_nat [simp]: "of_qr (of_nat n) = of_nat n"
  ⟨proof⟩

lemma of_qr_of_int [simp]: "of_qr (of_int n) = of_int n"
  ⟨proof⟩

lemma of_qr_eq_0_iff [simp]: "of_qr p = 0 ⟷ p = 0"

```

<proof>

lemma *to_qr_eq_0_iff*:
"to_qr p = 0 \longleftrightarrow qr_poly dvd p"
<proof>

Some more lemmas that will probably be useful.

lemma *to_qr_eq_iff [simp]*:
"to_qr P = (to_qr Q :: 'a :: qr_spec qr) \longleftrightarrow [P = Q] (mod qr_poly)"
<proof>

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

lemma *inj_on_to_qr*:
"inj_on
 (to_qr :: 'a :: qr_spec mod_ring poly \Rightarrow 'a qr)
 {P. degree P < deg_qr TYPE('a)}"
<proof>

Characteristic of quotient ring is exactly q .

lemma *of_int_qr_eq_0_iff [simp]*:
"of_int n = (0 :: 'a :: qr_spec qr) \longleftrightarrow int (CARD('a)) dvd n"
<proof>

lemma *of_int_qr_eq_of_int_iff*:
"of_int n = (of_int m :: 'a :: qr_spec qr) \longleftrightarrow
 [n = m] (mod (int (CARD('a))))"
<proof>

lemma *of_nat_qr_eq_of_nat_iff*:
"of_nat n = (of_nat m :: 'a :: qr_spec qr) \longleftrightarrow
 [n = m] (mod CARD('a))"
<proof>

lemma *of_nat_qr_eq_0_iff [simp]*:
"of_nat n = (0 :: 'a :: qr_spec qr) \longleftrightarrow CARD('a) dvd n"
<proof>

3 Specification of Kyber

definition *to_module* :: "int \Rightarrow '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"
<proof>

```

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)"
<proof>

```

```

lemma deg_of_qr:
  "degree (of_qr (x :: 'a qr)) < deg_qr TYPE('a::qr_spec)"
<proof>

```

```

lemma to_qr_smult_to_module:
  "to_qr (Polynomial.smult a p) = (to_qr (Poly [a])) * (to_qr p)"
<proof>

```

```

lemma of_qr_to_qr_smult:
  "of_qr (to_qr (Polynomial.smult a p)) =
  Polynomial.smult a (of_qr (to_qr p))"
<proof>

```

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"
<proof>

```

```

lemma q_gt_zero: "q>0"
<proof>

```

```

lemma q_gt_two: "q>2"
<proof>

```

```

lemma q_odd: "odd q"
<proof>

lemma nat_q: "nat q = q"
<proof>

Some properties of the degree n.

lemma n_gt_1: "n > 1"
<proof>

lemma n_nonzero: "n ≠ 0"
<proof>

lemma n_gt_zero: "n>0"
<proof>

lemma nat_n: "nat n = n"
<proof>

lemma deg_qr_n:
  "deg_qr TYPE('a) = n"
<proof>

end

```

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

```

n   = 256 = 2n'
n'  = 8
q   = 7681 or 3329
k   = 3

```

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

```

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:

```

$\langle \lfloor \text{real_of_int } x / 2 \rfloor = (x - 1) \text{ div } 2 \rangle$ if $\langle \text{odd } x \rangle$
 $\langle \text{proof} \rangle$

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 $(-q \text{ div } 2, q \text{ div } 2]$. Using these representatives, we ensure that the norm of the representative is as small as possible.

definition *mod_plus_minus* :: "int \Rightarrow int \Rightarrow int"
 (infixl $\langle \text{mod}+- \rangle$ 70) **where**
 "m mod+- b =
 (if m mod b > $\lfloor b/2 \rfloor$ then m mod b - b else m mod b)"

Range of the (re-centered) modulo operation

lemma *mod_range*: "b>0 \implies (a::int) mod (b::int) \in {0..b-1}"
 $\langle \text{proof} \rangle$

lemma *mod_rangeE*:
 assumes "(a::int) \in {0..<b}"
 shows "a = a mod b"
 $\langle \text{proof} \rangle$

lemma *half_mod_odd*:
 assumes "b > 0" "odd b" " $\lfloor \text{real_of_int } b / 2 \rfloor < y \text{ mod } b$ "
 shows " $-\lfloor \text{real_of_int } b / 2 \rfloor \leq y \text{ mod } b - b$ "
 " $y \text{ mod } b - b \leq \lfloor \text{real_of_int } b / 2 \rfloor$ "
 $\langle \text{proof} \rangle$

lemma *half_mod*:
 assumes "b>0"
 shows " $-\lfloor \text{real_of_int } b / 2 \rfloor \leq y \text{ mod } b$ "
 $\langle \text{proof} \rangle$

lemma *mod_plus_minus_range_odd*:
 assumes "b>0" "odd b"
 shows " $y \text{ mod}+- b \in \{-\lfloor b/2 \rfloor .. \lfloor b/2 \rfloor\}$ "
 $\langle \text{proof} \rangle$

lemma *odd_smaller_b*:
 assumes "odd b"
 shows " $\lfloor \text{real_of_int } b / 2 \rfloor + \lfloor \text{real_of_int } b / 2 \rfloor < b$ "
 $\langle \text{proof} \rangle$

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⟩

```

```

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⟩

```

```

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"
⟨proof⟩

```

Image of 0.

```

lemma mod_plus_minus_zero:
assumes "x mod+- b = 0"
shows "x mod b = 0"
⟨proof⟩

```

```

lemma mod_plus_minus_zero':
assumes "b>0" "odd b"
shows "0 mod+- b = (0::int)"
⟨proof⟩

```

mod+- with negative values.

```

lemma neg_mod_plus_minus:
assumes "odd b"
          "b>0"
shows "(- x) mod+- b = - (x mod+- b)"
⟨proof⟩

```

Representative with mod+-

```

lemma mod_plus_minus_rep_ex:
"∃k. x = k*b + x mod+- b"
⟨proof⟩

```

```

lemma mod_plus_minus_rep:
obtains k where "x = k*b + x mod+- b"
⟨proof⟩

```

Multiplication in mod+-

```

lemma mod_plus_minus_mult:

```

```

    "s*x mod+- q = (s mod+- q) * (x mod+- q) mod+- q"
  <proof>
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 ( $\lambda x. f\ x + g\ x$ ))"
  <proof>

lemma all_impl_Max:
  assumes " $\forall x. f\ x \geq (a::int)$ "
          "finite (range f)"
  shows " $(MAX\ x. f\ x) \geq a$ "
  <proof>

lemma Max_mono':
  assumes " $\forall x. f\ x \leq g\ x$ "
          "finite (range f)"
          "finite (range g)"
  shows " $(MAX\ x. f\ x) \leq (MAX\ x. g\ x)$ "
  <proof>

lemma Max_mono_plus:
  assumes "finite (range (f:: $\_ \Rightarrow \_::ordered\_ab\_semigroup\_add$ ))"
          "finite (range g)"
  shows " $(MAX\ x. f\ x + g\ x) \leq (MAX\ x. f\ x) + (MAX\ x. g\ x)$ "
  <proof>

Lemmas for porting to qr.

lemma of_qr_mult:
  "of_qr (a * b) = of_qr a * of_qr b mod qr_poly"
  <proof>

lemma of_qr_scale:
  "of_qr (to_module s * b) =
  Polynomial.smult (of_int_mod_ring s) (of_qr b)"
  <proof>

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

```

<proof>

Lemmas on *round* and *floor*.

```
lemma odd_round_up:
  assumes "odd x"
  shows "round (real_of_int x / 2) = (x + 1) div 2"
<proof>
```

```
lemma floor_unique:
  assumes "real_of_int a ≤ x" "x < a+1"
  shows "floor x = a"
<proof>
```

```
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"
<proof>
```

```
lemma one_mod_four_round:
  assumes "x mod 4 = 1"
  shows "round (real_of_int x / 4) = (x-1) div 4"
<proof>
```

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}"
<proof>
```

```
lemma finite_Max:
  "finite (range ( $\lambda$ xa. abs_infty_q (poly.coeff (of_qr x) xa)))"
  <proof>
```

```
lemma finite_Max_scale:
  "finite (range ( $\lambda$ xa. abs_infty_q (of_int_mod_ring s *
    poly.coeff (of_qr x) xa)))"
  <proof>
```

```
lemma finite_Max_sum:
  "finite (range ( $\lambda$ xa. abs_infty_q
    (poly.coeff (of_qr x) xa + poly.coeff (of_qr y) xa)))"
  <proof>
```

```
lemma finite_Max_sum':
  "finite (range
    ( $\lambda$ xa. abs_infty_q (poly.coeff (of_qr x) xa) +
    abs_infty_q (poly.coeff (of_qr y) xa)))"
  <proof>
```

```
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>
```

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

```
lemma abs_infty_q_definite:
  "abs_infty_q x = 0  $\longleftrightarrow$  x = 0"
  <proof>
```

```
lemma abs_infty_q_pos:
  "abs_infty_q x  $\geq$  0"
  <proof>
```

```
lemma abs_infty_q_minus:
  "abs_infty_q (- x) = abs_infty_q x"
  <proof>
```

```
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"
  <proof>
```

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

```
lemma mod_plus_minus_leq_mod:
  "|x mod+- q| ≤ |x|"
  ⟨proof⟩
```

```
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⟩
```

```
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)"
  ⟨proof⟩
```

```
lemma abs_infty_q_scale:
  "abs_infty_q ((of_int_mod_ring s :: 'a mod_ring) * x) ≤
    |s| * (abs_infty_q x)"
  ⟨proof⟩
```

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⟩
```

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⟩
```

```
lemma abs_infty_poly_pos:
  "abs_infty_poly x ≥ 0"
  ⟨proof⟩
```

Again, homogeneity is only true for inequality not necessarily equality! Need to add $q \equiv 1 \pmod{4}$ 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⟩
```

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>

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>

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>
```

```
lemma ceiling_int:
  "[of_int a + b] = a + [b]"
<proof>
```

```
lemma deg_Poly':
  assumes "Poly xs ≠ 0"
  shows "degree (Poly xs) ≤ length xs - 1"
<proof>
```

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"
<proof>
```

```
lemma mod_plus_minus_range_q:
  assumes "y ∈ {-[q/2]..[q/2]}"
```

shows "y mod+- q = y"
 ⟨proof⟩

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

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

definition *decompress* :: "nat \Rightarrow int \Rightarrow 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"
 ⟨proof⟩

lemma *compress_less*:
 <compress d x < 2^d>
 ⟨proof⟩

lemma *decompress_zero*: "decompress d 0 = 0"
 ⟨proof⟩

Properties of the exponent d .

lemma *d_lt_logq*:
 assumes "of_nat d < [(log 2 q)::real]"
 shows "d < log 2 q"
 ⟨proof⟩

lemma *twod_lt_q*:
 assumes "of_nat d < [(log 2 q)::real]"
 shows "2 powr (real d) < of_int q"
 ⟨proof⟩

lemma *break_point_gt_q_div_two*:
 assumes "of_nat d < [(log 2 q)::real]"
 shows "[q - (q / (2 * 2^d))] > [q / 2]"
 ⟨proof⟩

lemma *decompress_zero_unique*:
 assumes "decompress d s = 0"
 "s \in {0..2^d - 1}"
 "of_nat d < [(log 2 q)::real]"

shows "s = 0"
 ⟨proof⟩

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}"
 ⟨proof⟩

lemma range_decompress:
assumes "x ∈ {0..2^d - 1}" "of_nat d < ⌈(log 2 q) :: real⌉"
shows "decompress d x ∈ {0..q-1}"
 ⟨proof⟩

Compression is a function from $\mathbb{Z}/q\mathbb{Z}$ 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⟩

When does the modulo operation in the compression function change the output? Only when $x \geq \lceil q - (q / (2 * 2^d)) \rceil$. 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)"
 ⟨proof⟩

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 "
 ⟨proof⟩

lemma compress_mod:
assumes "x ∈ {⌈q-(q/(2*2^d))⌉..q-1}"
 "of_nat d < ⌈(log 2 q) :: real⌉"
shows "compress d x = 0"
 ⟨proof⟩

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⟩

```

```

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⟩

```

```

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⟩

```

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⟩

```

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⟩

```

We have now defined compression only on integers (ie $\{0..<q\}$, corresponding to \mathbb{Z}_q). We need to extend this notion to the ring $\mathbb{Z}_q[X]/(X^{n+1})$. 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 from 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)"
<proof>

```

```

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>

```

```

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)"
<proof>

```

```

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⟩

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⟩

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"
⟨proof⟩

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⟩

lemma length_coeffs_of_qr:
  "length (coeffs (of_qr (x :: 'a qr))) < Suc (nat n)"
⟨proof⟩
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⟩

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"
⟨proof⟩

```

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⟩

```

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

```

lemma compress_1:
  shows "compress 1 x ∈ {0,1}"
⟨proof⟩

```

```

lemma compress_poly_1:
  shows "∀ i. poly.coeff (of_qr (compress_poly 1 x)) i ∈ {0,1}"
⟨proof⟩
end

```

```

lemma of_int_mod_ring_mult:
  "of_int_mod_ring (a*b) = of_int_mod_ring a * of_int_mod_ring b"
⟨proof⟩

```

```

context kyber_spec
begin
lemma decompress_1:
  assumes "a ∈ {0,1}"
  shows "decompress 1 a = round(real_of_int q/2) * a"
⟨proof⟩

```

```

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⟩
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 $\mathbb{Z}_q[X]/(X^{n+1})$.

```

definition compress_vec ::
  "nat  $\Rightarrow$  ('a qr, 'k) vec  $\Rightarrow$  ('a qr, 'k) vec" where
  "compress_vec d = map_vector (compress_poly d)"

definition decompress_vec ::
  "nat  $\Rightarrow$  ('a qr, 'k) vec  $\Rightarrow$  ('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 from 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 $0 < d < \lceil \log_2 q \rceil$. 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 from uniformly random input. Notice that e and $e1$ are vectors while $e2$ is a mere element in $\mathbb{Z}_q[X]/(X^{n+1})$.

```

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

```

```

definition encrypt ::
  "('a qr, 'k) vec  $\Rightarrow$  (('a qr, 'k) vec, 'k) vec  $\Rightarrow$ 
   ('a qr, 'k) vec  $\Rightarrow$  ('a qr, 'k) vec  $\Rightarrow$  ('a qr)  $\Rightarrow$ 
   nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a qr  $\Rightarrow$ 
   (('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  $\Rightarrow$  ('a qr)  $\Rightarrow$  ('a qr, 'k) vec  $\Rightarrow$ 
   nat  $\Rightarrow$  nat  $\Rightarrow$  '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  $\Rightarrow$  int)  $\Rightarrow$  ('a qr)  $\Rightarrow$  ('a qr)" where
  "f_int_to_poly f =
   to_qr  $\circ$ 
   Poly  $\circ$ 
   (map of_int_mod_ring)  $\circ$ 
   (map f)  $\circ$ 
   (map to_int_mod_ring)  $\circ$ 
   coeffs  $\circ$ 
   of_qr"

```

Error of compression and decompression.

```

definition compress_error_poly ::
  "nat  $\Rightarrow$  'a qr  $\Rightarrow$  'a qr" where
  "compress_error_poly d y =
   decompress_poly d (compress_poly d y) - y"

```

```

definition compress_error_vec ::
  "nat  $\Rightarrow$  ('a qr, 'k) vec  $\Rightarrow$  ('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)"
  <proof>

```

```

lemma scalar_product_linear_right:
  "scalar_product a (b+c) =
   scalar_product a b + scalar_product a (c :: ('a qr, 'k) vec)"
⟨proof⟩

```

```

lemma scalar_product_assoc:
  "scalar_product (A *v s) (r :: ('a qr, 'k) vec) =
   scalar_product s (r v* A)"
⟨proof⟩

```

Lemma about coeff Poly

```

lemma coeffs_in_coeff:
  assumes "∀i. poly.coeff x i ∈ A"
  shows "set (coeffs x) ⊆ A"
⟨proof⟩

```

```

lemma set_coeff_Poly: "set ((coeffs o Poly) xs) ⊆ set xs"
⟨proof⟩

```

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 o of_qr) m) ⊆ {0,1}"
  shows "decrypt u v s du dv = m"
⟨proof⟩

```

end

end

```

theory Kyber_Values
  imports
    Crypto_Scheme

```

begin

8 Specification for Kyber

```
typedef fin7681 = "{0..<7681::int}"  
  morphisms fin7681_rep fin7681_abs  
  ⟨proof⟩
```

```
setup_lifting type_definition_fin7681
```

```
lemma CARD_fin7681 [simp]: "CARD (fin7681) = 7681"  
  ⟨proof⟩
```

```
lemma fin7681_nontriv [simp]: "1 < CARD(fin7681)"  
  ⟨proof⟩
```

```
lemma prime_7681: "prime (7681::nat)" ⟨proof⟩
```

```
instantiation fin7681 :: comm_ring_1  
begin
```

```
lift_definition zero_fin7681 :: "fin7681" is "0" ⟨proof⟩
```

```
lift_definition one_fin7681 :: "fin7681" is "1" ⟨proof⟩
```

```
lift_definition plus_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"  
  is "(λx y. (x+y) mod 7681)"  
  ⟨proof⟩
```

```
lift_definition uminus_fin7681 :: "fin7681 ⇒ fin7681"  
  is "(λx. (uminus x) mod 7681)"  
  ⟨proof⟩
```

```
lift_definition minus_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"  
  is "(λx y. (x-y) mod 7681)"  
  ⟨proof⟩
```

```
lift_definition times_fin7681 :: "fin7681 ⇒ fin7681 ⇒ fin7681"  
  is "(λx y. (x*y) mod 7681)"  
  ⟨proof⟩
```

```
instance  
  ⟨proof⟩
```

```
end
```

```

instantiation fin7681 :: finite
begin
instance
  ⟨proof⟩
end

instantiation fin7681 :: equal
begin
lift_definition equal_fin7681 :: "fin7681 ⇒ fin7681 ⇒ bool" is "(=)" ⟨proof⟩
instance ⟨proof⟩
end

instantiation fin7681 :: nontriv
begin
instance
  ⟨proof⟩
end

instantiation fin7681 :: prime_card
begin
instance
  ⟨proof⟩
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⟩
end

lift_definition to_int_fin7681 :: "fin7681 ⇒ int" is "λx. x" ⟨proof⟩

lift_definition of_int_fin7681 :: "int ⇒ fin7681" is "λx. (x mod 7681)"
  ⟨proof⟩

interpretation to_int_fin7681_hom: inj_zero_hom to_int_fin7681
  ⟨proof⟩

interpretation of_int_fin7681_hom: zero_hom of_int_fin7681
  ⟨proof⟩

lemma to_int_fin7681_of_int_fin7681 [simp]:
  "to_int_fin7681 (of_int_fin7681 x) = x mod 7681"
  ⟨proof⟩

```

```

lemma of_int_fin7681_to_int_fin7681 [simp]:
  "of_int_fin7681 (to_int_fin7681 x) = x"
  <proof>

lemma of_int_mod_ring_eq_iff [simp]:
  "(of_int_fin7681 a = of_int_fin7681 b)  $\longleftrightarrow$ 
  ((a mod 7681) = (b mod 7681))"
  <proof>

interpretation kyber7681: kyber_spec 256 7681 3 8 "TYPE(fin7681)" "TYPE(3)"
  <proof>

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)"
  <proof>

lemma to_int_mod_ring_of_nat [simp]:
  "to_int_mod_ring (of_nat n :: 'a :: nontriv mod_ring) = n mod CARD('a)"
  <proof>

lemma to_int_mod_ring_numeral [simp]:
  "to_int_mod_ring (numeral n :: 'a :: nontriv mod_ring) = numeral n mod
  CARD('a)"
  <proof>

lemma of_int_mod_ring_eq_iff [simp]:
  "((of_int a :: 'a :: nontriv mod_ring) = of_int b)  $\longleftrightarrow$ 
  ((a mod CARD('a)) = (b mod CARD('a)))"
  <proof>

lemma of_nat_mod_ring_eq_iff [simp]:
  "((of_nat a :: 'a :: nontriv mod_ring) = of_nat b)  $\longleftrightarrow$ 
  ((a mod CARD('a)) = (b mod CARD('a)))"
  <proof>

```

```

lemma one_eq_numeral_mod_ring_iff [simp]:
  "(1 :: 'a :: nontriv mod_ring) = numeral a  $\longleftrightarrow$  (1 mod CARD('a)) = (numeral
a mod CARD('a))"
  <proof>

lemma numeral_eq_one_mod_ring_iff [simp]:
  "numeral a = (1 :: 'a :: nontriv mod_ring)  $\longleftrightarrow$  (numeral a mod CARD('a))
= (1 mod CARD('a))"
  <proof>

lemma zero_eq_numeral_mod_ring_iff [simp]:
  "(0 :: 'a :: nontriv mod_ring) = numeral a  $\longleftrightarrow$  0 = (numeral a mod CARD('a))"
  <proof>

lemma numeral_eq_zero_mod_ring_iff [simp]:
  "numeral a = (0 :: 'a :: nontriv mod_ring)  $\longleftrightarrow$  (numeral a mod CARD('a))
= 0"
  <proof>

lemma numeral_mod_ring_eq_iff [simp]:
  "((numeral a :: 'a :: nontriv mod_ring) = numeral b)  $\longleftrightarrow$ 
((numeral a mod CARD('a)) = (numeral b mod CARD('a)))"
  <proof>

instantiation bit1 :: (finite) nontriv
begin
instance <proof>
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"
  <proof>

```

```

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  $\omega$  :: "('a::qr_spec) mod_ring"
  and  $\mu$  :: "'a mod_ring"
  and  $\psi$  :: "'a mod_ring"
  and  $\psi_{\text{inv}}$  :: "'a mod_ring"
  and  $n_{\text{inv}}$  :: "'a mod_ring"
  and mult_factor :: int
assumes
  omega_properties: " $\omega^n = 1$ " " $\omega \neq 1$ " " $(\forall m. \omega^m = 1 \wedge m \neq 0 \longrightarrow$ 
 $m \geq n)$ "
  and mu_properties: " $\mu * \omega = 1$ " " $\mu \neq 1$ "
  and psi_properties: " $\psi^2 = \omega$ " " $\psi^n = -1$ "
  and psi_psiinv: " $\psi * \psi_{\text{inv}} = 1$ "
  and n_ninv: "(of_int_mod_ring n) *  $n_{\text{inv}} = 1$ "
  and q_split: " $q = \text{mult\_factor} * n + 1$ "
begin

```

Some properties of the roots ω and ψ and their inverses μ and ψ_{inv} .

```

lemma mu_prop:
  " $(\forall m. \mu^m = 1 \wedge m \neq 0 \longrightarrow m \geq n)$ "
  <proof>

```

```

lemma mu_prop':
assumes " $\mu^{m'} = 1$ " " $m' \neq 0$ " shows " $m' \geq n$ "
  <proof>

```

```

lemma omega_prop':
assumes " $\omega^{m'} = 1$ " " $m' \neq 0$ " shows " $m' \geq n$ "
  <proof>

```

```

lemma psi_props:
shows " $\psi^{(2*n)} = 1$ "
  " $\psi^{(n*(2*a+1))} = -1$ "
  " $\psi \neq 1$ "
  <proof>

```

```

lemma psi_inv_exp:
  " $\psi^i * \psi_{\text{inv}}^i = 1$ "
  <proof>

```

```

lemma inv_psi_exp:
  " $\psi_{\text{inv}}^i * \psi^i = 1$ "
  <proof>

```

```

lemma negative_psi:

```

```

assumes "i<j"
shows " $\psi^j * \psi^{inv\ i} = \psi^{(j-i)}$ "
<proof>

```

```

lemma negative_psi':
assumes "i≤j"
shows " $\psi^{inv\ i} * \psi^j = \psi^{(j-i)}$ "
<proof>

```

```

lemma psiinv_prop:
shows " $\psi^{inv\ 2} = \mu$ "
<proof>

```

```

lemma n_ninv':
"ninv * (of_int_mod_ring n) = 1"
<proof>

```

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>

```

```

lemma map_upto_n_mod:
"(Poly (map f [0..<n]) mod qr_poly) = (Poly (map f [0..<n]) :: 'a mod_ring
poly)"
<proof>

```

```

lemma coeff_of_qr_zero:
assumes "i≥n"
shows "poly.coeff (of_qr (f :: 'a qr)) i = 0"
<proof>

```

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 = ( $\sum_{j \in \{0..<n\}} (\text{poly.coeff (of_qr g) j}) * \psi^{(j * (2*i+1))}$ )"

```

```

definition ntt_coeffs :: "'a qr ⇒ 'a mod_ring list" where

```

```
"ntt_coefs g = map (ntt_coeff_poly g) [0..<n]"
```

```
definition ntt_poly :: "'a qr ⇒ 'a qr" where
"ntt_poly g = to_qr (Poly (ntt_coefs 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_coefs :: "'a qr ⇒ 'a mod_ring list" where
"inv_ntt_coefs 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_coefs 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⟩
```

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

```
definition qr_mult_coefs :: "'a qr ⇒ 'a qr ⇒ 'a qr" (infixl <*> 70) where
"qr_mult_coefs f g = to_qr (map2_poly (*) (of_qr f) (of_qr g))"
```

The definition of the exponentiation $\hat{\cdot}$ 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)"
```

<proof>

Representation of f modulo qr_poly .

lemma *mod_div_qr_poly*:

"($f :: 'a \text{ mod_ring poly}$) = ($f \text{ mod } qr_poly$) + $qr_poly * (f \text{ div } qr_poly)$ "

<proof>

take_deg returns the first n coefficients of a polynomial.

definition *take_deg* :: " $\text{nat} \Rightarrow ('b::\text{zero}) \text{ poly} \Rightarrow 'b \text{ poly}$ " where

" $take_deg = (\lambda n. \lambda f. \text{Poly } (take \ n \ (coeffs \ f)))$ "

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

definition *drop_deg* :: " $\text{nat} \Rightarrow ('b::\text{zero}) \text{ poly} \Rightarrow 'b \text{ poly}$ " where

" $drop_deg = (\lambda n. \lambda f. \text{Poly } (drop \ n \ (coeffs \ f)))$ "

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

lemma *take_deg_monom_drop_deg*:

assumes " $degree \ f \geq n$ "

shows " $(f :: 'a \text{ mod_ring poly}) = take_deg \ n \ f + (\text{Polynomial.monom } 1 \ n) * drop_deg \ n \ f$ "

<proof>

lemma *split_mod_qr_poly*:

assumes " $degree \ f \geq n$ "

shows " $(f :: 'a \text{ mod_ring poly}) = take_deg \ n \ f - drop_deg \ n \ f + qr_poly * drop_deg \ n \ f$ "

<proof>

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

lemma *degree_drop_n*:

" $degree \ (drop_deg \ n \ f) = degree \ f - n$ "

<proof>

lemma *degree_drop_2n*:

assumes " $degree \ f < 2*n$ "

shows " $degree \ (drop_deg \ n \ f) < n$ "

<proof>

lemma *degree_take_n*:

" $degree \ (take_deg \ n \ f) < n$ "

<proof>

lemma *deg_mult_of_qr*:

" $degree \ (of_qr \ (f :: 'a \ qr) * of_qr \ g) < 2 * n$ "

<proof>

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⟩

```

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"
⟨proof⟩

```

```

lemma coeff_drop_deg:
assumes "i < n"
shows "poly.coeff (drop_deg n f) i = poly.coeff (f :: 'a mod_ring poly) (i+n)"
⟨proof⟩

```

```

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)"
⟨proof⟩

```

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⟩

```

```

lemma less_diff:
assumes "l1 < l2"
shows "{..<l2} - {...l1} = {l1<..

```

```

lemma sum_less_split:
assumes "l1 < (l2::nat)"
shows "sum f {...<l2} = sum f {...l1} + sum f {l1<..

```

```

lemma div_minus_1:
assumes "(x::int) ∈ {-b..shows "x div b = -1"
⟨proof⟩

```

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 =
 $(\sum_{j < n} \text{conv_sign} ((\text{int } i - \text{int } j) \text{ div } n) * \text{poly.coeff (of_qr f) } j * \text{poly.coeff (of_qr g) (nat ((\text{int } i - \text{int } j) \text{ mod } n))))$ "
 ⟨proof⟩

Polynomial multiplication in R_q is the negacyclic convolution.

lemma *mult_negacycl*:
 "f * g = negacycl_conv f g"
 ⟨proof⟩

Additional lemmas on *ntt_coeffs*.

lemma *length_ntt_coeffs*:
 "length (ntt_coeffs f) ≤ n"
 ⟨proof⟩

lemma *degree_Poly_ntt_coeffs*:
 "degree (Poly (ntt_coeffs f)) < n"
 ⟨proof⟩

lemma *Poly_ntt_coeffs_mod_qr_poly*:
 "Poly (ntt_coeffs f) mod qr_poly = Poly (ntt_coeffs f)"
 ⟨proof⟩

lemma *nth_default_map*:
 assumes "i < na"
 shows "nth_default x (map f [0..<na]) i = f i"
 ⟨proof⟩

lemma *nth_coeffs_negacycl*:
 assumes "j < n"
 shows "poly.coeff (of_qr (negacycl_conv f g)) j =
 $(\sum_{i < n} \text{conv_sign} ((\text{int } j - \text{int } i) \text{ div } \text{int } n) * \text{poly.coeff (of_qr f) } i * \text{poly.coeff (of_qr g) (nat ((\text{int } j - \text{int } i) \text{ mod } \text{int } n))))$ "
 ⟨proof⟩

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)"
 ⟨proof⟩

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>

```

```

lemma ntt_coeffs_mult:
assumes "i<n"
shows "ntt_coeffs (f*g) !i = ntt_coeffs f ! i * ntt_coeffs g ! i"
<proof>

```

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>

```

```

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)]))"
<proof>

```

Convolution theorem for NTT

```

lemma ntt_mult:
"ntt_poly (f * g) = qr_mult_coeffs (ntt_poly f) (ntt_poly g)"
<proof>

```

Correctness of NTT on polynomials.

```

lemma inv_ntt_poly_correct:
"inv_ntt_poly (ntt_poly f) = f"
<proof>

```

```

lemma ntt_inv_poly_correct:
"ntt_poly (inv_ntt_poly f) = f"
<proof>

```

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))"
<proof>

```

```

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"
  <proof>

```

```

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"
  <proof>

```

```

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"
  <proof>

```

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 \Rightarrow ('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"
⟨proof⟩

definition encrypt_ntt ::
"('a qr, 'k) vec \Rightarrow (('a qr, 'k) vec, 'k) vec \Rightarrow
('a qr, 'k) vec \Rightarrow ('a qr, 'k) vec \Rightarrow ('a qr) \Rightarrow
nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a qr \Rightarrow
(('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"
⟨proof⟩

definition decrypt_ntt ::
"('a qr, 'k) vec \Rightarrow ('a qr) \Rightarrow ('a qr, 'k) vec \Rightarrow
nat \Rightarrow nat \Rightarrow '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"
⟨proof⟩

(1 - δ)-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 \circ of_qr) m) \subseteq {0,1}"
shows "decrypt_ntt u v s du dv = m"

<proof>

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  $\Rightarrow$  nat  $\Rightarrow$  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 ( $\lambda$ l. fast_exp_7681 (3844::int) l  $\neq$  1) [1..<256]"  
  <proof>
```

```
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>
```

```
lemma fast_exp_7681_less256:  
assumes "0<1" "1<256"  
shows "fast_exp_7681 3844 l  $\neq$  1"  
<proof>
```

```
lemma powr_less256:  
assumes "0<1" "1<256"  
shows "(3844::fin7681 mod_ring)1  $\neq$  1"  
<proof>
```

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)"  
  <proof>
```

```
lemma "62 * 1115 = (1 :: fin7681 mod_ring)"  
  <proof>
```

```
lemma "256 * 7651 = (1 :: fin7681 mod_ring)"  
  <proof>
```

```
lemma "7681 = 30 * 256 + (1 :: int)" <proof>
```

```
lemma powr256: "3844 ^ 256 = (1 :: fin7681 mod_ring)"  
  <proof>
```

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

```
interpretation kyber7681_ntt: kyber_ntt 256 7681 3 8  
  "TYPE(fin7681)" "TYPE(3)" 3844 6584 62 1115 7651 30  
  <proof>
```

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.