The Matrix Reproved

Martin Clochard, Léon Gondelman, Mário Pereira

Noname manuscript No.
(will be inserted by the editor)

The Matrix Reproved (Verification Pearl)

Martin Clochard · Léon Gondelman · Mário Pereira

the date of receipt and acceptance should be inserted later

Abstract In this paper we describe a complete solution for the first challenge of the VerifyThis 2016 competition held at the 18th ETAPS Forum. We present the proof of two variants for the multiplication of matrices: a naive version using three nested loops and Strassen's algorithm. The proofs are conducted using the Why3 platform for deductive program verification and automated theorem provers to discharge proof obligations. In order to specify and prove the two multiplication algorithms, we develop a new Why3 theory of matrices. In order to prove the matrix identities on which Strassen's algorithm is based, we apply the proof by reflection methodology, which we implement using ghost state.To our knowledge, this is the first time such a methodology is used under an auto-active setting.

1 Introduction

In this paper we describe a complete solution for the first challenge of the VerifyThis 2016 competition[1] using the Why3 platform for deductive verification.

As it was asked in the original challenge, we prove the correctness of two different implementations of matrix multiplication, and prove that multiplication is associative. First, we specify and prove a naive algorithm which runs in cubic time, then the more efficient Strassen's algorithm. To our knowledge, this is the first proof of Strassen's algorithm for matrices of arbitrary size in a program verification environment based on automated theorem provers.

To make our solutions both concise and generic, we devise in Why3 an axiomatic theory for matrices and show various algebraic properties for their arithmetic operations, in particular multiplication distributivity over addition and associativity (which was asked in the challenge second task). Our full development is available online[2].

Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405 · INRIA, Université Paris-Saclay, Palaiseau F-91893

[1] http://etaps2016.verifythis.org/

[2] http://toccata.lri.fr/gallery/verifythis_2016_matrix_multiplication.en.html

It turns out that proving Strassen's algorithm seems to be too challenging for automated theorem provers due to their incapacity to perform reasoning on algebraic matrix identities. To overcome this obstacle, we devise an algebraic expression simplifier in order to conduct proof by reflection.

This paper is organized as follows. Section 2 briefly presents Why3. Section 3 describes our solution for naive matrix multiplication. Then, section 4 presents our solution for the second task and introduces our axiomatic matrix theory. We specify and prove Strassen's algorithm in sections 5 and 6. Section 7 presents proof statistics about our development. We discuss related work in section 8.

2 Why3 in a Nutshell

The Why3 platform proposes a set of tools allowing the user to implement, formally specify, and prove programs. It comes with a programming language, WhyML [8], an ML dialect with some restrictions in order to get simpler proof obligations. This language offers some features commonly found in functional languages, like pattern-matching, algebraic types, and polymorphism, but also imperative constructions, like records with mutable fields and exceptions. Programs written in WhyML can be annotated with contracts, that is, pre- and postconditions. The code itself can be annotated, for instance, to express loop invariants or to justify termination of loops and recursive functions. It is also possible to add intermediate assertions in the code to ease automatic proofs. The WhyML language features ghost code [7], which is used only for specification and proof purposes and can be removed with no observable modification in the program's execution. The system uses the annotations to generate proof obligations thanks to a weakest precondition calculus.

Why3 uses external provers to discharge proof obligations, either automatic theorem provers (ATP) or interactive proof assistants such as Coq, Isabelle, and PVS. The system also allows the user to manually apply *logical transformations* to proof obligations before they are sent to provers, in order to make proofs easier.

The logic used to write formal specifications is an extension of first-order logic with rank-1 polymorphic types, algebraic types, (co-)inductive predicates and recursive definitions [6], as well as a limited form of higher-order logic [4]. This logic is used to write theories for the purpose of modeling the behavior of programs. The Why3 standard library is formed of many such logic theories, in particular for integer and floating point arithmetic, sets, and sequences.

The entire standard library, numerous verified examples, as well as a more detailed presentation of Why3 and WhyML are available on the project web site, http://why3.lri.fr.

3 Naive Matrix Multiplication

The VerifyThis 2016 first challenge starts with a proposal to verify a naive implementation of the multiplication of two matrices using three nested loops, though using a non-standard order for indices.

Consider the following pseudocode algorithm, which is naive implementation of matrix multiplication. For simplicity we assume that the matrices are square.

```
int[][] matrixMultiply(int[][] A, int[][] B) {
   int n = A.length;

   // initialise C
   int[][] C = new int[n][n];

   for (int i = 0; i < n; i++) {
          for (int k = 0; k < n; k++) {
                 for (int j = 0; j < n; j++) {
                     C[i][j] += A[i][k] * B[k][j];
                 }
          }
   }
   return C;
}
```

Tasks.

1. Provide a specification to describe the behaviour of this algorithm, and prove that it correctly implements its specification.

2. Show that matrix multiplication is associative, i.e., the order in which matrices are multiplied can be disregarded: $A(BC) = (AB)C$. To show this, you should write a program that performs the two different computations, and then prove that the result of the two computations is always the same.

3. [Optional, if time permits] In the literature, there exist many proposals for more efficient matrix multiplication algorithms. Strassen's algorithm was one of the first. The key idea of the algorithm is to use a recursive algorithm that reduces the number of multiplications on submatrices (from 8 to 7), see Strassen_algorithm on wikipedia for an explanation. A relatively clean Java implementation (and Python and C++) can be found here. Prove that the naive algorithm above has the same behaviour as Strassen's algorithm. Proving it for a restricted case, like a 2×2 matrix should be straightforward, the challenge is to prove it for arbitrary matrices with size 2^n.

Fig. 1 The original text of the first challenge.

The original text of the first challenge is given in the Fig. 1. We first write the WhyML equivalent of the challenge code for the multiplication of two matrices `a` and `b`[3]:

```
let mult_naive (a b: matrix int) : matrix int
= let rs = make a.rows b.columns 0 in
  for i = 0 to a.rows - 1 do
    for k = 0 to b.rows - 1 do
      for j = 0 to b.columns - 1 do
        set rs i j (get rs i j + get a i k * get b k j)
      done;
    done;
  done;
  rs
```

We use matrices as provided in the Why3 standard library. Those implement two-dimensional arrays. Operations `get` and `set` have the usual semantics, and `make` carries out creation and initialization. Such matrices are represented by the following type:

```
type matrix 'a
  model { rows: int; columns: int; mutable elts: map int (map int 'a) }
val rows (a: matrix 'a) : int ensures { result = a.rows }
val columns (a: matrix 'a) : int ensures { result = a.columns }
```

Here, the keyword `model` indicates that `matrix` is an abstract type, whose fields can only be accessed in specifications. The keyword `mutable` indicates that the content of `elts` can be modified. As the dimensions should be accessible in programs as well, we use abstract accessors to get them. Note that in Why3 the notation `a.f` is syntactic sugar for `(f a)`. Let us now specify the multiplication procedure:

```
let mult_naive (a b: matrix int) : matrix int
  requires { a.columns = b.rows }
  ensures  { result.rows = a.rows ∧ result.columns = b.columns }
  ensures  { matrix_product result a b }
```

The predicate `matrix_product` mimicks the mathematical definition of matrix product

$$(AB)_{ij} = \sum_{k=0}^{m-1} A_{ik} B_{kj}$$

which we translate into WhyML as follows:

```
function mul_atom (a b: matrix int) (i j: int) : int → int =
  λk. a.elts[i][k] * b.elts[k][j]

predicate matrix_product (m a b: matrix int) =
  ∀i j. 0 ≤ i < m.rows → 0 ≤ j < m.columns →
    m.elts[i][j] = sum 0 a.columns (mul_atom a b i j)
```

[3] For simplicity, the original task assumes that the matrices are square. Our implementation deals more generally with rectangular matrices.

In order to define this predicate concisely, we use the higher-order function sum from Why3 standard library. Given a function `f: int → int`, this function returns the sum of `f n` for `n` ranging between `a` and `b`, as specified by the following axioms:

```
function sum (a b: int) (f: int → int) : int
axiom sum_def1: ∀f a b. b ≤ a → sum a b f = 0
axiom sum_def2: ∀f a b. a < b →
  sum a b f = sum a (b - 1) f + f (b - 1)
```

To prove that `mult_naive` meets its specification, we give suitable loop invariants. There are two kinds of invariant per loop. The first kind is the frame invariant, which describes the part of the matrix that is left unchanged. The second one describes the contents of cells affected by the loop. To reason about the state just before the execution enters the two inner loops, we also add labels `'M` and `'I` for the middle one and the innermost one respectively. We then refer to the state of matrix `rs` when the program is at point `'M` by the notation `(at rs 'M)`.

For instance, in the annotated code below, the invariants of the innermost loop describe that its effect consists in writing a partial sum into cells `0` to `j-1` of the `i`-th row, leaving other cells unchanged.

```
for i = 0 to a.rows - 1 do
  invariant { ∀i₀ j₀. 0 ≤ i₀ < i ∧ 0 ≤ j₀ < b.columns →
    rs.elts[i₀][j₀] = sum 0 a.columns (mul_atom a b i₀ j₀) }
  invariant { ∀i₀ j₀. i ≤ i₀ < a.rows ∧ 0 ≤ j₀ < b.columns →
    rs.elts[i₀][j₀] = 0 }
  'M:
  for k = 0 to b.rows - 1 do
    invariant { ∀i₀ j₀. 0 ≤ i₀ < a.rows ∧ 0 ≤ j₀ < b.columns ∧
      i₀ ≠ i → rs.elts[i₀][j₀] = (at rs 'M).elts[i₀][j₀] }
    invariant { ∀j₀. 0 ≤ j₀ < b.columns →
      rs.elts[i][j₀] = sum 0 k (mul_atom a b i j₀) }
    'I:
    for j = 0 to b.columns - 1 do
      invariant { ∀i₀ j₀. 0 ≤ i₀ < a.rows ∧
        0 ≤ j₀ < b.columns ∧ (i₀ ≠ i ∨ j₀ ≥ j) →
        rs.elts[i₀][j₀] = (at rs 'I).elts[i₀][j₀] }
      invariant { ∀j₀. 0 ≤ j₀ < j →
        rs.elts[i][j₀] = sum 0 (k+1) (mul_atom a b i j₀) }
      set rs i j (get rs i j + get a i k * get b k j)
    done;
  done;
done;
```

With the annotations above all the generated verification conditions are discharged in a fraction of second using the Alt-Ergo SMT solver.

4 From Multiplication Associativity to a Matrix Theory

The next task was to show that matrix multiplication is associative. More precisely, participants were asked to write a program that performs the two different

$$
\begin{aligned}
((AB)C)_{ij} &= \sum_k (\sum_\ell A_{i\ell} B_{\ell k}) C_{kj} && \text{(unfold definition of matrix product)} \\
&= \sum_k \sum_\ell A_{i\ell} B_{\ell k} C_{kj} && \text{(linearity of sum operator)} \\
&= \sum_\ell \sum_k A_{i\ell} B_{\ell k} C_{kj} && \text{(Fubini's principle)} \\
&= \sum_\ell A_{i\ell} \sum_k B_{\ell k} C_{kj} && \text{(linearity of sum operator)} \\
&= (A(BC))_{ij} && \text{(fold definition of matrix product)}
\end{aligned}
$$

Fig. 2 Proof sketch for matrix multiplication associativity.

computations $(AB)C$ and $A(BC)$, and then to prove the corresponding results to be always the same. In our case, this corresponds to proving the following program:

```
let assoc_proof (a b c: matrix int) : unit
  requires { a.columns = b.rows ∧ b.columns = c.rows }
= let ab_c = mult_naive (mult_naive a b) c in
  let a_bc = mult_naive a (mult_naive b c) in
  assert { ab_c.rows = a_bc.rows ∧ ab_c.columns = a_bc.columns ∧
    ∀i j. 0 ≤ i < ab_c.rows ∧ 0 ≤ j < ab_c.columns →
      ab_c.elts[i][j] = a_bc.elts[i][j] }
```

The proof of associativity relies essentially on the linearity of the sum operator and Fubini's principle according to which, given a finite sum indexed by i and j, we have $\sum_{i,j} a_{i,j} = \sum_i (\sum_j a_{i,j}) = \sum_j (\sum_i a_{i,j})$. Figure 2 shows a proof sketch for the multiplication associativity using mathematical notations. Let us illustrate how we establish the linearity of the sum (Fubini's principle is done in a similar way). First we define a higher-order function `smulf`:

```
function smulf (l: int) (f: int → int) : int → int = λx. l * f x
```

Then we prove the linearity using a lemma function:

```
let rec lemma sum_mult (a b l: int) (f: int → int) : unit
  ensures { sum a b ( smulf l f) = l * sum a b f }
  variant { b - a }
= if b > a then sum_mult a (b-1) l f
```

The fact that we define the lemma as a recursive function, instead of stating it as a logical predicate, allows us to do two important things. First, we simulate the induction hypothesis via a recursive call, which is useful since the ATPs usually do not support reasoning by induction. Second, writing a lemma as a program function allows us to call it with convenient arguments later, which is equivalent to giving an explicit lemma instance.

Notice that the lemma is given an explicit variant clause. Indeed, when one writes a lemma function, Why3 verifies that it is effect-free and terminating. In the definition above, the termination is ensured by a measure `b - a` which decreases at each recursive call and will stay non-negative.

Now, a possible way to complete the second task would be to show the associativity directly for the multiplication implemented by the naive algorithm from task one. However, such a solution would be *ad hoc*: each time we implement the matrix multiplication differently, the associativity must be reproved.

To make our solution more general, we opt for a different solution which consists roughly of two steps. First, we provide an axiomatized theory of matrices where we prove that matrix product, as a mathematical operation, is associative. Second, we create a model function from program matrices to their logical representation in our theory. Finally, we show that from the model perspective naive multiplication implements the mathematical product. When all this is done, we have the associativity of naive multiplication for free.

We split our matrix axiomatization into two modules. The first module introduces a new abstract type and declares the following functions

```
type mat 'a
function rows (mat 'a) : int
function cols (mat 'a) : int
function get (mat 'a) int int : 'a
function set (mat 'a) int int 'a : mat 'a
function create (r c: int) (f: int → int → 'a) : mat 'a
```

and adds a set of axioms that describes their behavior. We add the `create` function to build new matrices by comprehension. For instance, the axiom

```
axiom create_get:
  ∀r c: int, f: int → int → 'a, i j: int.
    0 ≤ i < r → 0 ≤ j < c → get (create r c f) i j = f i j
```

describes the content of each cell of the matrix `(create r c f)` in terms of the initialization function `f`. Additionally, we have an extensionality axiom that expresses that matrices are defined by their content.

```
predicate (==) (m1 m2: mat 'a) =
  m1.rows = m2.rows && m1.cols = m2.cols &&
  ∀i j: int. 0 ≤ i < m1.rows → 0 ≤ j < m1.cols →
  get m1 i j = get m2 i j
axiom extensionality:
  ∀m1 m2: mat 'a. m1 == m2 → m1 = m2
```

It is worth pointing out that we can define most useful operations on matrices using the `create` function. For instance, we define the update function `set` as:

```
function set (m:mat 'a) (x y:int) (z:'a) : mat 'a =
  create m.rows m.cols
    (λx₀ y₀. if x₀ = x && y₀ = y then z else get m x₀ y₀)
```

That is, instead of adding axioms on the behavior of `set` function with respect to the `cols`, `rows`, and `get` functions, these elementary properties are stated as lemmas which are then proved automatically, thus diminishing the number of axioms on which our theory relies.

The second module defines arithmetic operations over integer matrices as straightforward instances of `create`, and exports various proved lemmas about their algebraic properties, including associativity and distributivity. Although we are looking for associativity, the other properties are expected in such a theory, and we will use some of them in later sections. A typical proof amounts to writing the following:

```
function mul_atom (a b: mat int) (i j: int) : int → int =
  λk. get a i k  * get b k j

function mul (a b: mat int) : mat int =
  create a.rows b.cols b (λi j. sum 0 a.cols (mul_atom a b i j))

let lemma mul_assoc_get (a b c: mat int) (i j: int) : unit
  requires { a.cols = b.rows ∧ b.cols = c.rows }
  requires { 0 ≤ i < a.rows ∧ 0 ≤ j < c.cols }
  ensures  { get (mul (mul a b) c) i j = get (mul a (mul b c)) i j }
= ...

lemma mul_assoc: ∀a b c. a.cols = b.rows → b.cols = c.rows →
      mul a (mul b c) =  mul (mul a b) c
   by mul a (mul b c) == mul (mul a b) c
```

The `by` [3] connective in the last line instruments the lemma with a logical cut for its proof, to show the desired instance of extensionality, which follows by the auxiliary lemma function `mul_assoc_get`.

To prove this auxiliary lemma, we first define a parameterized version of `sum` to account for functions depending on two integers:

```
function sumf (a b: int) (f: int → int → int) : int → int =
  λx. sum a b (f x)
```

More precisely, we apply `sumf` operator exactly twice, for each of the two functions `ft1` and `ft2` defined below:

```
function ft1 (a b c: mat int) (i j: int) : int → int → int =
  λk. smulf (get c k j) (mul_atom a b i k)
function ft2 (a b c: mat int) (i j: int) : int → int → int =
  λk. smulf (get a i k) (mul_atom b c k j)
```

With these definitions and `sum_mult` lemma function, we can deduce that functions `λk.mul_atom (mul a b) c i j k` and `λk.sumf 0 (cols a) (ft1 a b c i j k)` give the same result for any `k` such that `0 ≤ k < cols b`. Similarly, the functions `λk.mul_atom a (mul b c) i j k` and `λk.sumf 0 (cols b) (ft2 a b c i j k)` give the same result for any `k` such that `0 ≤ k < cols a`. However, what we need exactly is the equality between the sums that use these functions. To achieve this, we define a ghost function `sum_ext`

```
let ghost sum_ext (a b: int) (f g: int → int) : unit
  requires { ∀i. a ≤ i < b → f i = g i }
  ensures  { sum a b f  = sum a b g }
= ()
```

which we apply then twice in the proof of `mul_assoc_get`:

```
let ft1 = ft1 a b c i j in
let ft2 = ft2 a b c i j in
sum_ext 0 (cols b) (mul_atom (mul a b) c i j) (sumf 0 (cols a) ft1);
fubini ft1 ft2 0 (cols b) 0 (cols a);
sum_ext 0 (cols a) (mul_atom a (mul b c) i j) (sumf 0 (cols b) ft2);
```

As one can see, the proof of this auxiliary lemma follows the sketch given in Fig. 2. However, the folding/unfolding of definition of matrix product require some attention to get it right in an actual proof, in particular due to the use of higher order in `sum` operator.

Once we have formalized the matrix theory and proved associativity, it remains to connect it to the implementation with a model function:

```
function mdl (m: matrix 'a) : mat 'a =
  create m.rows m.columns (get m)
```

Then, we change the specification of `mult_naive` to use the model. This turns the postcondition into `result.mdl = mul a.mdl b.mdl`. The proof of this new specification is immediate and makes the second task trivial.

5 Strassen's Algorithm

The last part in the VerifyThis challenge was to verify Strassen's algorithm for square matrices with power-of-two size. We prove a more general implementation that performs dynamic padding to handle matrices of any size, including rectangular matrices.

The principle behind Strassen's Algorithm is to use 2×2 block multiplication recursively, using a scheme that performs seven sub-multiplications instead of eight. More precisely, it first partitions both input matrices A and B and output matrix M in 4 equal-sized matrices.

$$A = \left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right] \qquad B = \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right] \qquad M = \left[\begin{array}{c|c} M_{1,1} & M_{1,2} \\ \hline M_{2,1} & M_{2,2} \end{array}\right]$$

Then, it computes the four components of M using additions and subtractions of seven intermediate matrices

$$\begin{aligned} M_{1,1} &= X_1 + X_4 - X_5 + X_7 & \qquad M_{2,1} &= X_2 + X_4 \\ M_{1,2} &= X_3 + X_5 & \qquad M_{2,2} &= X_1 - X_2 + X_3 + X_6 \end{aligned}$$

where matrices X_i are computed using additions, subtractions, and only seven multiplications:

$$\begin{aligned} X_1 &= (A_{1,1} + A_{2,2})\ (B_{1,1} + B_{2,2}) & \qquad X_2 &= (A_{2,1} + A_{2,2})\ B_{1,1} \\ X_3 &= A_{1,1}\ (B_{1,2} - B_{2,2}) & \qquad X_4 &= A_{2,2}\ (B_{2,1} - B_{1,1}) \\ X_5 &= (A_{1,1} + A_{1,2})\ B_{2,2} & \qquad X_6 &= (A_{2,1} - A_{1,1})\ (B_{1,1} + B_{1,2}) \\ X_7 &= (A_{1,2} - A_{2,2})\ (B_{2,1} + B_{2,2}) \end{aligned}$$

When matrices have odd sizes, this recursive scheme cannot be applied. This is typically solved by peeling or zero-padding, either statically or dynamically to recover an even size.

5.1 Implementation

Our code of Strassen's algorithm is given in Fig. 3. We use a dynamic padding solution to cope with matrices with at least one odd dimension. That is, we add a zero row and/or a zero column to recover even dimensions, process the augmented matrices by a recursive call, and then we extract the relevant sub-matrix from the result.

When the size gets below an arbitrary cutoff, we use a naive matrix multiplication. As we could take any positive cutoff, we prove the algorithm independently of the concrete value.

The code in Fig. 3 uses the following simple matrix routines:

- matrix addition (`add`) and subtraction (`sub`).
- matrix block-to-block copy (`blit`).
- sub-matrix extraction (`block`).
- zero padding to adapt to a larger size (`padding`).

We verify implementations of those operations as part of our Why3 development. We do not detail their proof as it is completely standard.

5.2 Specification and Proof

We give the same specification for Strassen's algorithm as for naive multiplication. As for the proof, let us first focus on the correctness of Strassen's recursive scheme. We break down that proof in two parts. First, we prove that the usual 2×2 block decomposition of matrix product is correct. Then, we prove that the efficient computation scheme that uses seven multiplication indeed computes that block decomposition. That second part boils down to checking four matrix identities, which we will cover in details in Section 6.

In order to prove block decomposition, we introduce a dedicated module where sub-matrix extraction is defined by comprehension. It extracts a rectangle from a matrix, given the low corner at coordinates `r, c` and with dimensions `dr, dc`:

```
function block (a: mat int) (r dr c dc: int) : mat int =
  create dr dc (λ i j. get a (r+i) (c+j))
```

The module essentially proves two lemmas about relations between sub-matrix extraction and product, which are illustrated as in Fig. 4. One expresses sub-matrices of the product as products of sub-matrices, while the other decomposes products into sums of sub-matrices products. We then expect to obtain the desired block decomposition by two successive partitioning steps, but there is a catch. Our implementation extracts directly the 4 sub-matrices in one step instead of two. We bridge the gap by reducing successive sub-matrix extraction to single ones. In practice, we do this by proving the following lemma:

```
constant cut_off : int
axiom cut_off_positive : cut_off > 0
let rec strassen (a b: matrix int) : matrix int
= let (rw, md, cl) = (a.rows, a.columns, b.columns) in
  if rw ≤ cut_off || md ≤ cut_off || cl ≤ cut_off
  then mul_naive a b else
  let (qr, rr) = (div rw 2, mod rw 2) in
  let (qm, rm) = (div md 2, mod md 2) in
  let (qc, rc) = (div cl 2, mod cl 2) in
  if rr ≠ 0 || rm ≠ 0 || rc ≠ 0 then begin (* Padding *)
    let (rw', md', cl') = (rw + rr, md + rm, cl + rc) in
    let ap = padding a rw' md' in
    let bp = padding b md' cl' in
    let m  = strassen ap bp in
    block m 0 rw 0 cl
  end else begin (* regular Strassen multiplication *)
    let (m11, m12, m21, m22) =
      let a11 = block e a 0 qr 0 qm in
      let a12 = block e a 0 qr qm qm in
      let a21 = block e a qr qr 0 qm in
      let a22 = block e a qr qr qm qm in
      let b11 = block e b 0 qm 0 qc in
      let b12 = block e b 0 qm qc qc in
      let b21 = block e b qm qm 0 qc in
      let b22 = block e b qm qm qc qc in
      let x1  = strassen (add e a11 a22) (add e b11 b22) in
      let x2  = strassen (add e a21 a22) b11 in
      let x3  = strassen a11 (sub e b12 b22) in
      let x4  = strassen a22 (sub e b21 b11) in
      let x5  = strassen (add e a11 a12) b22 in
      let x6  = strassen (sub e a21 a11) (add e b11 b12) in
      let x7  = strassen (sub e a12 a22) (add e b21 b22) in
      let m11 = add e (sub e (add e x1 x4) x5) x7 in
      let m12 = add e x3 x5 in
      let m21 = add e x2 x4 in
      let m22 = add e (add e (sub e x1 x2) x3) x6 in
      (m11, m12, m21, m22) in
    let res = make a.rows b.columns 0 in
    blit m11 res 0 0 qr 0 0 qc;
    blit m12 res 0 0 qr 0 qc qc;
    blit m21 res 0 qr qr 0 0 qc;
    blit m22 res 0 qr qr 0 qc qc;
    res
  end
```

Fig. 3 Strassen's algorithm implementation.

Fig. 4 Relations between sub-matrices and product.

```
let lemma double_block
  (a: matrix int) (r1 dr1 c1 dc1 r2 dr2 c2 dc2: int) : unit
  requires { 0 ≤ r1 ≤ r1 + dr1 ≤ a.mdl.rows }
  requires { 0 ≤ c1 ≤ c1 + dc1 ≤ a.mdl.cols }
  requires { 0 ≤ r2 ≤ r2 + dr2 ≤ dr1 }
  requires { 0 ≤ c2 ≤ c2 + dc2 ≤ dc1 }
  ensures  { block (block a.mdl r1 dr1 c1 dc1) r2 dr2 c2 dc2 =
    block a.mdl (r1+r2) dr2 (c1+c2) dc2 }
= assert   { block (block a.mdl r1 dr1 c1 dc1) r2 dr2 c2 dc2 ==
    block a.mdl (r1+r2) dr2 (c1+c2) dc2 }
```

This is sufficient to prove the algebraic relations between the partition of the product and the partitions of the input matrices. Also, note that we can readily reuse the same proof scheme for padding correctness. Indeed, it amounts to checking that the block we extract from the product of padded matrices is the right one. This follows immediately from the relevant block decomposition of the matrix product.

Finally, there is only one non-trivial remaining part: termination. It is non-trivial because our padding scheme increases the matrices dimensions. This does not cause any problem, because the next step will halve it. We prove termination by introducing an extra ghost argument `flag` that identifies which matrices do not require padding:

```
let rec strassen (a b: matrix int) (ghost flag: int) : matrix int
  requires { 0 ≤ flag }
  requires { flag = 0 →
    a.mdl.cols = 1 ∨ a.mdl.cols = 1 ∨ b.mdl.cols = 1 ∨
    (even a.mdl.rows ∧ even a.mdl.cols ∧ even b.mdl.cols) }
  variant  { (* lexicographic order *)
    a.mdl.rows + a.mdl.cols + b.mdl.cols + 3 * flag, flag }
```

We distinguish two situations in which padding is not required: either one of the matrices is a vector, in which case our implementation calls the naive multiplication algorithm; or the matrices dimensions are all even. The `variant` clause introduces a lexicographic order to prove termination of recursive calls. This should be understood as follows: in the case padding is applied it is the `flag` value that will serve as decreasing measure. We add `3 * flag` to account for the possibility of augmenting all the matrices dimensions by one, keeping the value of the first measure unchanged between the recursive calls. In the regular case (Strassen's recursive scheme can be applied), matrices dimensions are halved, which leaves enough room to account for padding being applied again.

6 Proving Validity of Matrix Identities by Reflection

Once we get rid of block matrix multiplication, proving validity of matrix identities turns out to be the major difficulty. Indeed, Strassen's algorithm relies on identities such as

$$A_{1,1}B_{1,2} + A_{1,2}B_{2,2} = A_{1,1}(B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2})B_{2,2}$$

which are obvious for humans, but turn out to be quite a trouble for ATPs.

A possible explanation may be that ATPs do not identify algebraic properties of matrices as such and handle them as any other quantified axioms. This suggests that they create a considerable number of useless instances of these axioms before finding the relevant ones. Moreover, those axioms have dimension constraints that must be derived for each instance. This makes the situation even worse, as for example the dimensions for the matrix AB must be derived from those of A and B by using yet another lemma.

One possible solution to bridge this gap is "assertion forcing", *i.e.* adding assertions about intermediate identities until the sub-problems are small enough to be handled by ATPs. However, after trying to go this way, we found that even the identity above (the easiest one) requires an unreasonable number of explicit steps.

Without support of automated provers, making use of an interactive one (typically Coq) would be a standard choice. If the interactive prover has support for proving ring-like identities, then it would suffice to embed our matrix theory inside the prover's ring framework. However, we were curious to see if we could embed some kind of similar ring support inside Why3 itself. That leads us to the technique known as proof by reflection [2]. The methodology we follow is actually very similar to the one presented by Bertot and Castéran [1, chapter 16].

6.1 Proof by Reflection Recipe

The proof by reflection approach essentially amounts to running a decision procedure within the logic of the proof environment. Reader can find some illustrative examples in Coq Reference Manual [11, Chapter 25]. In our case, it is more specifically a canonicalization procedure for matrix expressions. We use it to prove identities simply by comparing canonical forms. As it happens, we cannot define such procedures directly on the algebraic objects as we cannot analyze the structure of the matrix expressions. So the first step is to replace the expressions by abstract syntax trees, which are more convenient for computation.

More precisely, we create a symbolic model for expressions, taking into account the expression structure. This symbolic model comes with a companion interpretation $[\![\cdot]\!]$ translating them to algebraic objects. We also need some kind of reflection mechanism to turn the expressions e into symbolic objects $\hat{e}$ such that $[\![\hat{e}]\!] = e$. Figure 5 summarizes this process schematically.

6.2 Computation Transformation

Running the canonicalization procedure requires computation support within the logic of the proof environment. In Why3, we get it from the `compute` transforma-

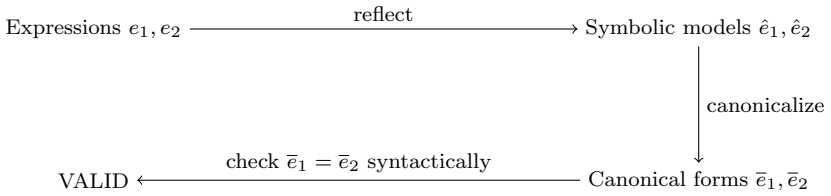

Fig. 5 Proof by reflection.

tion. It is a logical transformation that we manually apply to selected verification conditions before sending them to ATPs.

The computation transformation performs rewriting on formulas up to normal form, based on a user-supplied rewriting system. We select which identities are used as rewrite rules by tagging axioms with `"rewrite"` labels. We can also select definitions as rewrite rules by marking the symbols with `meta` declarations. For example, the following declarations add axiom `fact_rule` and `f`'s definition as rewrite rule.

```
axiom fact_rule :
  "rewrite" ∀n. fact n = if n ≤ 0 then 1 else n * fact (n-1)
function f (x: 'a) : 'a = x
meta rewrite_def function f
```

We typically use labeling in conjunction with an introduction transformation to locally add rewrite rules. This is extremely useful in postconditions as it creates logical objects whose structure can be exploited by computation:

```
ensures { "rewrite" result = fact arg }
```

By default, the set of rewrite rules also includes computation of built-in symbols, like integer arithmetic, Boolean operators and pattern-matching on algebraic datatypes. This means that based on the rules above the following goal is reduced by `compute` to `true`:

```
goal G: fact 4 = f 24 ∧ fact (fact 3) = 720 ∧ ∀x:'a. f (f x) = x
```

6.3 Representation of Expressions and Normalization Procedure

We choose to represent matrix expressions by a cascade of symbolic operators $\hat{+}, \hat{\times}, \ldots$ maintaining expressions in canonical form. In other words, we represent an expression $(a + b) \times c$ as $(\hat{a}\hat{+}\hat{b})\hat{\times}\hat{c}$. Moreover, the canonicalization procedure boils down to the computation of symbolic operators.

In order to represent canonical forms for expressions, we use a list of monomials. Monomials themselves consist of products of variables with explicit signs. These products are represented by lists of variables. We consider that an expression is in canonical form if the monomials are sorted and there are no opposite monomials in the expression. To ease cancellation, we choose the monomial order so that opposite monomials are consecutive.

```
type  var = int
type mono = { m_prod: list var; m_pos: bool; }
type expr = { e_body: list mono; e_rows: int; e_cols: int; }
```

As the canonicalization procedure efficiency is not our prime concern here, writing the symbolic operators is a simple programming exercise. Nevertheless, we need to prove them correct. More precisely, we need commutation theorems to replace regular operations by symbolic ones. Those theorems typically have the shape $[\![\hat{e}_1 \hat{+} \hat{e}_2]\!] = [\![\hat{e}_1]\!] + [\![\hat{e}_2]\!]$ for an interpretation function $[\![\cdot]\!]$. Note that we do not need to prove that our operators maintain the canonical form as well. Indeed, failing to meet this criterion does not endanger the procedure's soundness. Hence the only invariants we maintain on symbolic expressions are the mandatory dimension constraints between variables.

We define the interpretation function on expressions as follows:

```
function lm_mdl_simp
  (f: int → mat int) (r c: int) (l: list mono) : mat int =
  match l with
  | Nil → zero r c
  | Cons x Nil → m_mdl f x
  | Cons x q → add (lm_mdl_simp f r c q) (m_mdl f x)
  end
meta rewrite_def function lm_mdl_simp
function e_mdl (f: int → mat int) (e: expr) : mat int =
  lm_mdl_simp f e.e_rows e.e_cols e.e_body
meta rewrite_def function e_mdl
```

The parameter `f` corresponds to the environment, mapping variables to matrices, while the functions `m_mdl` and `l_vld` correspond to similarly-defined symbols for monomials. Note that `lm_mdl_simp` does not produce the zero whenever possible, which may seem a useless simplification. We carry it out because the interpretation function is also used to turn back symbolic expressions to regular ones, which we want to be as small as possible. However, this does not change the actual interpretation, so we use a more practical non-simplifying variant `lm_mdl` for proofs.

We define dimension constraints for symbolic expressions in a similar way, *i.e.* we enforce that variable dimensions match inside products. We also have to enforce that products are non-empty, as matrix unit does not exist except for square matrices. The following predicate defines the constraints for monomials, and is readily extended to full expressions:

```
predicate l_vld (f: int → mat int) (r c: int) (l: list int) =
  match l with
  | Nil → false
  | Cons x Nil → (f c).rows = r ∧ (f x).cols = c
  | Cons x q → (f x).rows = r ∧ l_vld f (f x).cols c q
  end
```

Finally, we prove the commutation theorems by embedding them within ghost procedures whose structure is identical to the corresponding operator implementation. The contracts of those procedures correspond exactly to the commutation

theorem. We can then recover the theorems by straightforward lemma functions, though ghost procedures suffice for our purposes as we explain later.

For example, symbolic multiplication is implemented by fully distributing all monomials on the left side over the right side. Addition is carried out by function `lm_merge`, a variation of sorted list merging. This fragment is implemented and proved as follows:

```
function lm_distribute (l1 l2: list mono) : list mono =
  match l1 with
  | Nil → Nil
  | Cons x q → lm_merge Nil (m_distribute x l2) (lm_distribute q l2)
  end
meta rewrite_def function lm_distribute
let rec ghost lm_distribute_ok (f: int → mat int) (r k c: int)
                               (l1 l2: list mono) : list mono
  requires { r ≥ 0 ∧ k ≥ 0 ∧ c ≥ 0 }
  requires { lm_vld f r k l1 ∧ lm_vld f k c l2 }
  ensures  { result = lm_distribute l1 l2 ∧ lm_vld f r c result }
  ensures  { lm_mdl f r c result =
               mul (lm_mdl f r k l1) (lm_mdl f k c l2) }
  variant  { l1 }
= match l1 with
  | Nil → Nil
  | Cons x q →
    lm_merge_ok f r c Nil (m_distribute_ok f r k c x l2)
                          (lm_distribute_ok f r k c q l2)
end
```

The correctness of the symbolic operators is a straightforward consequence of the algebraic properties provided by our matrix theory (distributivity for the one above).

6.4 Reflecting Expressions through Ghost Tagging

In order to use our canonicalization procedure, we need to reflect matrix expressions as symbolic expressions. Unfortunately, there are currently no evident way to carry out such replacement in a fully automatic manner in Why3. But note that we already built the result of all the involved expressions either as part of the implementation, or as part of the block decomposition proof. We can then build their symbolic counterparts automatically at the same time. In practice, we achieve that by tagging each matrix with a correlated ghost symbolic expression. As those tags are ghost, they do not incur any extra runtime cost.

```
type with_symb = { phy : matrix int;
             ghost sym : expr; (* reflection *) }
predicate with_symb_vld (env:env) (ws:with_symb) =
  e_vld env.ev_f ws.sym ∧              (* dimension constraints *)
  e_mdl env.ev_f ws.sym = ws.phy.mdl ∧ (* same model *)
  ws.sym.e_rows = ws.phy.mdl.rows ∧    (* same dimensions *)
  ws.sym.e_cols = ws.sym.mdl.cols
```

We then perform paired construction using straightforward tagging combinators, each arithmetic operation being naturally paired with its symbolic counterpart. Their proof follows by calling the ghost procedures corresponding to the commutation theorems.

The only subtleties arise in the case of variables. As we should have precisely one variable per block, we pair their introduction with the block extraction operators. However, this must change the environment. In order to account for such changes, we use a mutable environment, containing the variable map and a symbol generator. This also enforces that we always generate a fresh variable.

```
type env = { mutable ev_f: int → mat int; mutable ev_c: int; }
function extends
  (f: int → mat int) (c: int) (v: mat int) : int → mat int =
    λn. if n ≠ c then f n else v
meta rewrite_def function extends
let block_ws (ghost env:env) (a:matrix int) (r dr c dc: int) : with_symb
  requires { 0 ≤ r ≤ r + dr ≤ a.mdl.F.rows }
  requires { 0 ≤ c ≤ c + dc ≤ a.mdl.F.cols }
  ensures { let rm = result.phy.mdl in
    rm = block a.mdl r dr c dc ∧ rm.rows = dr ∧ rm.cols = dc }
  ensures { "rewrite"
            result.sym = symb_mat result.phy.mdl (old env.ev_c) }
  ensures { "rewrite"
            env.ev_f = old (extends env.ev_f env.ev_c result.phy.mdl) }
  ensures { "rewrite" env.ev_c = old env.ev_c + 1 }
  ensures { with_symb_vld env result }
```

We use `"rewrite"` labels to register the field definitions as rewrite rules, so that they are correctly unfolded during computation. Note that although mutability makes variable generation easy, it has the unfortunate pitfall of breaking the representation invariants `with_symb_vld`. We recover them when necessary using the computation transformation to break the invariant as a conjunction of immediate dimension constraints.

In practice, to prove `e1 = e2` we proceed as follow:

- build `e1` and `e2` using the tagging combinators. In our case this boils down to replacing arithmetic operators as we already build these expressions for other purposes.
- write the assertion `e_mdl env.ev_f e1.sym = e_mdl env.ev_f e2.sym`.
- use the `compute` transformation on the associated proof obligation. If the matrix identity is correct this reduces the above assertion to `true`.

Note that the above methodology implies that we build the matrices $A_{i,1}B_{1,j} + A_{i,2}B_{2,j}$ using our tagging combinators as well, which would lead to a total of 15 sub-multiplication instead of 7. However, those extra operations are only used for specification purposes, so we perform them in ghost code to avoid the execution penalty.

7 Proof Statistics

Our Why3 development is divided into five different `.mlw` files:

file	lines of specification	lines of code
matrices.mlw	236	0
matrices_ring_simp.mlw	396	0
naive.mlw	20	12
strassen.mlw	188	179
sum_extended.mlw	38	0
total	878	191

Table 1 Experimental results.

prover	# VCs proved	average time (sec)	# VCs proved only by this prover
CVC4 (1.4)	413	0.45	0
Alt-Ergo (1.30)	465	0.28	45
Z3 (4.4.1)	399	0.86	3

Table 2 Experimental results solvers.

- `sum_extended.mlw`, where we define and prove some extra properties of summation operator with respect to the Why3 standard library;
- `matrices.mlw`, containing our axiomatic theory of matrices;
- `matrices_ring_simp.mlw`, where we define the proof by reflection machinery;
- `naive.mlw`, in which we prove the naive multiplication algorithm;
- `strassen.mlw`, in which we prove Strassen's algorithm and its matrix subroutines (such as matrix addition).

Table 1 shows the distribution of specification among these five files. We observe that we needed a lot of specification to actually prove Strassen's algorithm as the specification ratio is about 4.6 (specification/code). However, most of the specification corresponds to a proof library for matrix algebra. Taking this out, we see that the part dedicated to Strassen's algorithm is reasonable, with a specification ratio of about 1.

After using the `compute` transformation, all the generated verification conditions (VCs) are automatically discharged using a combination of Alt-Ergo, CVC4, and Z3 SMT solvers. The proof by reflection setup generated a total of 11 goals requiring computation. Table 2 summarizes the contribution of each prover to our development.

8 Related Work

There are other works in the literature that tackle the proof of matrix multiplication algorithm similar to Strassen's. The closest to our work is that of Dénès *et al.* [5]. They propose a refinement-based mechanism to specify and prove efficient algebraic algorithms in the Coq proof assistant. The authors report on the use of the Coq's `ring` tactic to ease the proof of Winograd's algorithm (a variant of Strassen's with fewer additions and subtractions), a similar approach to our proof by reflection. To cope with the case of odd-sized matrices they implement dynamic peeling to remove extra rows or columns. However the use of Coq ring tactic limits this work to square matrices while our works tackle any rectangle ones.

Another work close to ours is a proof of Strassen's Algorithm in the Archives in Formal Proofs [12]. Their implementation tackles rectangular matrices of any

work	proof size for Strassen's algorithm
Dénès and al.	210
Thiemann and al.	250
Our work	200

Table 3 Proof size comparison

dimensions as well, using dynamic peeling. They prove matrix identities using Isabelle's built-in simplification mechanism directly on the matrix terms.

Strassen algorithm is proved as well in the ACL2 system [9]. The use of ACL2 with suitable rewriting rules and proper ring structure allows a high degree of automation in the proof process. However, they use an *ad hoc* definition of matrices whose sizes can only be powers of 2.

Srivastava *et al.* propose a technique for the synthesis of imperative programs [10] where synthesis is regarded as a verification problem. Verification tools are then used with a two-folded purpose: to synthesize programs and their correctness proof. One case study presented for this technique is multiplication for 2×2 integer matrices, for which the authors have been able to synthesize the cascade of arithmetic operations corresponding to Strassen's recursive scheme.

We compare the proof size for several developments in table 3. We only take into account the part devoted to Strassen's algorithm alone, as matrix primitives and properties are typically scattered among much larger libraries. We observe that the size of the core proof does not vary much.

9 Conclusion

We presented our solution for the first challenge of the VerifyThis 2016 competition. While presenting our solutions in detail, we took the opportunity to illustrate some interesting features of Why3, among which are higher-order functions in logic, lemma functions, ghost code, and proof obligation transformations. It would be interesting to see whether the proof by reflection methodology we use in this work can be helpful for verification of some other case studies, especially in a context which favors ATPs.

When these cases become larger, we would be inevitably confronted with performance issues related to the internal decision procedure. Since the soundness of such a procedure is necessary to use it within the reflection framework, we would be as well confronted to show the soundness of more efficient decision procedures.

Another direction would be to replace ghost tagging with even more automatic reflection procedures. To this end, we plan to investigate the possibility of using a dedicated module Why3 transformation to carry out the reflection phase.

Acknowledgements This work is partly supported by the Bware (ANR-12-INSE-0010, http://bware.lri.fr/), VOCAL (ANR-15-CE25-008, https://vocal.lri.fr/) projects of the French national research organization (ANR), and by the Portuguese Foundation for the Sciences and Technology (grant FCT-SFRH/BD/99432/2014). We thank Arthur Charguéraud, Jean-Christophe Filliâtre, and Claude Marché for their comments and remarks.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer-Verlag (2004)
2. Boutin, S.: Réflexions sur les quotients. thèse d'université, Paris 7 (April 1997)
3. Clochard, M.: Preuves taillées en biseau. In: Vingt-huitièmes Journées Francophones des Langages Applicatifs. Gourette, France (Jan 2017)
4. Clochard, M., Filliâtre, J.C., Marché, C., Paskevich, A.: Formalizing semantics with an automatic program verifier. In: Giannakopoulou, D., Kroening, D. (eds.) 6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE). Lecture Notes in Computer Science, vol. 8471, pp. 37–51. Springer, Vienna, Austria (Jul 2014)
5. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in Coq. In: Beringer, L., Felty, A. (eds.) ITP - 3rd International Conference on Interactive Theorem Proving - 2012. Lecture Notes In Computer Science, vol. 7406, pp. 83–98. Springer, Springer, Princeton, United States (2012), http://hal.inria.fr/hal-00734505
6. Filliâtre, J.C.: One logic to use them all. In: 24th International Conference on Automated Deduction (CADE-24). Lecture Notes in Artificial Intelligence, vol. 7898, pp. 1–20. Springer, Lake Placid, USA (June 2013)
7. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In: Biere, A., Bloem, R. (eds.) 26th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 8859, pp. 1–16. Springer, Vienna, Austria (Jul 2014)
8. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
9. Palomo-Lozano, F., Medina-Bulo, I., Alonso-Jiménez, J.: Certification of matrix multiplication algorithms. Strassen's algorithm in ACL2,. In: Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics. pp. 283–298. Edinburgh (Scotland) (2001)
10. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 313–326. POPL '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1706299.1706337
11. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.6 (2016), http://coq.inria.fr, http://coq.inria.fr
12. Thiemann, R., Yamada, A.: Matrices, jordan normal forms, and spectral radius theory. Archive of Formal Proofs 2015 (2015), https://www.isa-afp.org/entries/Jordan_Normal_Form.shtml