Formalization and Verification of Number Theoretic Algorithms Using the Mizar Proof Checker

Hiroyuki Okazaki†, Yoshiki Aoki† and Yasunari Shidama†
†Shinshu University
4-17-1 Wakasato Nagano-city, Nagano 380-8553 Japan
okazaki@cs.shinshu-u.ac.jp, 11ta501a@shinshu-u.ac.jp, shidama@shinshu-u.ac.jp
Tel: +81(26)269-5503 Fax: +81(26)269-5503

Abstract: In this paper, we introduce formalization of well-known number theoretic algorithms on the Mizar proof checking system. We formalized the Euclidean algorithm, the extended Euclidean algorithm and the algorithm computing the solution of the Chinese reminder theorem based on the source code of NZMATH which is a Python based number theory oriented calculation system. We prove the accuracy of our formalization using the Mizar proof checking system as a formal verification tool.

Keywords: Formal Verification, Mizar, Number theoretic algorithm

1 Introduction

Mizar[1, 2] is a project that formalizes mathematics with a computer-aided proving technique. Number theoretic algorithms play an important role in information society. For example, number theoretic algorithms are essential to cryptology and code theory because they provide secure and high-speed communications. However, there is no evidence that the calculated value produced by an algorithm is accurate, although the algorithm has a processing nature. Therefore, when we propose an algorithm, we have to prove the accuracy of the algorithm.

On the other hand, a developed program for an algorithm is not necessary to calculate a precise value although the accuracy of the algorithm was proven. This is because it is difficult to develop a program which functions exactly like the algorithm. We have to verify that the algorithm is accurately encoded into a programming language.

The objective of this study is to prove the accuracy of the algorithms, encoded in a programming language, using the Mizar proof checker. To achieve this, first, we formalized algorithms in the Mizar language to confirm that the formalization agrees with our aim. This is because there are several methods how to formalize algorithms.

In this paper, we introduce formalization of well-known number theoretic algorithms on the Mizar proof checking system. We formalized the Euclidean algorithm, the extended Euclidean algorithm and the algorithm computing the solution of the Chinese reminder theorem based on the source code of NZMATH[3], which is a Python based number theory oriented calculation system. Then we verified the accuracy of the formalized algorithms using the Mizar proof checker.

The remainder of the study is organized as follows. We briefly introduce the Mizar project in Section 2 and NZMATH in Section 3. In Section 4, we discuss our strategy for formalizing algorithms in Mizar, followed by Section 5 where we propose a formalization of the Euclidean algorithm. In Section 6, we propose the formalization of the extended Euclidean algorithm, and in Section 7, we propose a formalization of the algorithm, computing the solution of the Chinese reminder theorem. We conclude our discussions in Section 8. The definitions and theorems in this study have been verified for accuracy using the Mizar proof checker.

2 Mizar

Mizar[1, 2] is an advanced project of the Mizar Society led by Andrzej Trybulec. It formalizes mathematics with a computer-aided proving technique. The Mizar project describes mathematical proofs in the Mizar-language, which is created to describe mathematics formally. The Mizar proof checker operates in both Windows and UNIX environments and registers proven definitions and theorems in the Mizar Mathematical Library (MML).

Furthermore, the objective of the Mizar project is to create a checking system for mathematical theses. An “article” formalizes and describes mathematical proofs by Mizar. When an article is newly described, it is possible to advance it by referring to articles registered in MML, which have already been inspected as proofs. Similarly, other articles can refer to an article after it is registered in MML. Although the Mizar language is based on the description method for general mathematical proofs, the reader should consult the references for grammatical details, because Mizar uses a specific, unique notation[1, 2].
3 NZMATH

NZMATH[3] is a number theory oriented calculation system mainly developed by the Nakamula laboratory at Tokyo Metropolitan University. Number theoretic algorithms are implemented as Python functions on NZMATH, which is freely available and distributed under the BSD license. NZMATH is at an early stage of development and is currently being developed.

4 Strategy of Formalizing Algorithms in Mizar

In Mizar, there are several methods to define computational routines, representing algorithms. One method is to define a routine as a program for SCM. SCM is a general model of a stack machine defined in the Mizar system. In principle, we may formalize arbitrary programs in SCM. However, this approach may not be suitable to prove the accuracy of algorithms encoded in a high level programming language, because we have to define an algorithm as the machine code of SCM. For example, the Euclidean Algorithm has been formalized in SCM[4] (Definition A.1).

Another method is to define a routine as a functor or a Function. A functor is a relation between input and output of the routine in Mizar. It is easy to write and understand the formalization of a routine as a functor because the format of a functor in Mizar is similar to that of a function in programming languages. Thus, in this paper, we formalize an algorithm as a functor.

A Function is a map from the input space onto the output space. We can handle a Function as an element of the set of Functions. Note that both functor and Function can use a Function as their substitutable subroutine. For example, we formalized the algorithm of DES cipher as a functor, which uses substitution subroutines defined as Functions[5, 6].

4.1 Formalizing Loop Structure in Mizar

In this section, we propose an approach to describe programs with a loop structure. Particularly, we elucidate the method to formalize programs with a non-nested loop structure. This suggests that it is possible to formalize a program with looped structures by nesting single loop structures recursively.

A loop is a sequence of statements. In a computer implementation, variables that are allocated memory segments, are assigned destructively according to the given recursion formula in each iteration.

To describe the loop structure in the Mizar language, we consider that there are sequences of variables to capture the repetition of operations. For example, let a be the sequence of variables such that a_i, the ith member of a, represents the temporary value assigned as a in the ith iteration of the loop structure. Note that we can describe the control condition of the iteration using the index number of the sequences of variables.

We can employ the inductive method to prove the property of the variables using such sequences. The Mizar system has a mechanism, called “scheme”, which enables us to prove propositions using the inductive method. We will show an example of a proof using “scheme” in Section 5.

5 Formalization of the Euclidean Algorithm

In this section we introduce our formalization of the Euclidean algorithm.

The Euclidean algorithm is a method that computes the greatest common divisor of two given integers. This algorithm is implemented in NZMATH as follows:

Code 5.1 (Euclidean algorithm in NZMATH)

```python
def gcd(a, b):
    a, b = abs(a), abs(b)
    while b:
        a, b = b, a % b
    return a
```

We formalize this algorithm as the following functor in the Mizar language:

Definition 5.1 (Euclidean Algorithm in Mizar)

```mizar
let a,b be Element of INT;
func ALGO_GCD(a,b) -> Element of NAT
means
ex A,B be sequence of NAT
st
A.0 = abs(a) & B.0 = abs(b) &
(for i be Element of NAT holds
A.(i+1) = B.i &
B.(i+1) = A.i mod B.i) &
it = A.(min*(i where i is Nat: B.i = 0));
```

Note that all algorithms, which were formalized in this paper, do not have nested loops.

Note that a_i, b_i are the values of a, b in the ith iteration, respectively.
5.1 Accuracy of ALGO_GCD

In this section we prove the accuracy of our formalization that the functor ALGO_GCD returns the greatest common divisor of a given two integers.

We will prove the following theorem:

**Theorem 5.1 (Accuracy of ALGO_GCD)**

for a, b be Element of INT
holds
\[ \text{ALGO}_{\text{GCD}}(a, b) = a \gcd b \]

Here \( \gcd \) is the conceptual definition of the greatest common divisor in Mizar as follows:

**Definition 5.2 (gcd of Mizar)**

let a, b be Integer;
func a gcd b \to Nat means
it divides a
& it divides b
& for m being Integer st m divides a
& m divides b holds m divides it;

We proved Theorem 5.1 using the following lemma:

**Lemma 5.1** for i be Element of NAT
st B.i <> 0
holds
A.0 gcd B.0 = A.i gcd B.i

In the rest of this section, we show a schematic proof for Lemma 5.1. First, we defined the following predicate

\[ \text{defpred P[Nat] means } B.1 <> 0 \implies \text{A.0 gcd B.0 = A.1 gcd B.1} ; \]

Here, the symbol ‘$1’ denotes the argument of \( P^2 \).

Therefore, \( P[0] \) is evidently true. Next, we prove that \( P[i+1] \) is true if \( P[i] \) is true as follows:

for i being Element of NAT
st P[i] holds P[i+1];

Finally, we can prove Lemma 5.1 with the following mathematical induction scheme that had been defined in the Mizar system:

**Scheme 5.1 (Mathematical Induction scheme)**

\[ \text{Ind} \{ P [\text{Nat}] \} : \]
for k being Nat holds P1[k]
provided
P1[0] and
for k being Nat st P1[k] holds
P1[k + 1]

It should be noted that we are allowed to create new schemes.

6 Formalization of the Extended Euclidean Algorithm

In this section we formalize the extended Euclidean algorithm. The extended Euclidean algorithm can compute \( a, b \) and \( g \) for given integers \( x, y \) such that \( ax + by = g \) (\( g \) is the greatest common divisor of \( x, y \)). This algorithm is implemented in NZMATH as follows:

**Code 6.1 (The extended Euclidean algorithm in NZMATH)**

\[
\begin{align*}
def \text{extgcd}(x,y): & \\
& a, b, g, u, v, w = 1, 0, x, 0, 1, y \\
& \text{while } w: \\
& \quad q, t = \text{divmod}(g, w) \\
& \quad a, b, g, u, v, w = u, v, w, a-q*u, b-q*v, t \\
& \quad \text{if } g \geq 0: \\
& \quad \quad \text{return } (a, b, g) \\
& \quad \text{else:} \\
& \quad \quad \text{return } (-a, -b, -g) \\
\end{align*}
\]

We formalize this algorithm as the following functor in the Mizar language:

**Definition 6.1 (The extended Euclidean algorithm in Mizar)**

let x, y be Element of INT;
func ALGO_EXGCD(x, y)
\to Element of [:INT,INT,INT:] means
ex g, w, q, t be sequence of INT,
a, b, v, u be sequence of INT,
istop be Element of NAT
st
a.0 = 1 & b.0 = 0 & g.0 = x & q.0 = 0 \\
& u.0 = 0 & v.0 = 1 & w.0 = y & t.0 = 0 \\
& (for i be Element of NAT holds \\
q.(i+1) = g.i \div \text{w.i} \\
& t.(i+1) = g.i \mod \text{w.i} \\
& a.(i+1) = u.i & b.(i+1) = v.i \\
& g.(i+1) = w.i \\
& u.(i+1) = a.i - q.(i+1)*u.i \\
& v.(i+1) = b.i - q.(i+1)*v.i \\
& w.(i+1) = t.(i+1)) \\
& istop = \text{min*}\{i where i is Nat: w.i = 0\} \\
& (0 \leq g.\text{istop implies} \\
\quad \text{it =}[a.\text{istop}, b.\text{istop}, g.\text{istop}] \\
& \quad (g.\text{istop} < 0 \implies \text{it =}[-(a.\text{istop}), -(b.\text{istop}), -(g.\text{istop})])
\end{align*}\]

Note that ALGO_EXGCD(x,y) returns the 3-tuple \((a, b, g)\) such that \(ax + by = g\) and \(g\) is the greatest common divisor of \(x\) and \(y\).
6.1 Accuracy of ALGO_EXGCD

In this section we prove the accuracy of our formalization that the functor ALGO_EXGCD returns \( a, b \) and \( g \) for given integers \( x \) and \( y \) such that \( ax + by = g \) (\( g \) is the greatest common divisor of \( x \) and \( y \)).

We can prove the following theorem in a similarly for proving Theorem 6.1:

**Theorem 6.1** (Accuracy of ALGO_EXGCD)

for \( x, y \) be Element of INT
holds
\[
\text{ALGO_EXGCD}(x, y)\'3 = x \gcd y
\]
& \[
\text{ALGO_EXGCD}(x, y)\'1 \cdot x + \text{ALGO_EXGCD}(x, y)\'2 \cdot y
= x \gcd y,
\]
where \( \text{ALGO_EXGCD}(x, y)\'n \) denotes the \( n \)th member of \( \text{ALGO_EXGCD}(x, y) \). Thus we proved the accuracy of our formalization of extended Euclidean algorithm.

6.2 Multiplicative Inverse

Then, we define the functor that computes the multiplicative inverse over a residue class ring using the ALGO_EXGCD as follows:

**Definition 6.2** (Inverse)

let \( x, p \) be Element of INT;
func ALGO_INVERSE(x, p) -> Element of INT
means
for \( y \) be Element of INT
st \( y = (x \mod p) \)
holds
\[
\text{ALGO_EXGCD}(p, y)\'3 = 1 \text{ implies } ((\text{ALGO_EXGCD}(p, y)\'2 < 0) \text{ implies (ex z be Element of INT st z = ALGO_EXGCD(p, y)\'2 & it = p + z ))}
\]
& ((0 <= ALGO_EXGCD(p, y)\'2) implies it = ALGO_EXGCD(p, y)\'2 ) )
& ( ALGO_EXGCD(p, y)\'3 <> 1 implies it = {} );

We will define another algorithm with this functor in Section 7.

7 Formalization of the Algorithm Computing the Solution of the Chinese Reminder Theorem

In this section we formalize the algorithm computing the solution of the Chinese reminder theorem.

7.1 The Chinese Reminder Theorem

First, we review the Chinese reminder theorem briefly.

**Theorem 7.1** (Chinese Reminder Theorem) Let \( m_1, m_2, \ldots, m_r \) be relatively prime. For any integer \( a_1, a_2, \ldots, a_r \), there exists the unique solution \( x \in \mathbb{Z} / (m_1 \cdot m_2 \cdots m_r) \mathbb{Z} \) such that

\[
\begin{align*}
    x &\equiv a_1 \mod m_1 \\
    x &\equiv a_2 \mod m_2 \\
    &\vdots \\
    x &\equiv a_r \mod m_r. \\
\end{align*}
\]

We can compute such a solution \( x \) by the following steps:
First, we solve

\[
\begin{align*}
    x &\equiv a_1 \mod m_1 \\
    x &\equiv a_2 \mod m_2
\end{align*}
\]
by

\[
    x_0 = a_1 + (a_2 - a_1)(m_1^{-1} \mod m_2)m_1.
\]

Then:

\[
\begin{align*}
    x_0 \mod m_1 &= a_1 \\
    x_0 \mod m_2 &= a_1 + (a_2 - a_1) \\
    &= a_2
\end{align*}
\]
Thus, \( x_0 \) is the solution of (1).
Next, we solve the congruencies:

\[
\begin{align*}
    x &\equiv x_0 \mod m_1m_2 \\
    x &\equiv a_3 \mod m_3
\end{align*}
\]
Then we solve the next congruent expression and the solution of (3) sequentially. Finally, we can solve (1).

7.2 Formalization of the Algorithm Computing the Solution of the Chinese Reminder Theorem

In this paper, let us term the algorithm mentioned in Sec. 7.1 as “CRT algorithm”. The CRT algorithm is implemented in NZMATH as follows:

**Code 7.1** (CRT in NZMATH)

```python
def CRT(nlist):
    r = len(nlist)
    if r == 1:
        return nlist[0][0]
    product = []
    prodiv = []
    m = 1
    for i in range(1, r):
        m = m*nlist[i-1][1]
        c = inverse(m, nlist[i][1])
        product.append(m)
        prodiv.append(c)
    M = product[r-2]*nlist[r-1][1]
    n = nlist[0][0]
```

In this paper, let us term the algorithm mentioned in Sec. 7.1 as “CRT algorithm”. The CRT algorithm is implemented in NZMATH as follows:
for i in range(1, r):
    u = ((nlist[i][0]-n)*prodinv[i-1])
    n += u*product[i-1]
return n % M

Here nlist denotes the given congruencies. For example, if the given congruencies are

\[
\begin{align*}
    x &\equiv 2 \pmod{3} \\
    x &\equiv 3 \pmod{5} \\
    x &\equiv 2 \pmod{7}
\end{align*}
\]

then nlist is as follows:

\[
\text{nlist} = \{(2,3), (3,5), (2,7)\}
\]

We then formalize the algorithm as the following functor in the Mizar language:

**Definition 7.1** (CRT in Mizar)

let nlist be non empty FinSequence of [:INT,INT:];
func ALGO_CRT(nlist)->Element of INT means
( len nlist=1 implies it=(nlist.1)`1 \\
  & len nlist <> 1 implies \\
  &   ex m,n,prodc,prodi
    be FinSequence of INT, M0,M be Element of INT \\
    st len m = len nlist \\
    & len n = len nlist \\
    & len prodc = len nlist - 1 \\
    & len prodi = len nlist - 1 \\
    & m.1 = 1 \\
    & ( for i be Nat \\
        st 1<=i & i<=(len m) - 1 \\
        holds \\
        ex d,x,y be Element of INT \\
        st x = (nlist.i)`2 \\
        & m.(i+1) = m.i * x \\
        & y = m.(i+1) \\
        & d = (nlist.(i+1))`2 \\
        & prodi.i = ALGO_INVERSE(y,d) \\
        & prodc.i = y ) \\
    & M0 = (nlist.(len m))`2 \\
    & M = (prodc.(len m)-1)*M0 \\
    & n.1 = (nlist.1)`1 \\
    & ( for i be Nat \\
        st 1<=i & i<=len m - 1 \\
        holds \\
        ex u,u0,u1 be Element of INT \\
        st u0 = (nlist.(i+1))`1 \\
        & u1 = (nlist.(i+1))`2 \\
        & u = ((u0-n.i) * (prodi.i)) mod u1 \\
        & n.(i+1) = n.i + u*(prodc.i) ) \\
    & it = n.(len m) mod M );

Here, m, prod, prodi, M and n are finite sequences of N such that

\[
\begin{align*}
    \text{prodc} &= \{\text{prodc}_1, \text{prodc}_2, \ldots, \text{prodc}_r\}, \\
    \text{prodi} &= \{\text{prodi}_1, \text{prodi}_2, \ldots, \text{prodi}_r\}, \\
    m &= \{m_1, m_2, \ldots, m_i, m_{i+1}, \ldots, m_r\}, \\
    n &= \{n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_r\}, \\
    m_1 &= 1, n_1 = nlist[1][1], \\
    m_{i+1} &= m_i * nlist[i][2], \\
    \text{prodc}_i &= n_{i+1}^{-1} \mod nlist[i+1][2] \\
    \text{prodi}_i &= m_{i+1} \\
    n_{i+1} &= n_i + u * \text{prodc}_i,
\end{align*}
\]

Note that prodc_i, prodi_i, m_i and n_i are the value of product, prodinv, m and n in the ith iteration respectively. Additionally, we do not use infinite sequences but finite sequences for this algorithm because the count of the iteration is predetermined.

We then prove the following theorem:

**Theorem 7.2** (Accuracy of ALGO_CRT)

for nlist be non empty FinSequence of [:INT,INT:], a,b be non empty FinSequence of INT, x,y be Element of INT
st len a = len b \\
& len n = len nlist \\
& ( for i be Nat \\
    st i in Seg (len nlist) \\
    holds \\
    b.i <> 0 ) \\
& ( for i be Nat \\
    st i in Seg (len nlist) \\
    holds \\
    (nlist.i)´1 = a.i \\
    & (nlist.i)´2 = b.i ) \\
& ( for i,j be Nat \\
    st i in Seg (len nlist) \\
    & j in Seg (len nlist) \\
    & i <> j \\
    holds \\
    b.i,b.j are_relative_prime ) \\
& ( for i be Nat \\
    st i in Seg (len nlist) \\
    holds \\
    x mod b.i = a.i mod b.i )
& y = \text{Product } b
\text{ holds}
\text{ALGO\_CRT(nlist) mod } y = x \mod y

Here \texttt{are\_relative\_prime} and \texttt{Product} denote the definition in Mizar (Definitions A.3 and A.4). Thus, we proved the accuracy of our formalization of the CRT algorithm.

8 Conclusions

In this study, we introduced our formalization of the Euclidean algorithm, the extended Euclidean algorithm, and the CRT algorithm based on the source code of NZMATH. Moreover, we proved the accuracy of our formalization using the Mizar proof checking system as a formal verification tool. Therefore, we can conclude that our approach can formalize algorithms with a single loop structure precisely. Currently, we are attempting to develop methods to convert the encoded algorithms from NZMATH into Mizar automatically.\footnote{We have already been able to convert Python programs with a single loop structure.}

Acknowledgments

This work was supported by JSPS KAKENHI 21240001 and 22300285.

References

[1] Mizar Proof Checker, Available at \url{http://mizar.org/}.

A Related Definitions of Functors in Mizar

\textbf{Definition A.1} (Euclidean Algorithm in SCM)
\begin{verbatim}
func GCD-Algorithm -> Program of SCMPDS
equals
(((GBP:=0 );' (SBP := 7 ) ';'
saveIC(SBP,RetIC) ';' goto 2 ';'
halt SCMPDS ) ';'
(SBP,3)<=0 goto 9 ';'
((SBP,6):=(SBP,3)) ';'
Divide(SBP,2,SBP,3) ';'
((SBP,7):=(SBP,3)) ';'
((SBP,4+RetSP):=(GBP,1))) ';'
AddTo(GBP,1,4) ';'
saveIC(SBP,RetIC) ';'
(goto -7) ';'
((SBP,2):=(SBP,6)) ';'
return SBP;
\end{verbatim}

\textbf{Definition A.2} (Minimum Member)
\begin{verbatim}
let A be set ;
func min* A -> Element of NAT means
( it in A & ( for k being Nat st k in A holds it <= k )
) if A is non empty Subset of NAT otherwise it = 0 ;
\end{verbatim}

\textbf{Definition A.3} (Relatively Prime)
\begin{verbatim}
let a, b be Ordinal;
pred a,b are_relative_prime means
for c, d1, d2 being Ordinal st a = c *^ d1 & b = c *^ d2
holds c = 1;
\end{verbatim}

\textbf{Definition A.4} (Product)
\begin{verbatim}
func product f -> set means
for x being set holds
( x in it iff ex g being Function st
( x = g & dom g = dom f
& ( for y being set st y in dom f
holds g . y in f . y ) ) )
\end{verbatim}