VERIFIED COMPUTER LINEAR ALGEBRA

JESÚS ARANSAY AND JOSE DIVASÓN

Abstract. We present the execution tests and benchmarks of some Linear Algebra programs generated from their verified formalisation in Isabelle/HOL; more concretely, the Gauss-Jordan algorithm and the QR decomposition, together with the techniques used to improve the performance of the extracted code, are described.

Introduction

Computer Algebra systems are commonly seen as *black boxes* in which one has to trust, but they are no error-free [6]. Theorem provers are designed to prove the correctness of algorithms and mathematical results, but this task is far from trivial and it has a significant cost in terms of performance. One of the most accepted techniques to verify a program is to describe the algorithm within the language of the proof checker, then extract code and run it independently. Following this strategy, we have formalised some well-known Linear Algebra algorithms in Isabelle/HOL and then code is extracted to SML and Haskell. In this paper, we briefly present the techniques that we followed to improve the performance of the generated code, as well as some execution tests. This code cannot compete with Computer Algebra systems in terms of efficiency, but it pays off in feasibility and the results also show the code to be useful for matrices of considerable size.

1. Verified Computing

Isabelle is a generic interactive theorem prover, in the sense that different logics can be implemented on top of it. The most widespread logic is HOL (Higher-Order Logic, whose Isabelle implementation is referred to as Isabelle/HOL), which includes interesting features such as code generation. The HOL Multivariate Analysis (or *HMA* for short) library is a set of Isabelle/HOL theories which contains theoretical results in mathematical fields such as Analysis and Linear Algebra. It is based on the work by Harrison in HOL Light and one of the fundamentals of the library is the representation of n-dimensional vectors (type `vec`) by means of functions from a finite type [7]. We have formalised several algorithms and their applications (Gauss-Jordan algorithm, QR decomposition...) based on the *HMA* library [4, 5], all of them are defined making use of the representation based on `vec` (functions over finite domains). Following the *data refinement* strategy, we have refined the algorithms to the more efficient representation `iarray`, which is later code-generated to its corresponding implementation in SML (*Vector.vector*) and Haskell (*IArray.array*). This representation defines polymorphic vectors, immutable sequences with constant-time access. Furthermore, *serialisations* are a process to map Isabelle types and operations to the corresponding ones in the target languages. They are common practice to avoid Isabelle generating from scratch. We focus our work on $\mathbb{Z}_2$, $\mathbb{Q}$, and $\mathbb{R}$ (types `bit`, `rat`, and `real` in Isabelle). We serialised them to their corresponding structures in SML and Haskell (see [3] for further details). Let us note that `real` can be serialised to both fractions of integers (obtaining arbitrary precision) and floating-point numbers in the target languages. In the latter case, although the original algorithm is formalised, the computations cannot be trusted.

2. Experimental Outcomes

Let us present the performance tests that we have carried out to our verified programs obtained from the formalisation of the Gauss-Jordan algorithm [4] and the QR decomposition [5]. The times presented throughout the tables are expressed in seconds. The benchmarks have been carried out in a laptop with an Intel Core i5-3360M processor, 4 GB of RAM, Poly/ML 5.6, Ubuntu 14.04, GHCi 7.6.3, and Isabelle2016. We have noticed that Poly/ML, which is an interpreter, performs as well as an optimiser compiler as MLton when executing our generated code (times are similar, so we just present here the Poly/ML ones).

2.1. **The Gauss-Jordan algorithm.**

We have formalised a version of the well-known Gauss-Jordan algorithm to compute the *reduced row echelon form* (from here on, rref) of a matrix in Isabelle/HOL, as well as its applications such as the computation of ranks and determinants. The algorithm has been formalised over an arbitrary field. Some preliminary experiments had been already carried out in Poly/ML [8], but developers of the compiler suggested us improvements that eliminated the *processing* time of the input matrices (which showed to be the real bottleneck, see the figures in [2]). More concretely, in our first experiments, the input matrices were directly introduced in the system by means of an explicit binder as static data. The Poly/ML maintainer also modified the system behaviour in the SVN version of the tool to improve the processing capabilities of big inputs. From our side, we changed our methodology to input matrices from external files by means of an ad-hoc parser. Processing input matrices this way proved to be no time consuming. In addition, we have serialised the `bit` type in Isabelle to the booleans in SML and Haskell, whereas in the results presented in [2] the `bit` type was serialised to integers. The experimental tests presented here shows that this change provides a significant improvement of the computation times (the computation of the *rref* of a 800×800 $\mathbb{Z}_2$ matrix needed 43.9s in Poly/ML, now only 15.96s).

Let us show a fragment of the experiments carried out with the new methodology. The input matrices can be downloaded from [1]. Table 1 presents the times of computing the *rref* of randomly generated $\mathbb{Z}_2$ matrices. The same randomly generated matrices have been used across the different systems. It is well-known that the Gauss-Jordan algorithm has arithmetic complexity of $\mathcal{O}(n^3)$. The computing times of our programs also grow cubically (in both SML and Haskell) with respect to the number of elements in the input matrices (let us note that the underlying field notably affects the performance and can even affect the complexity bounds).

Size (n)	**Poly/ML**	**Haskell**
100	0.04	0.36
200	0.25	2.25
400	2.01	17.17
800	15.96	131.73
1200	62.33	453.57
1600	139.70	1097.41
2000	284.28	2295.30

Table 1. Time to compute the *rref* of randomly generated $\mathbb{Z}_2$ matrices.

An interesting case appears when working with matrices over $\mathbb{Q}$. Following the standard code generator setup to SML (serialising `rat` to fractions of *IntInf.int*), we detected that the greatest amount of time was spent in reducing fractions (operations `gcd` and `divmod`). Serialising the Isabelle operation `gcd` to the corresponding built-in Poly/ML function (which is not part of the SML Standard Library, but particular to the compiler), decreased by a factor of 20 the computing times. In addition, the natural serialisation for the Isabelle operation `divmod` would be *IntInf.divmod* in SML, which returns the pair (*i IntInf.div j*, *i IntInf.mod j*) where the result of *div* is truncated toward negative infinity (for example, `divmod (-10, 6)` returns `(-2, 2)`). However, in SML (and Haskell) there also exists a more efficient operation *IntInf.quotrem*, which returns the pair (*i IntInf.quot j*, *i IntInf.rem j*) where *quot* is integer division truncated toward zero (for instance, `quotrem (-10, 6)` returns `(-1, -4)`). Since in the case of $\mathbb{Q}$ matrices we only divide when normalising fractions in SML (and thus, we only divide by divisors) the remainder is always 0, so we can serialise to the more efficient operation *IntInf.quotrem*. Table 2 presents the performance tests to compute determinants of randomly generated matrices over $\mathbb{Q}$. Apparently, Haskell takes advantage of its native *Rational* type to match the results of Poly/ML. Table 3 presents the times used to compute the *rref* of matrices over $\mathbb{R}$ represented as floating-point numbers. In this case, numerical stability problems arise (the *rref* of matrices contains small nonzero entries), as also happens in Computer Algebra systems. Once again, Poly/ML outperforms Haskell and the performance is cubic as well.

Size (n)	**Poly/ML**	**Haskell**
10	0.01	0.01
20	0.02	0.03
40	0.21	0.24
80	3.77	3.53

TABLE 2. Time to compute the *rref* of randomly generated $\mathbb{Q}$ matrices.

2.2. **The QR Decomposition.** We have also formalised the QR decomposition in Isabelle/HOL and its application to compute the *least squares approximation* to an unsolvable system of linear equations. The QR decomposition decomposes a real matrix A into the product of two different matrices Q and R (the first one containing an *orthonormal* collection of vectors, the second one being upper triangular). The QR decomposition is important, among other things, since it significantly reduces round-off errors when computing the least squares approximation (which can also be solved by means of the Gauss-Jordan algorithm). Since the Isabelle type `real` is used and square roots are involved in the algorithm (they are necessary to normalise the vectors), the use of a representation of `real` based on fractions of integers (*IntInf.int*) in SML is not possible (square roots are not computable in such a setting). A development by Thiemann was published in the Archive of Formal Proofs [9]. This development provides a refinement for real numbers of the form $p + q\sqrt{b}$ (with $p, q \in \mathbb{Q}$, $b \in \mathbb{N}$). We make use of this development to get exact symbolic computations. The performance obtained by means of this refinement depends much on the size of the entries. For example the computation of the QR decomposition of a 10×10 matrix requires several minutes.

Size (n)	**Poly/ML**	**Haskell**
100	0.03	0.38
200	0.25	2.62
400	1.85	19.51
800	13.99	148.20

TABLE 3. Time to compute the *rref* of randomly generated $\mathbb{R}$ matrices.

The other possibility is the use of the serialisation to floating-point numbers, which is specially interesting (despite the computations cannot be trusted) when comparing the precision obtained with the one of the Gauss-Jordan development. We present an experiment involving the Hilbert matrix (which is known to be very ill-conditioned) in dimension 6, H_6. We have computed the least squares solution to the system $H_6\, x = (1\,0\,0\,0\,0\,0\,5)^T$ using both the QR decomposition and the Gauss-Jordan algorithm. The exact solution of the least squares approximation can be obtained symbolically: `"["-13824","415170","-2907240","7754040","-8724240","3489948"]"` . If we now use the refinement from Isabelle `real` to SML *floats*, and both algorithms to solve the least squares problem, the following solutions are obtained:

- QR solution using floats:
  `[-13824.0,415170.0001,-2907240.0,7754040.001,-8724240.001,3489948.0]`
- Gauss-Jordan solution using floats:
  `[-13808.64215,414731.7866,-2904277.468,7746340.301,-8715747.432,3486603.907]`

As it can be noticed, the QR decomposition is much more precise than the Gauss-Jordan algorithm. Table 4 shows the performance obtained with this serialisation.

3. Conclusions and Further Work

We have presented the results obtained in the execution of verified Linear Algebra programs generated from their formalisation in Isabelle/HOL, as well as some serialisations and data refinement devoted to improve the performance. The verified code cannot compete with Computer Algebra systems, but it is usable with matrices of remarkable dimensions. This is an attempt to try to reduce the existing gap between software verification and working software. As a future work, the study of the performance of our verified algorithms to compute the echelon form and the Hermite normal form of a matrix would be desirable. For the moment, the Hermite normal form of a 25×25 integer matrix can be completed in seconds, but we run out of memory in higher dimensions.

Size (n)	Poly/ML
100	0.748
160	4.621
220	18.941
280	42.100
340	97.360
400	183.754

Table 4. Time to compute the QR decomposition of Hilbert matrices over $\mathbb{R}$.

References

[1] `http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2-Generalized/`.

[2] J. Aransay and J. Divasón. Performance Analysis of a Verified Linear Algebra Program in SML. In Fredlund and Castro, editors, *TPF 2013*, pages 28 – 35, 2013.

[3] J. Aransay and J. Divasón. Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. *J. of Func. Programming*, 25, 2015.

[4] J. Divasón and J. Aransay. Gauss-Jordan Algorithm and Its Applications. *Archive of Formal Proofs*, September 2014. `http://afp.sf.net/entries/Gauss_Jordan.shtml`, Formal proof development.

[5] J. Divasón and J. Aransay. QR Decomposition. *Archive of Formal Proofs*, 2015. `http://afp.sf.net/entries/QR_Decomposition.shtml`, Formal proof development.

[6] A. J. Durán, M. Pérez, and J. L. Varona. Misfortunes of a mathematicians' trio using Computer Algebra Systems: Can we trust? *Notices of the AMS*, 61(10):1249 – 1252, 2014.

[7] J. Harrison. The HOL Light Theory of Euclidean Space. *J. of Autom. Reasoning*, 50(2):173 – 190, 2013.

[8] The Poly/ML website. `http://www.polyml.org/`, 2015.

[9] R. Thiemann. Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$. *Archive of Formal Proofs*, February 2014. `http://afp.sf.net/entries/Real_Impl.shtml`, Formal proof development.

Universidad de La Rioja
E-mail address: `{jesus-maria.aransay,jose.divasonm}@unirioja.es`