The Matrix Reproved (Verification Pearl)

Martin Clochard, Léon Gondelman, Mário Pereira

The Matrix Reproved* (Verification Pearl)

Martin Clochard[1,2], Léon Gondelman[1,2], and Mário Pereira[1,2]

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

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 the 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 and apply the proof by reflection methodology.

1 Introduction

In this paper we describe a complete solution for the first challenge of the VerifyThis 2016 competition 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. 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 square matrices of arbitrary size in a program verification environment based on automated theorem provers.

Wishing to make our solutions both concise and generic, we devised in Why3 an axiomatic theory for matrices and showed 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[3].

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

* This work is partly supported by the Bware (ANR-12-INSE-0010, http://bware.lri.fr/) and 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).

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

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

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 [6], 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 [5], 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 *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 [4], as well as a limited form of higher-order logic [2]. 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 in non-standard order. In this section we present our solution to this part of the challenge.

We first write the WhyML equivalent of the challenge code for multiplication of two matrices `a` and `b`:

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

To encode matrices, we use two-dimensional arrays as provided in the Why3 standard library. Operations `get` and `set` have the usual semantics, and `make` carries out creation and initialization. Such arrays are represented by the following abstract type, whose fields can only be accessed in specifications.

```
type matrix 'a
  model { rows: int; columns: int; mutable elts: map int (map int '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 `matrix_product` predicate mimicks the mathematical definition of matrix product $(AB)_{ij} = \sum_{k=0}^{m-1} A_{ik}B_{kj}$:

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

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

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

The last argument we give to `sum` is a first-class function $\lambda k.M$. Why3 supports such definitions by translating them to first-order values [2].

To prove that `mult_naive` meets its specification, we give suitable loop invariants. There are two kinds of invariant per loop. The first one 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. Let us illustrate this with the inner loop. In that case, the loop has the effect of writing a partial sum into cells `0` to `j-1` of the `i`-th row, leaving other cells unchanged.

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

With the given specification 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 computations $(AB)C$ and $A(BC)$, and then prove the corresponding results are always the same. In our case, this corresponds to prove the following:

```
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 ∧
    forall i j. 0 ≤ i < ab_c.rows ∧ 0 ≤ j < ab_c.columns →
      ab_c.elts[i][j] = a_bc.elts[i][j] }
```

As one can guess, the proof of associativity relies essentially on the linearity properties of the sum operator and Fubini's principle. Let us illustrate how we establish the additivity of the sum (the homogeneity of the sum and Fubini's principle are done in a similar way). First we define a higher-order function `addf` which, given two integer functions `f` and `g`, returns a function `\x. f x + g x`. Then, we state the additivity as a lemma function:

```
let rec lemma additivity (a b: int) (f g: int → int) : unit
  ensures { sum a b (addf f g) = sum a b f + sum a b g }
  variant { b - a }
= if b > a then additivity a (b-1) f g
```

The fact that we write the lemma not as a logical statement but as a recursive function 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 amounts to giving instances. 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.

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 in 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. Additionally, we have an extensionality axiom, *i.e.* that for any pair of matrices `m1, m2`, we have `m1 == m2 → m1 = m2` where `m1 == m2` means that both matrices have the same dimensions and the content of their cells are the same.

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 =
  \k. get a i k  * get b k j
function mul (a b: mat int) : mat int =
  create (rows a) (cols b) (\i j. sum 0 (cols a) (mul_atom a b i j))
let lemma mul_assoc_get (a b c: mat int) (i j: int)
  requires { cols a = rows b ∧ cols b = rows c }
  requires { 0 ⩽ i < rows a ∧ 0 ⩽ j < cols c }
  ensures  { get (mul (mul a b) c) i j = get (mul a (mul b c)) i j }
= ...
lemma mul_assoc: forall a b c. cols a = rows b → cols b = rows c →
      mul a (mul b c) =  mul (mul a b) c
   by mul a (mul b c) == mul (mul a b) c
```

The `by` connective in the last line instruments the lemma with a logical cut for its proof, to show the desired instance of extensionality. It follows by the auxiliary lemma function `mul_assoc_get`, whose proof is omitted here.

Once we formalized the matrix theory and proved associativity, it remains to connect it to the implementation by 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 to `mdl result = mul (mdl a) (mdl b)`. The proof of this new specification is immediate and makes the second task trivial.

5 Strassen's Algorithm in Why3

The last (and optional) part in the VerifyThis challenge was to verify Strassen's algorithm for power-of-two square matrices. We prove a more general implementation that uses a padding scheme to handle square matrices of any size.

5.1 Implementation

The principle behind Strassen's Algorithm is to use 2×2 block multiplication recursively, using a scheme that uses 7 sub-multiplications instead of 8. More precisely, it works in 3 phases. It first partitions both input matrices in 4 equal-sized square matrices. Then, it computes 7 products of matrices obtained by additions and subtractions. Finally, it obtains a similar partition of the result using addition and subtractions from those products. The details can be found in appendix B.

For even sizes, our implementation closely follows Strassen's recursive scheme. To this end, we first implement and prove a few simple matrix routines:

- Matrix addition (`add`) and subtraction (`sub`);
- Matrix block-to-block copy (`blit`);
- Sub-matrix extraction (`block`).

For odd sizes, the recursive scheme cannot be applied. This is typically solved by peeling or zero-padding, either statically or dynamically to recover an even size. We use a dynamic padding solution. In case the matrices have odd size, we add a zero row and column to recover an even size, make a recursive call and extract the relevant sub-matrices.

When the size gets below an arbitrary cutoff we use naive matrix multiplication. Although we used a concrete value, the code is correct for any positive cutoff. We ensure this by wrapping the cutoff value under an abstract block, which hides its precise value in verification conditions.

5.2 Specification and Proof

Except for the additional requirements that the matrices are square, 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 ring equations, 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`:

Fig. 1. Relations between sub-matrices and product

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

The module essentially proves two lemmas about relations between sub-matrix extraction and product, which are best seen graphically as in Figure 1. 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 input sub-matrices, while using those lemmas implies extracting from intermediate sub-matrices. We bridge the gap by reducing successive sub-matrix extraction to single ones. In practice, we do this by proving and then calling a lemma function with the following postcondition:

```
ensures { 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 matrix size. This does not cause any problem, because the next step will halve it. We prove termination by introducing an extra ghost argument identifying matrices that will not require padding:

```
requires { 0 ⩽ flag }
requires { flag = 0 → a.mdl.cols = 1  ∨  exists k. a.mdl.cols = 2 * k }
variant  { a.mdl.cols + flag, flag } (* lexicographic order *)
```

All generated VCs for the described specification are automatically discharged using a combination of Alt-Ergo, CVC4, and Z3 SMT solvers.

6 Proving Validity of Ring Equations by Reflection

Once we got rid of block matrix multiplication, proving validity of algebraic equations was the major difficulty. Indeed, Strassen's algorithm relies on equations

like

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

which is obvious for humans, but turns out to be quite a trouble for ATPs.

A possible explanation is that ATPs are not directly aware that fixed-size square matrices form a ring, struggling to instantiate relevant lemmas correctly. Moreover, the dimension constraints from those lemmas must be proved at each application, which makes the situation even worse.

One possible solution would be to add assertions about intermediate equations inside the code until they are easy enough to be exploitable by ATPs to bridge the gap. However, after trying to go this way, we found that even for the equality above (the easiest one), the gap was too large for ATPs which were still spending too much time to discharge the proof obligations.

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 equations, then it would suffice to embed our matrix theory inside the prover's ring structure. 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. The methodology we follow is actually very similar to the one presented in [1, chapter 16].

To carry out proof by reflection of algebraic equations, we have to do two things. First, we have to *reflect* (translate) the original expressions on each side by equivalent syntactical forms. Second, we need a procedure that normalizes a syntactical form so that the comparison of algebraic expressions becomes trivial. This can be implemented in Why3 logic, and run using the `compute_specified` transformation. This transformation normalizes a given goal, making boolean and integer arithmetic simplifications, and applying user-defined rewrite rules (in the source code one can add declarations to configure the transformation). To complete the proof, we need a lemma saying that the normalization procedure preserves the interpretation of the syntactic form. Let us now describe how we carry out these two phases in more detail.

6.1 Reflecting Algebraic Expressions by Ghost Tagging

Essentially, the reflection phase amounts to building an equivalent syntactic form for each algebraic expression. In our case, we achieve that by tagging each matrix with a ghost symbolic expression:

```
type with_symb = { phy : matrix int;
                   ghost sym : expr; (* reflection *) }

predicate with_symb_vld (env:env) (ws:with_symb) =
  e_vld env ws.sym ∧ (* internal dimension conditions *)
  e_mdl env ws.sym = ws.phy.mdl ∧ (* Model correlation *)
  ws.sym.e_rows = ws.phy.mdl.rows ∧ (* Dimensions correlation *)
  ws.sym.e_cols = ws.phy.mdl.cols
```

Notice that the representation predicate above is parametrized by an environment, which binds expression variables to matrices. Also, as field `sym` is ghost, it will not incur any extra runtime cost.

It remains then to provide, for each arithmetic operation, a tagging combinator that wraps in parallel the corresponding matrix computations and symbolic executions on their reflection. For instance, the combinator for addition is defined by:

```
let add_ws (ghost env:env) (a b:with_symb) : with_symb
  requires { a.phy.mdl.rows = b.phy.mdl.rows }
  requires { a.phy.mdl.cols = b.phy.mdl.cols }
  requires { with_symb_vld env a ∧ with_symb_vld env b }
  ensures { result.phy.mdl = add a.phy.mdl b.phy.mdl }
  ensures { result.sym = symb_add a.sym b.sym }
  ensures { with_symb_vld env result }
= { phy = add a.phy b.phy;
    sym = ghost symb_add env a.sym b.sym }
```

Introduction of variables is carried out by a similar combinator on top of submatrix extraction.

6.2 Normalizing Algebraic Expressions

In practice, we choose not to reflect completely algebraic expressions as syntactic objects. Instead, we implement in Why3 logic smart constructors that maintain normal forms, and reflect algebraic expressions as a computation tree made of those constructors. This has the advantage that we can use the ghost tagging mechanism to instantiate correctness lemmas as well. Also, this reduces the proof work to first write an assertion like

```
assert { e_mdl e m11.sym = e_mdl e egm11 }
```

then to apply the transformation on the associated goal. This completely reduces both computation trees and interprets back the results as standard expressions. Since they are in normal form, the equation becomes trivial and is reduced to `true` by the transformation directly.

The normal form we choose for algebraic expressions is a sorted sequence of signed variable products (monomials), interpreted as the sum of those monomials. We represent them using Why3 algebraic datatype of lists, and integers for variables. To correlate variables with effective matrices, we use a simple environment composed of a mapping and a symbol generator.

```
type mono = { m_prod : list int; m_pos : bool }
type expr = { e_body : list mono; e_rows : int; e_cols : int }
type env = { mutable ev_f : int → mat int; mutable ev_c : int }
```

The smart constructor implementations are fairly straightforward. For instance, addition is done by merging sorted lists followed by collapsing opposite terms. Multiplication is reduced to addition by distributing. We carried out smart constructors correctness proof by writing ghost procedures that mimic

the control structure of the logical functions. Those procedures are then called in the ghost tagging combinators.

Note that we only prove that the simplifications are correct, not that we indeed compute normal forms. Although it may be desirable, it is not necessary if our goal is to prove algebraic equations. We only need both sides to be reduced to the same term. This also makes the aforementioned correctness proofs very easy, as the simplifications we carry out mirror the lemmas of our matrix theory.

All generated proof obligations for the correctness of smart constructors are automatically discharged using a combination of Alt-Ergo and CVC4 SMT solvers. The algebraic equations involved in Strassen's algorithm are directly eliminated by `compute_specified`.

7 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.* [3]. 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 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 implemented dynamic peeling to remove extra rows or columns.

Another work is the proof of Strassen's algorithm in the ACL2 system [7]. 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 [8] 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 Strassen's algorithm for 2×2 integer matrices, for which the authors have been able to synthesize the additions and subtractions operations over block matrices.

8 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 favours ATPs.

Acknowledgements 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. 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)
3. 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, États-Unis (2012), http://hal.inria.fr/hal-00734505
4. 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)
5. 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)
6. 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)
7. 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)
8. 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

A Challenge 1 original text

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.

B Strassen recursion scheme

Given three matrices A,B and $M = AB$ partitionned as:

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

Then we can compute the partition of M from the two others as follow:

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

With

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