

Statically Typed Linear Algebra in Haskell

(extended abstract)

Frederik Eaton

Computer Science Department
University College London
frederik@ofb.net

Abstract

Numerical computations are often specified in terms of operations on vectors and matrices. This is partly because it is often natural to do so; but it is also partly because, being otherwise useful, such operations have been provided very fast implementations in linear algebra libraries such as ATLAS (implementing BLAS), LAPACK, fftw, etc. Due to their better cache awareness and use of specialized processor instructions, a high-level invocation of an operation such as matrix multiplication using these libraries may execute orders of magnitude more quickly than a straightforward low-level implementation written in, say, C. The combination of efficiency and expressivity has made the framework of linear algebra an exceedingly popular one for scientists. For example, Matlab[1], an interpreter of a simple linear algebra language, has become a standard research tool in many fields.

The process of expressing an algorithm in terms of matrices can be error-prone. Matlab and other popular matrix languages are dynamically-typed, which means that type errors are only detected at run-time. Even statically typed languages rarely keep track of more information in an object type than its tensor rank (e.g., to discriminate between matrices and vectors) and element type.

If we could additionally expose object dimensions to the type system then we would ideally be able to detect a much wider variety of common errors at compile time than is currently possible. This would in turn make it easier to build and maintain larger numerics-intensive software projects.

We call our idea of exposing dimensions to the type system “statically typed linear algebra”. We have written a prototype implementation in Haskell, which is based on Alberto Ruiz’s GSLHaskell[2] and which uses techniques from Kiselyov and Shan’s “Implicit Configurations” paper[3].

The presentation will cover the key aspects of our design such as the use of GADTs to combine matrix and vector types, and the use of higher-rank types and staging (namely, Template Haskell) to closely approximate a dependent-type system. Then, we will give a demonstration of interactive use, and compare our system to existing systems in the areas of speed and usability.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Language*

Classifications]: Haskell; D.3.3 [*Programming Techniques*]: Language Constructs and Features—constraints; abstract data types; polymorphism

General Terms Design, Languages

Keywords Linear algebra, Higher-rank polymorphism, Existential types, Template Haskell, Staging

1. System Overview

The problem of designing a system in which dimensions are exposed to the type system is, by itself, an easy one. In our system, we follow the straightforward approach: dimensions are represented as type parameters to vectors. Matrices are vectors whose dimensions are pair types.

In implementing such an approach, there are two primary difficulties. The first is to make it possible for the user to create new objects of any size at runtime. For instance, we might like to read in a matrix from a file at runtime, where the dimensions of the matrix are contained in the file. We solve this problem using type classes and higher order polymorphism as demonstrated in [3].

The second difficulty stems from the solution to the first. In the higher-order polymorphism method of [3], new types are “constructed” at runtime, but they are only accessible within functions from which they cannot escape. As a result, most of the computation in a typical program must take place within those functions. This makes it hard to use the system in an interactive fashion - objects with the new types will never be accessible from the top level of the interpreter. Restoring the ability to use the system interactively was a high priority. Much of the usefulness of systems such as Matlab arises from the way in which they allow researchers to conduct experiments with matrix quantities and small pieces of code interactively within an interpreter, before incorporating them into a larger program. We overcame this hurdle by providing a set of mechanisms for constructing vectors of variable size in Template Haskell splices.

The result is very promising. The system as it stands still suffers from a number of additional limitations which make it impossible for us to recommend adopting it for serious use, but those limitations should be addressed by future improvements to compiler technology.

The biggest current limitation is that although we are able to guarantee operand conformability through the type system, the error messages that are returned by most compilers are much less helpful in tracking down conformability bugs, than the error messages returned at runtime by Matlab and Octave¹[5]. So, even for

Copyright is held by the author/owner(s).

Haskell '06 September 18-20, Portland, Oregon.
ACM 1-59593-489-8/06/0009.

¹Octave is a free software program which is mostly compatible with Matlab, which we use for our tests. It is widely used as a Matlab replacement, and was roughly half as fast as Matlab in 2003[4].

medium-sized algorithms, we find that debugging runtime errors in Octave is much faster than debugging compile-time errors in Haskell (under GHC).

Perhaps surprisingly, we don't see performance as being an issue with our system. Since we use similar optimized linear algebra libraries to the ones which Octave uses, we get similar performance for linear algebra operations. Additionally, it turns out that GHC (and even GHCi) is much faster than Octave for tight loops. We implemented a medium-sized algorithm in both our system and Octave, and found that the Haskell version was 63% faster than the Octave version.

The source code for our library is online, together with a draft of a more extensive paper.[6]

2. An Example

Our library design is based on a *Vector* type-class, of which all vector data types are implementors. Almost all vector operations are members of the *Vector* class or its subclasses (*NumVector*, etc.). Dimensions are types of class *Dom*.

```
class Vector v e | v → e where
  vector :: Dom a ⇒ (a → e) → v a
  (!) :: Dom a ⇒ v a → a → e
  ...
```

There is an *Array*-based back-end called *AVector* which supports elements of any type, and a fast *GSLHaskell*-based back-end called *FVector* which only supports elements of type *Double*.

```
instance Vector (AVector e) e where ...
instance Vector FVector Double where ...
```

We will present a short interactive session below.

First we will construct a set of data points, and then we will try to fit them to a curve using least squares. The points are on a parabola, but with random noise added.

Here we initialize the data abscissas from a list.

```
let x = $(qFast (listVec [0, 0.5..10]))
```

Briefly, *listVec* is used to turn a list into a vector, and *qFast* quotes the vector so that it can be used as a Template Haskell splice; it also selects the *FVector* back-end². Actually, *listVec* wants a polymorphic function of a vector as its second argument, and *qFast* passes its argument such a function; the vector escapes as a value of type *ExpQ*, which encodes an expression in Template Haskell:

```
listVec :: (Vector v e) ⇒ [e] →
  (∀ n. (ReflectNum n) ⇒ v (L n) → w) → w
qFast :: ((∀ w. (Dom w, GetType w) ⇒ FVector w → ExpQ)
  → ExpQ)
  → ExpQ
```

Here we generate the random noise, which is uniformly distributed between -0.5 and 0.5 :

```
r' ← randIO ≍ (return. ('asTypeOf'x). (λ e → e - 0.5))
```

The use of *asTypeOf* ensures that our noise vector has the correct dimensions.

Now we create the output vector, which is a polynomial function of the input *x*:

```
let y = 3 × (x ** 2) - 20 × x + 14 + 10 × r
```

Since *FVector* is an instance of *Num*, the constants 3, or 2, etc. are instantiated as vectors of all 3's, or 2's, etc.

To fit a polynomial using least squares, we must create a matrix whose entries are powers of the input vector elements. The element in row *i*, column *j* (counting from 0) should be $(x_i)^j$:

```
let a = $(dim2 21 3) (byCol
  (λ i → x ** . (fromIntegral $ fromEnum i)))
```

The *dim2* function creates a type-level assertion that the quantity is a matrix with the specified dimensions; *** .* is element-wise exponentiation by a scalar. The type of *byCol* is:

```
byCol :: (Vector v e, Dom b, Dom a) ⇒
  (b → v a) → v (a, b)
```

Now we calculate the coefficients using the normal equations.

```
let c = pinv (trans a * > a) @ > (trans a @ > y)
```

Here, ** >* is matrix multiplication:

```
(* >) :: (NumVector v e, Dom c, Dom b, Dom a) ⇒
  v (a, b) → v (b, c) → v (a, c)
```

@ > is matrix-vector multiplication, *pinv* calculates the pseudo-inverse, and *trans* the transpose of a matrix:

```
trans :: (Vector v e, Dom b, Dom a) ⇒
  v (a, b) → v (b, a)
```

As expected, the least-squares estimates have values close to the actual coefficients:

```
> c
<13.961770, -19.843045, 2.964106>
```

3. Acknowledgements

Thanks to Alberto Ruiz for *GSLHaskell*, and to Oleg Kiselyov, Chung-chieh Shan, Ralf Lammel, and John Meacham for useful discussions.

References

- [1] Matlab. <http://www.mathworks.com/>.
- [2] A. Ruiz. Matrix computations in haskell based on the gsl. <http://dis.um.es/~alberto/GSLHaskell/matrix.pdf>, June 2005.
- [3] O. Kiselyov and C. Shan. Functional pearl: implicit configurations—or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM Press.
- [4] SciViews. <http://www.sciviews.org/benchmark/index.html>, August 2003.
- [5] J.W. Eaton. GNU Octave: a high-level interactive language for numerical applications. *GNU/Free Software Foundation, Boston*, 1998.
- [6] F. Eaton. Statically typed linear algebra in haskell (papers and source code). <http://ofb.net/~frederik/stla/>, July 2006.

²There is also a *qArray* function for the *Array*-based back-end.