A Simple and Practical Linear Algebra Library Interface with Static Size Checking

Akinori Abe Eijiro Sumii
Tohoku University
{abe,sumii}@kb.ecei.tohoku.ac.jp

1. Introduction

While advanced type systems—specifically, dependent types on natural numbers—can statically ensure consistency among the sizes of collections such as lists and arrays [**? ? ?**], such type systems generally require non-trivial changes to existing languages and application programs, or tricky type-level programming. We have developed a linear algebra library interface that guarantees consistency (with respect to dimensions) of matrix (and vector) operations by using *generative phantom types* as fresh identifiers for statically checking the equality of sizes (i.e., dimensions). This interface has three attractive features in particular.

- It can be implemented only using fairly standard ML types and its module system. Indeed, we implemented the interface in OCaml (without significant extensions like GADTs) as a wrapper for an existing library.
- For most high-level operations on matrices (e.g., addition and multiplication), the consistency of sizes is verified statically. (Certain low-level operations, like accesses to elements by indices, need dynamic checks.)
- Application programs in a traditional linear algebra library can be easily migrated to our interface. Most of the required changes can be made mechanically.

To evaluate the usability of our interface, we ported to it a practical machine learning library (OCaml-GPR [**?**]) from an existing linear algebra library (Lacaml [**?**]), thereby ensuring the consistency of sizes.

2. Our idea

Let `'n vec` be the type of `'n`-dimensional vectors and `('m, 'n) mat` be the type of `'m`-by-`'n` matrices. The formal type parameters `'m` and `'n` are instantiated with actual types that represent the sizes of the vectors or matrices. Here we only explain how the dimensions of vectors are represented since those of matrices can be similarly represented.

The abstract type `'n vec` can be implemented as any data type that can represent vectors, e.g., `float array`, where the type parameter `'n` is phantom, meaning that it does not appear on the right hand side of the type definition. A phantom type parameter is often instantiated with a type that has no value (i.e., no constructor), which we call a *phantom type*[1]. The type `'n vec` must be made abstract by hiding its implementation in the module signature so that the size information in the (phantom) type parameter `'n` is not ignored by the typechecker.

It is relatively straightforward to represent dimensions (size information) as types by, for example, using type-level natural numbers when this information is decided at compile time. The main problem is how to represent dimensions that are unknown until runtime. Consider the following code for example:

```
let (x : ?[1] vec) = loadvec "file1" in
let (y : ?[2] vec) = loadvec "file2" in
add x y (* 'n vec -> 'n vec -> 'n vec *)
```

The function `loadvec` of type `string -> ? vec` returns a vector of some dimension, loaded from the given path. The third line should be ill-typed because the dimensions of `x` and `y` are probably different. (Even if `"file1"` and `"file2"` were the same path, the addition should be ill-typed because the file might change between the two loads.) Thus, the return type of `loadvec` should be different every time it is called (regardless of the specific values of the argument). We call such a return type *generative* because the function returns a value of a fresh type for each call.

The vector type with generative size information essentially corresponds to an existentially quantified sized type like $\exists$`n. n vec`. Our basic idea is to verify *only* the equality of dimensions by representing them as (only) generative phantom types. We implemented this idea in OCaml (partly using first-class modules for packages of types like $\exists$`n. n vec`) and carried out a realistic case study to demonstrate that it is mostly straightforward to write (or port) application programs by using our interface (see Section 4 for details).

3. Typing of BLAS and LAPACK functions

BLAS (Basic Linear Algebra Subprograms) [**?**] and LAPACK (Linear Algebra PACKage) [**?**] are the major linear algebra libraries for Fortran. To evaluate the effectiveness of our idea, we implemented a linear algebra library interface as a "more statically typed" wrapper of Lacaml, which is a BLAS and LAPACK binding in OCaml. Our interface is largely similar to Lacaml so that existing application programs can be easily ported. Here we explain several techniques required for typing the BLAS and LAPACK functions.

3.1 Transpose flags for matrices

In BLAS, `gemm` multiplies two general matrices:

```
val gemm : ?beta:num_type -> ?c:mat (*C*) ->
  ?transa:['N|'T|'C] -> ?alpha:num_type -> mat (*A*) ->
  ?transb:['N|'T|'C] -> mat (*B*) -> mat (*C*)
```

Basically, it executes $C := \alpha AB + \beta C$. The parameters `transa` and `transb` specify no transpose (`'N`), transpose (`'T`), or conjugate transpose (`'C`) of the matrices A and B. For example, if `transa='N` and `transb='T`, then `gemm` executes $C := \alpha AB^\top + \beta C$. Thus, the *types* (dimensions) of the matrices change depending on the *values* of the flags (the transpose of an m-by-n matrix is an n-by-m matrix). To implement this behavior, we give each transpose flag a function type that represents the change in types induced by that particular transposition, like:

[1] This term is used differently in some other papers.

```
type 'a trans (* = [ `N | `T | `C ] *)
val normal : (('m,'n) mat->('m,'n) mat) trans (* = `N *)
val trans  : (('m,'n) mat->('n,'m) mat) trans (* = `T *)
val conjtr : (('m,'n) mat->('n,'m) mat) trans (* = `C *)
val gemm : ?beta:num_type -> ?c:('m, 'n) mat (*C*) ->
  transa:(('am, 'ak) mat -> ('m, 'k) mat) trans ->
  ?alpha:num_type -> ('am, 'ak) mat (*A*) ->
  transb:(('bk, 'bn) mat -> ('k, 'n) mat) trans ->
  ('bk, 'bn) mat (*B*) -> ('m, 'n) mat (*C*)
```

3.2 Side flags for square matrix multiplication

The BLAS function `symm` multiplies a symmetric matrix A by a general matrix B:

```
val symm : ?side:[`L|`R] -> ?beta:num_type ->
  ?c:mat (*C*) -> ?alpha:num_type -> mat (*A*) ->
  mat (*B*) -> mat (*C*)
```

The parameter `side` specifies the "direction" of the multiplication: function `symm` executes $C := \alpha AB + \beta C$ if `side` is `` `L ``, and $C := \alpha BA + \beta C$ if it is `` `R ``. If B and C are m-by-n matrices, A is an m-by-m matrix in the former case and n-by-n in the latter case. We implemented these flags as follows:

```
type ('k, 'm, 'n) side (* = [ `L | `R ] *)
val left  : ('m, 'm, 'n) side (* = `L *)
val right : ('n, 'm, 'n) side (* = `R *)
```

The parameter `'k` in type `('k,'m,'n) side` corresponds to the dimension of the `'k`-by-`'k` symmetric matrix A, and the other parameters `'m` and `'n` correspond to the dimensions of the `'m`-by-`'n` general matrix B. When A is multiplied from the left by B (i.e., AB), `'k` is equal to `'m`; therefore, the type of the flag `left` is `('m,'m,'n) side`. Conversely, if A is right-multiplied by B (i.e., BA), `'k` is equal to `'n`. Thus, the flag `right` is given the type `('n,'m,'n) side`. By using this trick, we can type `symm` as:

```
val symm : side:('k,'m,'n) side -> ?beta:num_type ->
  ?c:('m,'n) mat (*C*) -> ?alpha:num_type ->
  ('k,'k) mat (*A*)-> ('m,'n) mat (*B*)-> ('m,'n) mat
```

The same trick can be applied to other square matrix multiplications as well.

3.3 Subtyping for discrete memory access

In Fortran, elements of a matrix are stored in column-major order in a flat, contiguous memory region. BLAS and LAPACK functions can take part of a matrix (like a row, a column, or a submatrix) and use it for computation without copying the elements, so they need to access the memory discretely in order to access the elements. However, some original functions of Lacaml do not support such discrete access. For compatibility and soundness, we need to prevent those functions from receiving (sub)matrices that require discrete accesses while allowing the converse (i.e., the other functions may receive contiguous matrices as well as discrete ones). We achieved this by extending the type definition of matrices by adding a third parameter for "contiguous or discrete" flags (in addition to the existing two parameters for dimensions):

```
type ('m, 'n, 'cnt_or_dsc) mat (* 'm-by-'n matrices *)
type cnt (* phantom *)
type dsc (* phantom *)
```

Then, formal arguments that may be either contiguous or discrete matrices are given type `('m,'n,'cnt_or_dsc) mat`, while the types of (formal) arguments that *must* be contiguous are specified in the form `('m,'n,cnt) mat`. In contrast, return values that may be either contiguous or discrete have type `('m,'n,dsc) mat`, while those that are known to be always contiguous are typed `('m,'n,'cnt_or_dsc) mat` so that they can be mixed with discrete matrices.

3.4 Dynamic checks remaining

Although many high-level operations provided by BLAS and LAPACK can be statically verified by using the scheme described above as far as equalities of dimensions are concerned, other operations still require dynamic checks for inequalities:

- `get` and `set` operations allow accesses to an element of a matrix by using the given indices, which must be less than the dimensions of the matrix.
- As mentioned above, BLAS and LAPACK functions can take a submatrix without copying it. Our original function `submat` returns such a submatrix for the dimensions given. This submatrix must be smaller than the original matrix.
- There are several high-level functions with specifications that essentially involve submatrices and inequalities of indices, such as `syevr` (for finding eigenvalues), `orgqr`, and `ormqr` (for QR factorization).
- For most LAPACK functions, the workspace for computation can be given as an argument. It must be larger than the minimum workspace required as determined by each function (and other arguments).

These inequalities are dynamically checked by our library interface. We gave such functions the suffix `_dyn`, like `get_dyn` and `set_dyn` (with the exception of the last case since almost *all* LAPACK functions can take the workspace as a parameter, with a size that must be checked dynamically).

4. Porting of OCaml-GPR

We implemented our static size checking scheme as a linear algebra library interface that we call SLAP (Sized Linear Algebra Package, `https://github.com/akabe/slap/`) on top of Lacaml (`https://bitbucket.org/mmottl/lacaml`). To evaluate its usability, we ported to it OCaml-GPR (`https://bitbucket.org/mmottl/gpr`), a practical machine learning library for Gaussian process regression, written using Lacaml by the same author. The ported library (SGPR) is available at `https://github.com/akabe/sgpr`.

Just to give a (very) rough feel for the library via a simple (but non-trivial) example, the type of a function that "calculates a covariance matrix of inputs given a kernel"

```
val calc_upper : Kernel.t -> Inputs.t -> mat
```

is augmented like

```
val calc_upper : ('D, _, _) Kernel.t ->
  ('D, 'n) Inputs.t -> ('n, 'n, 'cnt) mat
```

indicating that it takes `'n` vectors of dimension `'D` and returns an `'n`-by-`'n` contiguous matrix.

We classified the changes required for the porting into 19 categories. Due to lack of space, we outline only a few of them here (see `https://akabe.github.io/sgpr/changes.pdf` for more details). Most of the changes could be made mechanically, including

- replacing the syntax sugar `x.{i,j}` with calls to `get_dyn` or `set_dyn` functions (for index-based accesses to elements of matrices or vectors).
- rewriting the transpose flags from `` `N ``, `` `T ``, and `` `C `` to `normal`, `trans`, and `conjtr` (and similarly for side flags).
- removing dynamic size checking when the consistency of sizes is already ensured by (static) types.

- changing the types `vec` and `mat` on the right hand side of a type definition to `('n,'cd) vec` and `('m,'n,'cd) mat`, respectively. It was then necessary to add the type parameters `'m`, `'n`, and `'cd` on the left hand side. Theoretically, it suffices to give fresh type parameters to all `vec` and `mat` (whenever necessary, the type inference engine of OCaml unifies them automatically). In practice, however, doing so introduced too many type parameters in the OCaml-GPR library. We reduced the number by unifying parameters known to be equal.

Other changes had to be made manually. For instance, when a matrix operation is implemented by using low-level index-based accesses, its size constraints cannot be inferred statically (since they are checked only at runtime). It must therefore be type-annotated by hand (or, alternatively, the operation must be re-implemented using high-level functions). We encountered five such cases in OCaml-GPR. Other cases include generative phantom types that escaped their scopes and function types that depended on the values of arguments.

Overall, 18.4 % of the lines of the code required some changes, out of which (with some overlap) 15.4 % were mechanical and 3.6 % required a human brain.

5. Conclusions

Our proposed linear algebra library interface SLAP uses generative phantom types to statically ensure that most operations on matrices satisfy dimensional constraints. It is based on a simple idea—only the equality of sizes needs to be verified—and can be realized by using a fairly standard type and module system of ML. We implemented this interface on top of Lacaml and then ported OCaml-GPR to it. Most of the high-level matrix operations in the BLAS and LAPACK linear algebra libraries were successfully typed, and few non-trivial changes were required for the port.

We did not find any bug in Lacaml or OCaml-GPR, maybe because both libraries have already been well tested and debugged or carefully written in the first place. However, we conjecture that our version of the libraries are particularly useful when developing a new library or application program on top since they detect an error not only *earlier* (i.e., at compile time instead of runtime) but also *at higher level*; for instance, if the programmer misuses a function of SGPR, an error is reported at the caller site rather than somewhere deep inside the call stack from the function.

Interesting directions for future work include formalization of the idea of generative phantom types and extension of the static types to enable verification of inequalities (in addition to the present verification of equalities), just to name a few.

References

[1] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[2] Sa Cui, Kevin Donnelly, and Hongwei Xi. ATS: A language that combines programming with theorem proving. In Bernhard Gramlich, editor, *FroCoS*, volume 3717 of *Lecture Notes in Computer Science*, pages 310–320. Springer, 2005.

[3] Markus Mottl. OCaml-GPR – efficient Gaussian process regression in OCaml. `https://bitbucket.org/mmottl/gpr`.

[4] Markus Mottl and Christophe Troestler. LACAML – linear algebra for OCaml. `https://bitbucket.org/mmottl/lacaml`.

[5] NetLib. BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org/blas/`.

[6] NetLib. LAPACK – Linear Algebra PACKage. `http://www.netlib.org/lapack/`.

[7] Hongwei Xi. Dependent ML – an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.