Performance Analysis of a Verified Linear Algebra Program in SML

Jesús Aransay[1*], **Jose Divasón**[2†]

[1] jesus-maria.aransay@unirioja.es, [2] jose.divasonm@unirioja.es
Departamento de Matemáticas y Computación
Universidad de La Rioja, Logroño, Spain

Abstract: In this paper we describe the performance results that have been obtained after executing a verified Linear Algebra SML program automatically generated from an Isabelle/HOL formalization. This SML program computes the reduced row echelon form of a matrix, using the well-known Gauss-Jordan algorithm. We explain how the code generator of Isabelle has been configured to obtain the SML program, the data structures involved, the accuracy of computations and the performance tests carried out over $\mathbb{Z}_2$, $\mathbb{Q}$ and $\mathbb{R}$ matrices with the SML implementation Poly/ML and the optimizing compiler MLton.

Keywords: Gauss-Jordan algorithm, Verified code, Computer Algebra, Isabelle

Introduction

Isabelle [NPW02] is a generic theorem prover (in the sense that different logics can be implemented on top of it). One of those logics is HOL (acronym for Higher-Order Logic) and Isabelle/HOL is the specialization of Isabelle for it. HOL is the most widely used logical setting by the Isabelle community, so that usually Isabelle/HOL is commonly referred to as Isabelle. Its expressiveness has been helpful to formalize relevant results in diverse fields, from software and hardware verification (for instance, the seL4 project on the formal verification of an operating system kernel [K+09]) to mathematical foundations (as, for instance, the Basic Perturbation Lemma, an intricate result in Homological Algebra [ABR08]).

Thanks to the facilities presented in the code generator of Isabelle, one can obtain certified executable programs in languages such as SML (abbreviation of Standard ML), OCaml, Haskell and Scala from a suitable subset of HOL specifications. This way, verified code is generated in those languages (in the sense that one can be sure that the output of the generated code will satisfy the properties proved in its Isabelle/HOL specification).

Isabelle is programmed in SML [Pau96], a well-known functional programming language. Features of SML include, among others, first-class functions, automatic memory management, parametric polymorphism, static typing, type inference, algebraic data types and pattern matching. Some SML compilers are Poly/ML [POL] and SML/NJ [NJ]. In addition, an optimizing SML compiler is MLton [MLT] which enables faster computations at the expense of greater compilation times.

* This work has been supported by the Spanish Government (project MTM2009-13842-C02-01 of MEC) and by FORMATH project, nr. 243847, of the FET program within the FP7 European Commission.
† This author is sponsored by the Universidad de La Rioja under a research grant FPI-UR-12.

The main objective of this paper is to present the results obtained in the execution of a verified SML program generated from the formalization in Isabelle/HOL of a well-known Linear Algebra algorithm: the Gauss-Jordan elimination. In order to do that, we present the specific setup that we have configured on top of the standard Isabelle code generator trying to obtain better performance. This article is divided as follows: in Section 1 we introduce the Gauss-Jordan algorithm whose properties we have formalized in Isabelle/HOL. Then, in Section 2 we present how the code generator of Isabelle has been configured in order to obtain an SML program. In Section 3 the experimental results obtained from the benchmarks are shown and explained, together with some conclusions and further work.

1 The Gauss-Jordan Algorithm

The Gauss-Jordan algorithm is a well-known and useful Linear Algebra algorithm. It is a method with several applications; for instance, it is used to solve systems of linear equations, compute the rank of matrices and calculate the inverse or the determinant of a matrix. Given any matrix over a field, the algorithm returns its *reduced row echelon form*. The reduced row echelon form of a matrix (hereafter, rref) is another matrix that satisfies several properties (see [Rom08] for a complete definition). The Gauss-Jordan algorithm and its properties have been formalized in Isabelle/HOL (see [ADb]); a technical report which shows how this formalization has been carried out is presented in [ADa].

2 Setting Up the Code Generator of Isabelle

Once we have formalized in Isabelle/HOL the Gauss-Jordan algorithm and its properties, we aim at generating a verified SML version from it. In order to do that, we have to suitably configure the code generator of Isabelle to obtain an efficient implementation in SML.

2.1 Mapping Matrices from Isabelle to SML

The data types used both in Isabelle/HOL and SML are crucial, because of their influence in the formalization and the performance tests. In Isabelle/HOL, two representations have been used. In the first one the correctness of the algorithm is proved. The second one is introduced to obtain a better performance, preserving the correctness since the equivalence between operations in both representations is formally proved inside Isabelle/HOL. The idea is that formalizations have to be carried out over an abstract representation (the first one) which facilitates this work, whereas an efficient concrete representation (the second one) is only used during the code generation process.

The first representation defines matrices as functions over finite domains. The idea of having underlying finite types for the indices of matrices has great advantages, from the formalization point of view. For instance, the type system enforces that matrix operations (such as addition or substraction) are only performed on matrices of equal dimensions (this would not be the case if we were to use, for instance, a list of lists to represent matrices). The algorithm can be executed using this representation, but it offers very poor performance.

The second representation is introduced to solve this performance problem. We have chosen nested *iarrays* (*iarray* stands in the Isabelle library for *inmmutable arrays*, which are represented in the SML Library by the structure *Vector*) to represent matrices. As we will show in Section 3, this representation offers a better performance; additionally, it preserves the functional style of operations. This means that the definitions of operations over both representations are rather similar and this makes easier proving the equivalence between an operation in the functional representation of matrices and the corresponding one in the representation with *iarrays*. As we have already said, the Isabelle type *iarray* is implemented by the *Vector* structure of the SML Library. The *Vector* structure defines polymorphic vectors, immutable sequences with constant-time access (*a priori*, more efficient and faster than, for instance, lists of lists).

2.2 Mapping Matrices Elements from Isabelle to SML

The Gauss-Jordan algorithm is designed to be carried out over matrices whose elements belong to a field. Now we show how the code generator of Isabelle has been configured to obtain efficient code for executing this algorithm over matrices with elements in $\mathbb{Z}_2$, $\mathbb{Q}$ or $\mathbb{R}$.

2.2.1 The $\mathbb{Z}_2$ Implementation

To represent $\mathbb{Z}_2$ in Isabelle, we have made use of the *bit* type provided in the Isabelle library. We have considered three different alternatives to implement the Isabelle type *bit* in SML:

1. Using the *Bool* SML structure (mapping 0 of Isabelle type *bit* to the *false* boolean value of SML and 1 of Isabelle type *bit* to the *true* boolean value of SML).
2. Using the *IntInf* SML structure (mapping the *bit* values to the corresponding ones of *IntInf*) implementing the arithmetic operations by *extensional* definitions (by a table).
3. Using the *IntInf* SML structure implementing the arithmetic making use of integer operations modulo 2.

We have decided to use the third option. Experimentally, we have checked that a better performance is achieved using the *IntInf* structure than using the *Bool* one (in our case, allocating *IntInf.int* elements in memory is faster). Regarding the implementation of operations, better results have been obtained mapping the Isabelle *bit* operations to *IntInf* operations modulo 2 (a 15% reduction in the execution time compared to using an implementation by a table). Therefore, the code generator of Isabelle has been configured as follows to obtain an efficient representation of $\mathbb{Z}_2$ and its arithmetic in SML:

```
code_datatype "0::bit" "(1::bit)"
code_type bit (SML "IntInf.int")
code_const "0::bit" (SML "0")
code_const "1::bit" (SML "1")
code_const "op + :: bit => bit => bit"
  (SML "IntInf.rem ((IntInf.+ ((_), (_))), 2)")
code_const "op * :: bit => bit => bit" (SML "IntInf.* ((_), (_))")
code_const "op / :: bit => bit => bit" (SML "IntInf.* ((_), (_))")
```

Then, in SML a $\mathbb{Z}_2$ matrix is a nested vector of elements of type *IntInf.int*.

2.2.2 The $\mathbb{Q}$ Implementation

$\mathbb{Q}$ is representated in Isabelle by fractions of elements of type *int*. Without doing any modification in the code generator of Isabelle, a similar data type in SML is obtained (this representation allows to exploit the full arithmetic power of the *IntInf* SML structure, obtaining arbitrary precision):

```
data type rat = Frct of (IntInf.int * IntInf.int);
```

2.2.3 The $\mathbb{R}$ Implementation

To implement the Isabelle *real* type ($\mathbb{R}$), the Isabelle code generator offers two alternatives: using fractions of integers[1] (obtaining the same performance and precision than in $\mathbb{Q}$ because it is the same representation) or serialising it to machine reals in SML (type *Real.real*). This last one is formally inconsistent, but convenient from a performance point of view; as far as one only pretends to carry out computations in the target language (and not bringing them back to Isabelle for proving purposes), the formalization will preserve its soundness.

2.3 The Generated Code

Once the data type refinements for both matrices and matrices elements have been formalized in Isabelle, an efficient and certified program can be obtained. We must remark that the code generator of Isabelle allows us to generate stand-alone verified SML programs. In our case, from the Isabelle formalization presented in [ADb] we have generated a verified SML program which computes the rank and the rref of a matrix using the Gauss-Jordan algorithm. The exported code takes up 1743 lines (65'8 kb) and it can be downloaded from [ADb].

3 Time Comparisons

3.1 The Benchmarks

The tests have been run on an Intel Core i3-370M Processor (2 cores of 2.4 GHz) with 4GB of RAM and Ubuntu GNU/Linux 11.10. The SML compilers that have been used are Poly/ML (versions 5.2 and 5.5) and MLton 20100608; the benchmarks consisted in computing the rref of randomly generated matrices of different sizes. To enrich the comparison, we will make use of some matrices obtained from neuronal digital images. The total elapsed time in computations is divided into two columns:

- **Processing Time:** This is the time that the compilers take up to read and process the file with the input matrix.
- **Execution time:** This is the time that the SML program spends on applying the Gauss-Jordan algorithm and printing the result.

[1] By default the code generator of Isabelle will make the implementation this way.

3.2 Results Obtained with $\mathbb{Z}_2$ Matrices

Table 1 shows that a remarkable performance is achieved in $\mathbb{Z}_2$ by using Poly/ML. MLton is slower: it increases the processing time without reducing the execution one. Both the processing and the execution times grow following a linear pattern with respect to the number of elements of each matrix using Poly/ML. Moreover, the randomness of the matrices has an influence in the time. For instance, for a 1024×1024 $\mathbb{Z}_2$ matrix generated from a digital image of a neuron (Figure 3.2), the execution time is reduced to half (to 21 seconds, whereas a randomly generated matrix of a similar size takes up 45.1 seconds). This is caused by the presence of numerous zero columns (the SML program is faster because the Gauss-Jordan algorithm doesn't process these zero columns). Thus, it is reasonable to think that if the algorithm is applied to matrices obtained from the *real world*[2], the execution time (the processing one did not) could be reduced significantly.

3.3 Results Obtained with $\mathbb{Q}$ Matrices

Table 2 shows the results obtained for matrices whose elements belong to $\mathbb{Q}$. In this case MLton is faster than Poly/ML: it reduces the execution time with a moderate increment of the processing time. Times also grow in the same linear pattern with respect to the number of elements of each matrix in both SML compilers.

The implementation for $\mathbb{Q}$ matrices explained in Section 2.2.2 allows arbitrary precision, at the expense of a slower performance. It can be noted that most of the computing time of $\mathbb{Q}$ matrices is spent on the integer arithmetic. Profiling the computation of the rref of a 70×70 rational matrix, one can see that the algorithm spends most of the time on performing integer divisions (functions *bigRem*, *bigQuot* and *bigDivMod*), related to the concrete representation of elements of $\mathbb{Q}$, whereas matrices operations (*plus_iarray*, *mult_iarray*) take up minor times.

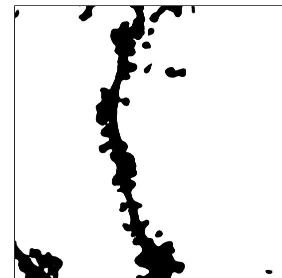

Figure 1: Digital image (1024×1024 px.) of a neuron.

```
18.39 seconds of CPU time (0.54 seconds GC)
             function                cur    raw
-------------------------------- ----- -------
Primitive.IntInf.bigRem          29.6% (5.61s)
Primitive.IntInf.bigQuot         24.9% (4.71s)
Primitive.IntInf.bigDivMod       13.0% (2.47s)
Primitive.IntInf.make             9.0% (1.71s)
GJ_GENERATED_CODE.plus_iarray.fn  0.1% (0.01s)
GJ_GENERATED_CODE.mult_iarray.fn  0.1% (0.01s)
```

3.4 Results Obtained with $\mathbb{R}$ Matrices

In Section 2.2.3 we have presented two possible implementations for the Isabelle type *real*: using fractions of integers or using machine numbers (type *Real.real* in SML). The first representation

[2] *Real world* matrices usually present repetitions or patterns which increase the computing performance.

$\mathbb{Z}_2$ matrices				
	Poly/ML		MLton	
Size (n)	Processing Time (seconds)	Execution Time (seconds)	Processing Time (seconds)	Execution Time (seconds)
50	0.0	0.0	0.8	0.0
100	0.3	0.0	4.0	0.1
150	0.6	0.1	16.3	0.3
200	1.0	0.3	54.6	0.6
250	1.6	0.7	124.7	1.3
300	2.2	1.2	262.9	2.2
350	3.0	1.9	480.4	3.5
400	4.6	2.9	809.2	5.2
500	7.3	6.1	-	-
600	10.6	9.8	-	-
800	19.8	24.1	-	-
1000	31.8	45.1	-	-
1200	53.7	79.7	-	-
1400	65.6	143.0	-	-
1600	107.0	200.5	-	-

Table 1: Elapsed time (in seconds) to process randomly generated $(\mathbb{Z}_2)^{n\times n}$ matrices and computing their corresponding rrefs using the Gauss-Jordan algorithm with Poly/ML 5.5 and MLton 20100608.

is the same as the one chosen for $\mathbb{Q}$ matrices, so the results are the same as the ones presented in Table 2. The results for the second representation are presented in Table 3.

MLton is faster than Poly/ML executing the program, although at the expense of a huge increment in the processing time. In addition, it is easy to see that $\mathbb{R}$ matrices implemented using machine numbers have a better performance than $\mathbb{Q}$ ones (or $\mathbb{R}$ matrices whose elements have been implemented as fractions of integers). For instance, for the same 100×100 matrix, in $\mathbb{Q}$ takes up 200.9 seconds using Poly/ML and 84.1 using MLton, whereas in $\mathbb{R}$ is almost immediate using Poly/ML. Nevertheless, operations involving $\mathbb{Q}$ matrices have arbitrary precision (because integer arithmetic is used) and, on the contrary, some precision errors could appear working with $\mathbb{R}$ matrices due to the implementation of floating-point numbers. The user can decide which serialisation of the Isabelle *real* type fits better for matrices elements in each use case, according to the size of the input matrix and the required precision.

3.5 Conclusions and Further Work

We have presented the results obtained in the execution of a verified SML program generated from the formalization in Isabelle/HOL of a well-known Linear Algebra algorithm. We must remark that the main objective of this work is to experiment in order to get better performance. In our opinion, we have achieved a verified SML code to compute the rref of matrices using the Gauss-Jordan algorithm with a remarkable performance. Taking a quick look at Tables 1, 2 and 3, it can be checked that the processing and execution times grow following a linear pattern with respect to the number of elements of each matrix. In general, the execution limits will be determined by the memory and not by the time (as it can be seen in Table 3). Another remarkable fact is that MLton, despide being an “optimized SML compiler”, does not outperform Poly/ML

Rational matrices				
	Poly/ML		MLton	
Size (n)	**Processing Time (seconds)**	**Execution Time (seconds)**	**Processing Time (seconds)**	**Execution Time (seconds)**
10	0.0	0.0	0.2	0.0
20	0.0	0.2	0.3	0.0
30	0.0	1.0	0.6	0.5
40	0.1	3.7	0.9	1.5
50	0.1	10.2	1.4	4.5
60	0.2	22.7	1.9	9.6
70	0.3	43.0	2.7	18.4
80	0.5	77.0	3.5	32.7
90	0.6	126.9	4.5	54.1
100	0.7	200.9	6.0	84.1

Table 2: Elapsed time (in seconds) to process random $\mathbb{Q}^{n\times n}$ matrices (with elements between -10 and 10) and computing their rrefs using the Gauss-Jordan algorithm with Poly/ML 5.5 and MLton 20100608.

Real matrices				
	Poly/ML		MLton	
Size (n)	**Processing Time (seconds)**	**Execution Time (seconds)**	**Processing Time (seconds)**	**Execution Time (seconds)**
10	0.0	0.0	0.8	0.0
20	0.0	0.0	2.5	0.0
30	0.0	0.0	6.4	0.0
40	0.1	0.0	13.8	0.0
60	0.2	0.0	56.9	0.0
80	0.3	0.0	164.3	0.0
100	0.6	0.2	361.6	0.1
200	3.7	0.7	9145.4	0.5
300	9.6	2.4	-	-
400	20.3	5.9	-	-
500	37.3	10.2	-	-
600	65.8	20.5	-	-
700	98.6	44.4	-	-
800	Segmentation Fault	-	-	-

Table 3: Elapsed time (in seconds) to process random $\mathbb{R}^{n\times n}$ matrices (with elements between -10 and 10) and computing their rrefs using the Gauss-Jordan algorithm with Poly/ML 5.2 and MLton 20100608.

significantly; the experimental results show that Poly/ML works quite well computing the rref of $\mathbb{Z}_2$ and $\mathbb{R}$ matrices. In fact, MLton needs a lot of time to process those matrices and it doesn't achieve a reduction of the execution times that Poly/ML take up (see Table 1 and Table 3). Nevertheless, an improvement on the performance is obtained working in MLton with $\mathbb{Q}$ matrices: the time is reduced to half (see Table 2). As it was expected, scientific computational commercial software obtains better performance than our SML program. For instance, *Mathematica 7* takes up 0.25 seconds in computing the rref of a 100×100 matrix with rational elements between -10 and 10, whereas our SML program takes up 84.1 seconds using MLton (both of them with exact arithmetic, see Table 2). Moreover, *Mathematica 7* takes up 109.14 seconds for a 500×500

rational matrix. From our point of view, the SML program performs quite remarkably, taking into account that it was designed for being formalized and not for computational performance.

As further work, our SML program could be optimized to achieve better performance. The first way for doing that is to make new refinements on the formalized algorithm in Isabelle/HOL. Our SML program has been automatically generated from an Isabelle/HOL development, so if we optimize the algorithm in Isabelle/HOL we will obtain more efficient SML code. The second way is to parallelize it. MLton allows us to export SML functions to C, where we could explore how to parallelize the generated C code. A different possibility would be to use MultiMLton [MUL], a compiler that targets scalable multicore platforms and provides a sophisticated runtime system tuned to efficiently handle large numbers of lightweight threads.

Bibliography

[ABR08] J. Aransay, C. Ballarin, J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning* 40(4):271–292, 2008.

[ADa] J. Aransay, J. Divasón. Formalization and execution of Linear Algebra: from theorems to algorithms. Technical report.
http://wiki.portal.chalmers.se/cse/uploads/ForMath/felafte

[ADb] J. Aransay, J. Divasón. Formalization of the Gauss-Jordan algorithm in Isabelle/HOL. Formal development.
http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan/

[K+09] G. Klein et al. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. Pp. 207–220. ACM, 2009.

[MLT] The MLton Home Page.
http://mlton.org/

[MUL] The MultiMLton Home Page.
http://multimlton.cs.purdue.edu/mML/Welcome.html

[NJ] The Standard ML of New Jersey Home Page.
http://www.smlnj.org/

[NPW02] T. Nipkow, L. Paulson, M. Wenzel. *Isabelle/HOL: A proof assistant for Higher-Order Logic*. Springer, 2002.

[Pau96] L. C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[POL] The Poly/ML Home Page.
http://www.polyml.org/

[Rom08] S. Roman. *Advanced Linear Algebra*. Graduate Texts in Mathematics. Springer, 2008.