Formal proof for delayed finite field arithmetic using floating point operators

Marc Daumas, Pascal Giorgi

Formal proof for delayed finite field arithmetic using floating point operators*

Marc Daumas[1] and Pascal Giorgi[2]

[1] LIRMM (UMR 5506 CNRS–UM2) and ELIAUS (EA 3679 UPVD)
[2] ELIAUS (EA 3679 UPVD)

Abstract Formal proof checkers such as Coq are capable of validating proofs of correction of algorithms for finite field arithmetics but they require extensive training from potential users. The delayed solution of a triangular system over a finite field mixes operations on integers and operations on floating point numbers. We focus in this report on proof obligations that state that no round off error occurred on any of the floating point operations. We use a tool named Gappa that can be learned in a matter of minutes to generate proofs related to floating point arithmetic and hide technicalities of formal proof checkers.

1 Introduction

Introducing a new algorithm is a difficult task. Authors have to persuade readers that their algorithm is correct and efficient. Such goals are usually attained by providing pen-and-paper proofs of correction more or less interlaced with the description of the algorithm. Authors may also provide results of tests to guarantee correction and efficiency on random cases and on known or new hard cases. Alas, this process is known to fail on mundane as well as notorious occurrences [1,2].

Developing a proof of correction in a formal proof checker using higher order logic such as Coq [3,4] would be a nice alternative but such a task usually represents a large amount of work outside the fields of expertise of most authors.

The delayed solver studied here works on a $N \times N$ unitary triangular matrix on $\mathbb{Z}/p\mathbb{Z}$ finite field. The key improvement of this algorithm compared to state of the art lies in the fact that delayed algorithms use floating point units to perform operations with no rounding error and delay computations of remainders as much as possible. Operations on floating point numbers are limited to two functions. The other functions use integer arithmetic.

The first function (`DGEMM_NEG`) performs a naive matrix multiplication and Gappa handles easily the proof obligation generated by a tool such a Why [5]. The second function (`DTRSM`) is invoked only under the predicate $Pred(N,p) = N \leq n_{\max}(p)$. This is enforced by the condition on the induction of the invoking function (`LZ_TRSM`). The predicate may be rewritten to $Pred(N,p) = p \leq$

* This work has been partially founded by PICS 2533 of the CNRS and project EVA-Flo of the ANR.

$p_{\max}(N)$ for N between 2 and 54. Gappa generated a proof that no rounding error occurred whatever the value of the input matrix for each of the 53 different values of N.

Proof obligations are usually derived from a static analysis of the source code considered. Our work showed that generating proof obligations from traces of execution after most parameters have been instantiated may also be useful. We have set up a C++ class to provide such proof obligations but we hope that such capability will be provided by Why and similar tools in the future.

We present in Section 2 the background and one algorithm of delayed solutions of a triangular system over a finite field. We continue in Section 3 by the background and tools of formal proof checking including Gappa and excerpts of our C++ trace class. We conclude this work in Section 4.

2 Finite field arithmetic and application to linear algebra

Finite field arithmetic plays a crucial role in nowadays applications. One of the most extensively studied application of finite fields is cryptography with major concerns on security and efficiency (see [6] for an introduction on cryptography). Another key application of finite field arithmetic arises with exact linear algebra computation where modular techniques (e.g. CRT or P-adic lifting) allow some control on expression swell with high performances [7,8,9]. Arithmetic implementations used in cryptography differ from the ones used in linear algebra. While cryptographic applications need finite fields of large cardinality (e.g. 1024-2048 bits for RSA [10], 300 bits for ECC [11]) for security purpose, most exact linear algebra restrains to machine word size prime field (e.g. 32 or 64 bits) in order to benefit from machine arithmetic units.

A classical way to perform one arithmetic operation in a prime field, here we refer to integers modulo a prime number, is to first perform the operation on integers and second reduce the result to the destination field. Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ and $* \in \{+, \times\}$. One may compute $z = x * y \in \mathbb{Z}/p\mathbb{Z}$ by computing $t = x * y \in \mathbb{Z}$ and a modular reduction $z = t \mod p$.

In the rest of this section, we present basic facts on word size prime field arithmetics and how to efficiently integrate them into an exact linear algebra application: solutions of triangular systems.

2.1 Word-size prime field arithmetic

When one deals with fixed precision prime field arithmetic, two majors issues arise: performances and cardinality limitation. The latter issue can have a non-negligible impact on the former one. As was just said, the classical way to perform arithmetic operations over a prime field is to perform operations on integers and reduce intermediate results. Therefore, $(p-1)^2$ must be representable to correctly perform multiplications over $\mathbb{Z}/p\mathbb{Z}$. This limitation slightly increase to perform an `AXPY` operation (a multiplication followed by an addition) with only one reduction step. This implies that $p \times (p-1)$ must be representable.

Using word-size machine integers and classic arithmetic we obtain the following cardinality limitation: $p < 2^{16}$ on 32 bit architectures and $p < 2^{32}$ on 64 bit architectures with `unsigned` types. These bounds are reduced by one bit with signed types such as `long`.

Note that even on 32 bit architectures, one can obtain 64 bit integers by using `long long` data. Full use of `long long` may reduce performances drastically as the reduction phase involves a 64 bit division. Integer multiplications have no impact on performances since most of 32 bit processors provide a multiplier $32\text{bits} \times 32\text{bits} \rightarrow 64\text{bits}$ (see the `imul` instruction on Intel architectures [12]).

Another alternative to increase cardinality of word-size prime fields is to use floating point numbers. According to the IEEE 754 standard [13,14], mantissas of double precision floating point numbers can store 53 bit integers (including the implicit bit). Therefore, we can perform prime field arithmetic with cardinality up to 2^{26} using `double`. Note that, the reduction is easily obtained by the `fmod` function available in standard libraries. This approach is quite interesting in practice since floating point multiplications and divisions may be faster than their integer counterparts and delayed algorithms such as the ones presented here allows to use optimized numerical BLAS.

One can get even larger prime fields by mixing integers and floating point numbers. The idea is then to use an approximation of the quotient with floating point and Barret's method to compute the remainder [6, chapter 14] as demonstrated in NTL library [15]. It leads to prime fields with cardinality up to 2^{29} on 32 bit architectures and up to 2^{52} on 64 bit architectures.

Prime field implementations above benefit from the arithmetic units of processors but performances depend on the target architecture (see [16,17] for detailed comparisons). On selected classes of algorithms, delayed prime field arithmetic sustains better performances. The idea is to perform several integer operations before reduction into the field. It has been very fruitful for exact linear algebra [8]. We perform a dot-product or a matrix multiplication with a delayed reduction when $(p-1)^2 \times N < 2^\beta$, where n is the dimension of matrices or vectors, p defines the prime field $\mathbb{Z}/p\mathbb{Z}$ and β represents the precision of integers. Delayed exact linear algebra computations also benefit from optimized numerical BLAS (e.g. ATLAS [18], GOTO [19]) libraries for exact computations as shown in [8] and they often reach maximal FPU throughput for operations over a finite field.

Beside basics linear algebra operations such as matrix-vector products and matrix multiplications, delayed arithmetic over a prime field is valuable when expressions swell largely such as solving systems of linear equations. This approach works perfectly for unitary triangular system (only ones along the diagonal) despite the exponential growth of the intermediate variables.

2.2 Triangular system solving with delayed prime field arithmetic

A key application in exact linear algebra is the resolution of triangular systems over finite fields. The resolution of such systems is a classical problem of

linear algebra and it is one of the main operations in block Gaussian elimination when right hand side is given as a matrix [20]. Resolution of triangular systems with matrix as right hand side reduces to matrix multiplication and provides the best known complexity for this problem. The complexity is going from $O(N^3)$ finite field operations with classical algorithm to $O^{\sim}(N^{2.81})$ using Strassen-Winograd variant and reaches the best asymptotic known value of $O^{\sim}(N^{2.37})$ with Coppersmith-Winograd [21]. The following algorithm achieves this reduction.

Algorithm `LZ_TRSM`(A, B)
Input: $A \in \mathbb{Z}/p\mathbb{Z}^{N\times N}$, $B \in \mathbb{Z}/p\mathbb{Z}^{N\times K}$.
Output: $X \in \mathbb{Z}/p\mathbb{Z}^{N\times K}$ such that $AX = B$.

---

if N=1 **then**
  $X := A_{1,1}^{-1} \times B$.
else *(splitting matrices into* $\lfloor \frac{N}{2} \rfloor$ *and* $\lceil \frac{N}{2} \rceil$ *blocks)*

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

  X_2 :=`LZ_TRSM`(A_3, B_2).
  $B_1 := B_1 - A_2X_2$.
  X_1 :=`LZ_TRSM`(A_1, B_1).
return X.

A delayed prime field arithmetic version of this algorithm can be constructed by simply doing a delayed matrix multiplication on the operation $B_1 := B_1 - A_2X_2$. One can easily see that modular reductions will be performed at each level of the reduction. However, it appears that enough bits are available to combine a few recursion steps without reduction. This has been the purpose of the implementation based on BLAS triangular solver function (i.e. `dtrsm` routine) proposed in a previous work from one of the authors (see [8, §3.2]).

Bounds on the coefficient growth during backward substitution allow to guarantee that numerical results are exact. In [8], an optimal bound on integer coefficients has been proposed. For the sake of completeness we recall it here.

Corollary 21 *[8, corollary 3.3]: Let $A \in \mathbb{Z}^{N\times N}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^N$, with $|A|, |b| \leq p-1$. Then $x \in \mathbb{Z}^N$ the solution of the system $Ax = b$ is such that*

$$|x| \leq \frac{p-1}{2}\left[p^{N-1} + (p-2)^{N-1}\right],$$

and this bound is optimal.

We construct an improved version for the delayed prime field arithmetic of algorithm `LZ_TRSM` by replacing the last levels of the recursion by calls to `dtrsm`

numerical solver according to the above corollary. Recursion is stopped for the maximal integer $n_{\max}$ such that

$$\frac{p-1}{2}\left[p^{n_{max}-1}+(p-2)^{n_{max}-1}\right] < 2^{53}. \quad (1)$$

3 Formal proof checking and Gappa

Gappa [22,23] has been created to generate formal certificates of correction for programs that use floating point arithmetic [24,25,26,27]. As it was developed we extended it to simple bounds in general calculus. It will in the future be able to interact seamlessly with Why [5], a tool to certify programs written in a generic language. C and Java can be converted to this language.

Gappa manipulates arithmetic expressions on real and rational numbers and their evaluations on computers. Exact and rounded expressions are bounded using interval arithmetic [28], forward error analysis and properties of dyadic fractions. Proof obligation are generated by stating that expressions that were **intended** to be computed with no round-off error are equal to expressions that were **actually** computed possibly with round-off errors. Gappa generates a proof for each expression.

To the authors' best knowledge, Gappa is the first tool that can convert the simple task performed here into a formal proof validated by an automatic proof checker. Such behavior has previously been quoted as *invisible formal methods* [29] in the sense that Gappa delivers formal certificates to users that are not expected to ever write any piece of proof in any formal proof system.

Listing 1.1 is the direct transcription of the algorithm presented in Section 2 to solve unitary triangular systems. We used naming conventions of BLAS and LAPack for the function and the parameter names. For the sake of simplicity some parameters have be omitted and some function names were slightly modified.

Appropriate assertions on the `DGEMM_NEG` function generate proof obligation

$$Y[i \times LDY + k]/(j+1) \in [-p+1, p-1]$$

for all iterations defined by i, j and k. It can be proved easily by induction on j with the help of Gappa as soon as variable $Y[i \times LDY + k]$ as been abstracted to a generic name Z.

The `DTRSM` function is invoked by `LZ_TRSM` only on the condition $Pred(N,p) = N \leq n_{\max}(p)$. Below is the function that computes `nmax` for a given value of `p` where `cmax` has been set to 2^{53}.

```
pp = 1; p2 = 1;
for (nmax = 0; ((p-1)*(pp+p2))/2 < cmax; nmax++)
  {pp*=p; p2*=p-2;};
```

Gappa does not handle loops and branches. We perform a case analysis on the value of `N`. The code below computes the value $p_{\max}(n)$ such as $Pred(N,p) = p \leq p_{\max}(N)$. It generates Table 1.

Listing 1.1. Delayed solution of a unitary triangular system over a finite field

```
// Solutions to a small unitary triangular system
void DTRSM (int N, int K,
            trace *A, int LDA,
            trace *X, int LDX) {
  A->bmes("# DTRSM ", N);
  for (int i = N - 2; i >= 0; i--)
    for (int j = i+1 ; j < N; j++)
      for (int k = 0; k < K; k++)
        X[i*LDX+k] = X[i*LDX+k] - A[i*LDA+j] * X[j*LDX+k];
}

// Remainder of a matrix modulo p
void DREMM (int N, int K, int p,
            trace *X, int LDX) {
  X->bmes("# DREMM ", N);
  for (int i = 0; i < N; i++)
    for (int k = 0; k < K; k++)
      X[i*LDX+k].init(p);
}

// Matrix-matrix multiplication Y <- Y - AX
void DGEMM_NEG (int N, int M, int K,
                trace *A, int LDA,
                trace *X, int LDX,
                trace *Y, int LDY) {
  A->bmes ("# DGEMM ", N);
  for (int i = 0; i < N; i++)
    for (int j = 0 ; j < M; j++)
      for (int k = 0; k < K; k++)
        Y[i*LDY+k] = Y[i*LDY+k] - A[i*LDA+j] * X[j*LDX+k];
}

// Inductive solutions to a unitary triangular system
void LZ_TRSM (int N, int K, int nmax, int p,
              trace *A, int LDA,
              trace *B, int LDB) {
  if (N <= nmax) {
    DTRSM (N, K, A, LDA, B, LDB);
    DREMM (N-1, K, p, B, LDB);
  } else {
    int P = N/2, G = N - P;
    LZ_TRSM (G, K, nmax, p, A+P*(LDA+1), LDA, B+P*LDB, LDB);
    DGEMM_NEG (P, G, K, A+P, LDA, B+P*LDB, LDB, B, LDB);
    DREMM (P, K, p, B, LDB);
    LZ_TRSM (P, K, nmax, p, A, LDA, B, LDB);
  }
}
```

Table1. Generated values of parameter `pmax` for allowed valued parameter `N`

N	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_{\max}$	94906266	208064	9739	1553	457	191	97	59	39	29	19	17	13	11

N	$16 \cdots 19$	$20 \cdots 23$	$24 \cdots 34$	$35 \cdots 54$
$p_{\max}$	7	5	3	2

```
for (p = (int) sqrt(cmax)+10, nmax = 1; p > 1; p--) {
  if ((p > 10000) || (isprime(p))) {
    tmax = nmax(p);
    while (tmax > nmax)
      cout << "pmax=" << p << " n=" << ++nmax << endl;
  }
}
```

The `DREMM` function computes the remainder modulo p of all the components of a matrix. As visible from the proof of Corollary 21, we do not use the exact value of these remainders but only the fact that they are between 0 and $p-1$.

Most questions raised here are not new to automated reasoning. We focus in this report on the case analysis of `DTRSM` and the new technique leading to the proof that this function never produces any round-off error.

3.1 A C++ trace class to generate input scripts to Gappa

Gappa produces a Coq file, for example Listing 1.5, for a given input script, in this case Listing 1.3. The Coq file contains properties and proofs. Validity of proofs can automatically be checked by Coq. The C++ trace class of Listing 1.2 is responsible to produce Listing 1.3.

The later file contains three sections each associated with a `static ostringstream` variable in our C++ class. The first section is built in `bout` (beginning), the second in `mout` (middle) and the last one in `eout` (end).

The first section defines aliases. Gappa uses these aliases for its outputs and in the formal proof instead of machine generated names. It starts with the definitions of `rnd` rounding operator. It is a real function yielding rounded values according to the target data format (`ieee_64` - double precision in this case) and a predefined rounding mode (`ne` - nearest with even tie breaking as specified in IEEE 754 standard in this case).

Statements `DGxxx_exact = expr;` build the list of **intended** results and statements on `DGxxx` the list of results **actually** computed. When a rounding operator appears left of the equal symbol (as used in `DGxxx rnd= expr;`), all the arithmetic operations on the right side are rounded individually. We introduced dummy identifiers in aliases `DGxxx = int<ne>(DGxxx_dum);` to express that `DGxxx` are integer numbers.

Listing 1.2. A C++ trace class to instantiate some parameters, implement single-assignment behavior and ask Gappa to prove that no operation introduced any round-off error

```
class trace;
ostream &operator<<(ostream &os, trace const &a);

class trace {
public :
  static int lastid;
  static ostringstream bout, mout, eout;
  int id;
  trace() {}
  void init(int p) {
    id = lastid++;
    bout << *this
         << " = int<ne>(" << *this << "_dum);" << endl;
    mout << *this
         << " in [0, " << p - 1 << "] ->"      << endl;
  }
  trace(char * oper, trace const &a, trace const &b) {
    id = lastid++;
    bout << *this << "    rnd= "
         << a     << oper  << b << ";"         << endl;
    bout << *this << "_exact = "
         << a     << oper  << b << ";"         << endl;
    eout << *this << " - "
         << *this << "_exact in [0, 0] /\\"    << endl;
    eout << *this << " in ? /\\"               << endl;
  }
  void dump() {
    cout << bout.str() << "{"                  << endl
         << mout.str()
         << eout.str()
         << *this      << " in ?"              << endl
                       << "}"                  << endl;
  }
  void bmes (char * mes, int n = 0)
  {bout << mes ; if (n) bout << n; bout << endl;}
};

ostream &operator<<(ostream &os,trace const &a)
  {return os << "DG" << a.id;}

trace operator+(trace const &a, trace const &b)
  {return trace(" + ", a, b);}
trace operator-(trace const &a, trace const &b)
  {return trace(" - ", a, b);}
trace operator*(trace const &a, trace const &b)
  {return trace(" * ", a, b);}

int trace::lastid = 0;
ostringstream trace::bout, trace::mout, trace::eout;
```

Listing 1.3. Reordered Gappa input script for a small example (4×4 matrices on $\mathbb{Z}/101\mathbb{Z}$)

```
@rnd = float< ieee_64, ne>;
# Matrix
DG0 = int<ne>(DG0_dum);
...
DG15 = int<ne>(DG15_dum);
# Solution
DG16 = int<ne>(DG16_dum);
...
DG19 = int<ne>(DG19_dum);
# DTRSM 4
DG20     rnd= DG5 * DG19;
DG21     rnd= DG18 - DG20;
DG22     rnd= DG3 * DG21;
DG23     rnd= DG17 - DG22;
DG24     rnd= DG4 * DG19;
DG25     rnd= DG23 - DG24;
DG26     rnd= DG1 * DG25;
DG27     rnd= DG16 - DG26;
DG28     rnd= DG2 * DG21;
DG29     rnd= DG27 - DG28;
DG30     rnd= DG3 * DG19;
DG31     rnd= DG29 - DG30;
DG20_exact = DG5 * DG19;
...
DG31_exact = DG29 - DG30;
# DREMM 3
DG32 = int<ne>(DG32_dum);
DG33 = int<ne>(DG33_dum);
DG34 = int<ne>(DG34_dum);
{
DG0 in [0, 100] ->
...
DG19 in [0, 100] ->
DG32 in [0, 100] ->
...
DG34 in [0, 100] ->
DG20 - DG20_exact in [0, 0] /\
...
DG31 - DG31_exact in [0, 0] /\
DG20 in ? /\
...
DG31 in ? /\
DG0 in ?
}
```

Listing 1.4. Test

```
void TEST (int n, int nmax, int p, int k) {
  int i;
  trace *A = new trace[n*n], *B = new trace[k*n];
  A->bmes("@rnd = float< ieee_64, ne>;");
  A->bmes("# Matrix");
  for (i = 0; i < n*n; i++) {A[i].init(p);}
  A->bmes("# Solution");
  for (i = 0; i < k*n; i++) {B[i].init(p);}
  LZ_TRSM (n, k, nmax, p, A, k, B, k);
  A->dump();
  delete []A; delete []B;
}
```

The second and third sections are written together between brackets (`{ }`) and in this work, it is a large implication (`->`) of a large conjunction (`/\`) of interval enclosures of mathematical expressions.

The second section contains a set of hypotheses each stating that a variable or an expression is within an interval. Note that $p1 \rightarrow p2 \rightarrow p3$ is logically equivalent to $p1 \wedge p2 \rightarrow p3$ and this section is built from the lines finished by an implication sign (`->`). Identifiers without definition, such as the `DGxxx_dum` variables used in the first section are assumed to be universally quantified over the set of real numbers the first time Gappa encounters them.

The third section contains goals. It is a large conjunction (`/\`). Statements `DGxxx - DGxxx_exact in [0, 0]` mean that we ask Gappa to prove that `DGxxx` and `DGxxx_exact` are identical. Statements `DGxxx in ?` are inserted for diagnostics as Gappa proposes enclosing intervals.

The `TEST` function in Listing 1.4 presents memory allocation and parts of the Coq script related to initial aliases and some comments.

3.2 Insight into the Coq generated script

The generated Coq script visible in Listing 1.5 contains `Variables`, `Definitions`, `Notations` and `Lemmas`. Users do not need to be able to write such a file but they can check the work of Gappa by reading it or parsing it. Comments are between `(*` and `*)` signs. They were generated by Gappa.

Lemma `l134` is one of the tasks assigned to Gappa. It proves that property `p134` is valid under a large number of assumptions (`p3`, `p18`, `p43`, `p92`...) that is a subset of the hypotheses given in the second section of Gappa input script. The lines between the statement of a lemma and the `Qed.` sign are proof scripts intended for Coq only. All the properties are defined in the script. The `BND` predicate of `p134` holds, when its first argument, an expression on real numbers, is an element of its second argument, an interval defined by dyadic fraction bounds.

Proof of Lemma `l134` requires a bound on `DG31 - DG31_exact` provided by Lemma `t696`. Although enclosure (`BND`) is the only predicate available to users,

Listing 1.5. Excerpts of the 3548 line Coq script generated for a small example (4×4 matrices on $\mathbb{Z}/101\mathbb{Z}$)

```
Require Import Gappa_library.
Section Generated_by_Gappa.
Definition f1 := Float2 (25) (2).
Definition f2 := Float2 (0) (0).
...
Variable _DG19_dum : R.
Notation _DG19 := ((rounding_fixed roundNE (0)) (_DG19_dum)).
Notation p3 := (BND _DG19 i1). (* BND(DG19, [0, 100]) *)
...
Definition i3 := makepairF f2 f2.
...
Definition f15 := Float2 (1) (-27).
Definition f16 := Float2 (-1) (-27).
Definition i13 := makepairF f16 f15.
...
Variable _DG16_dum : R.
Notation _DG16 := ((rounding_fixed roundNE (0)) (_DG16_dum)).
Notation p92 := (BND _DG16 i1). (* BND(DG16, [0, 100]) *)
...
Notation _DG31_exact := ((_DG29 - _DG30)%
Notation _DG31 :=
  ((rounding_float roundNE (53) (1074)) (_DG31_exact)).
...
Notation r70 := ((_DG31 - _DG31_exact)%
Notation p134 := (BND r70 i3).
  (* BND(DG31 - DG31_exact, [0, 0]) *)
Notation p135 := (BND r70 i13).
  (* BND(DG31 - DG31_exact, [-7.45058e-09, 7.45058e-09]) *)
...
Notation p141 := (FIX r70 (0)). (* FIX(DG31 - DG31_exact, 0) *)
...
Lemma l133 : p141. (* FIX(DG31 - DG31_exact, 0) *)
 assert (h0 : p142). apply l132.
 assert (h1 : p143). apply l131.
 apply t695. exact h0. exact h1.
Qed.
Lemma t696 : p135 -> p141 -> p134.
 intros h0 h1.
 apply bnd_of_bnd_fix with (1 := h0) (2 := h1) ; finalize.
Qed.
Lemma l134 : ... -> p92 -> p43 -> p18 -> p3 -> p134.
  (* BND(DG31 - DG31_exact, [-0, 0]) *)
 intros h0 h1 h2 h3 h4 h5 h6 h7 h8.
 assert (h9 : p135). apply l130. exact h0. exact h1...
 assert (h10 : p141). apply l133.
 apply t696. exact h9. exact h10.
Qed.
End Generated_by_Gappa.
```

Gappa internally relies on more predicates to describe properties on expressions.

Predicate	Definition
$\texttt{BND}(x,[a,b])$	$a \leq x \leq b$
$\texttt{ABS}(x,[a,b])$	$0 \leq a \leq \lvert x\rvert \leq b$
$\texttt{FIX}(x,e)$	$\exists m \in \mathbb{Z}, x = m \cdot 2^e$
$\texttt{FLT}(x,p)$	$\exists m,e \in \mathbb{Z}, x = m \cdot 2^e \wedge \lvert m\rvert < 2^p$

The proof of Lemma `t696` uses `bnd_of_bnd_fix` of Gappa support library with the `apply` tactic. Automatic validation of the hypotheses is triggered by the `finalize` tactic that checks that the current goal can be reduced to $true = true$. More insights to Gappa are presented in [23].

4 Perspectives and concluding remarks

This report presents a new use of Gappa based on proof obligations generated from a trace of execution. It presents Gappa on an example in sufficient details so that this report is a tutorial on how to use Gappa, how to read files generated by Gappa and how Gappa works internally.

`http://lipforge.ens-lyon.fr/www/gappa/`

We were able to prove in Coq that expression swell within the studied algorithm of delayed finite field arithmetic does not introduce round off errors as stated in Corollary 21. This certification was not obtained by porting the proof of Corollary 21 to Coq but by a case analysis on the 53 possible values of the size of the matrix N. A formal proof was generated by Gappa for each individual value of N. The other questions raised in this report are not related to floating arithmetic. Answering these questions involve techniques common to program analysis that are or should be available in tools that perform automatic validation of programs.

Our approach can be easily reproduced to other exact linear applications over finite fields. More precisely, the FFLAS-FFPACK project has been successful on using delayed prime field arithmetic for linear algebra applications.

`http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/FFLAS/`

We will perform validation of symmetric representations of prime field in the future.

References

1. Lamport, L., Melliar-Smith, P.M.: Synchronizing clocks in the presence of faults. Journal of the ACM **32**(1) (1985) 52–78
2. Rushby, J., von Henke, F.: Formal verification of algorithms for critical systems. In: Proceedings of the Conference on Software for Critical Systems, New Orleans, Louisiana (1991) 1–15
3. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq proof assistant: a tutorial: version 8.0. (2004)

4. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer-Verlag (2004)
5. Boldo, S., Filliâtre, J.C.: Formal verification of floating point programs. In: Proceedings of the 18th Symposium on Computer Arithmetic, Montpellier, France (2007)
6. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc. (1996)
7. Eberly, W., Giesbrecht, M., Giorgi, P., Storjohann, A., Villard, G.: Solving sparse rational linear systems. In: ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation, New York, NY, USA, ACM Press (2006) 63–70
8. Dumas, J.G., Giorgi, P., Pernet, C.: FFPACK: Finite field linear algebra package. In: ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation, New York, NY, USA, ACM Press (2004) 63–74
9. Chen, Z., Storjohann, A.: A BLAS based C library for exact linear algebra on integer matrices. In: ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation, New York, NY, USA, ACM Press (2005) 92–99
10. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signature and public-key cryptosystems. Communication of the ACM **21**(2) (1978)
11. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
12. Intel: Intel Architecture Software Developer's Manual. (1999) Instruction Set Reference.
13. Stevenson, D., et al.: An American national standard: IEEE standard for binary floating point arithmetic. ACM SIGPLAN Notices **22**(2) (1987) 9–25
14. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1) (1991) 5–47
15. Shoup, V.: NTL 5.3: A library for doing number theory (2002) `www.shoup.net/ntl`.
16. Giorgi, P.: Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LinBox. PhD thesis, École normale supérieure de Lyon (December 2004)
17. Dumas, J.G.: Efficient dot product over finite fields. In: Computer Algebra in Scientific Computing, CASC'04, Saint Pertersburg, Russie (July 2004)
18. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27**(1–2) (January 2001) 3–35 `www.elsevier.nl/gej-ng/10/35/21/47/25/23/article.pdf`.
19. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical report, University of Texas (2002) FLAME working note #9.
20. Bunch, J.R., Hopcroft, J.E.: Triangular factorization and inversion by fast matrix multiplication. Mathematics of Computation **28** (1974) 231–236
21. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation **9**(3) (1990) 251–280
22. Daumas, M., Melquiond, G.: Generating formally certified bounds on values and round-off errors. In: Real Numbers and Computers, Dagstuhl, Germany (2004) 55–70
23. Daumas, M., Melquiond, G.: Generating certified properties for numerical expressions and their evaluations. Technical Report hal-00127769, Centre pour la Communication Scientifique Directe, Villeurbanne, France (2007)
24. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France (2006) 1318–1322

25. Michard, R., Tisserand, A., Veyrat-Charvillon, N.: Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In: Symposium en Architecture de Machines, Perpignan, France (2006) 1318–1322
26. Revy, G.: Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Technical Report ensl-00119498, École Normale Supérieure de Lyon (2006)
27. Melquiond, G., Pion, S.: Formally certified floating-point filters for homogeneous geometric predicates. Theoretical Informatics and Applications (2007) To appear.
28. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied interval analysis. Springer (2001)
29. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE **91**(1) (2003) 29–39