Changes required to port OCaml-GPR library to Sized Linear Algebra Package interface

Akinori Abe Eijiro Sumii
Tohoku University

May 22, 2014

1 Introduction

This document elaborates on the changes required for Sized GPR (`https://github.com/akabe/sgpr`), a porting of OCaml-GPR (`https://bitbucket.org/mmottl/gpr`) version 1.1.3 from Lacaml (`https://bitbucket.org/mmottl/lacaml`) to our interface SLAP (Sized Linear Algebra Package, `https://github.com/akabe/slap`). See the paper `https://akabe.github.com/sgpr/paper.pdf` for details of SLAP.

We have developed a linear algebra library interface called "SLAP" 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). SLAP is implemented as a "more statically typed" wrapper of Lacaml, which does not statically ensure the consistency of sizes. To evaluate the usability of SLAP, we ported the OCaml-GPR library from Lacaml.

To investigate the kinds and numbers of changes required for the porting, we added uniquely-formed comments `(*! ... *)` on the changed lines in the SGPR source code. We classified them into 19 categories as follows:

- Mechanical changes
  1. Conversion from sizes to integers (S2I)
  2. Replacing of size constants (SC)
  3. Replacing of size operations (SOP)
  4. Conversion from integers to sizes (I2S)
  5. Replacing of index-based accesses (IDX)
  6. Replacing of flags (RF)
  7. Insertion of flags (IF)
  8. Using of subvectors and submatrices (SUB)
  9. Eta-conversion (ETA)
  10. Replacing of identifiers (RID)
  11. Removing of dynamic checks (RMDC)
  12. Insertion of type parameters (ITP)
- Manual changes
  12. Insertion of type annotations (ITA)
  13. Optional arguments to labeled arguments (O2L)
  14. Escaping generative phantom types (EGPT)
  15. Function types that depend on values of arguments (FT)
  16. Expression types that depend on values of free variables (ET)
  17. Fitting of signatures (FS)
  18. Default kernel size (DKS)

We next explain the kinds of changes made through simple examples.

2 Mechanical changes

Twelve of the required kinds of changes could be made mechanically (i.e., automatically). They accounted for most of the lines of code requiring a change (see Section 4 for details).

2.1 Conversion from sizes to integers (S2I)

The following code can be compiled in Lacaml but not in SLAP.

```
let n = Vec.dim x in
for i = 1 to n do ... done
```

The variable `n` is a size since `Vec.dim : ('n,_) vec -> 'n size`, while an integer is required between **to** and **do**. (A size is a value that has a singleton type `'n size` on natural numbers; i.e., evaluation of a term with type `'n size` *always* results in the natural number corresponding to `'n`.)

To write the code, we needed to write conversion from a size to an integer as follows:

```
let n = Vec.dim x in
for i = 1 to Slap.Size.to_int n (*! S2I *) do ... done
```

The label `S2I` was placed on lines requiring this kind of change.

2.2 Replacing of size constants (SC)

The following code results in the creation of a 1-by-n matrix `a`.

```
let a = Mat.create 1 n
```

It is ill-typed since the integer `1` is passed to a size argument of `Mat.create`. Thus, the integer constant had to be replaced with the corresponding size constant:

```
let a = Mat.create Slap.Size.one (*! SC *) n
```

or

```
module N = Slap.Size.Of_int_dyn(struct let value = 1 end) (*! SC *)
let a = Mat.create N.value (*! SC *) n
```

(The functor `Slap.Size.Of_int_dyn` returns a module `N` containing the size `N.value` that has the type `N.n size` with a generative phantom type `N.n` as a package of an existential quantified sized type like `∃n. n vec`.)

2.3 Replacing of size operations (SOP)

It was necessary to rewrite operations on integers with the corresponding operations on sizes when the operands were sizes. For example, if `m` and `n` are sizes,

```
let a = Mat.create (m + n) n
```

is ill-typed since the operator `+` requires integers as the left and right operands. To make this code well-typed, it was replaced with `Slap.Size.add`.

```
let a = Mat.create (Slap.Size.add m n) (*! SOP *) n
```

(see SLAP documentation for details of size operations).

2.4 Conversion from integers to sizes (I2S)

The following function `f` returns the squared norm of the product of a vector `x` and a randomly created matrix.

```
open Lacaml.D

let f x =
  let n = Random.int 100 in
```

Lacaml	SLAP
`'L`	`Slap.Common.left`
`'R`	`Slap.Common.right`
`'N`	`Slap.Common.normal`
`'T`	`Slap.Common.trans`
`'C`	`Slap.Common.conjtr`

Table 1: Correspondence between flags in Lacaml and SLAP

```
let a = Mat.random n (Vec.dim x) in
let y = gemv a x in
nrm2 y
```

The `n` parameter must be given type-level size information because it is passed to the size argument of `Mat.random`. We thus rewrote it as:

```
open Slap.D

let f x =
  let n = Random.int 100 in
  let module N = Slap.Size.Of_int_dyn(struct let value = n end) in (*! I2S *)
  let a = Mat.random N.value (*! I2S *) (Vec.dim x) in
  let y = gemv a x in
  nrm2 y
```

2.5 Replacing of index-based accesses (IDX)

The vector and matrix types (`vec` and `mat`) of Lacaml are implemented by using the OCaml module `Bigarray` to share numerical arrays between OCaml and Fortran. In Lacaml, the syntax sugar `x.{i,j}` for index-based accesses to elements of big arrays (i.e., vectors or matrices) can be used:

```
a.{i, j} <- x.{i + j}
```

In SLAP, `('n, _) vec` and `('m, 'n, _) mat` are abstract types: i.e., the right hand sides of the type definitions are hidden by signature. This means that the syntax sugar cannot be used since the typechecker does not know that matrices and vectors are implemented as big arrays. Hence, the `get_dyn` or `set_dyn` function was used instead of the syntax sugar:

```
Mat.set_dyn a i j (Vec.get_dyn x (i + j)) (*! IDX *)
```

2.6 Replacing of flags (RF)

In Lacaml, transpose flags and side flags for matrix multiplication:

```
trmm ~side:'R ~transa:'T ~a b
```

We redefined them to represent changes in matrix type, with the change depending on the flag's value. Therefore, they were replaced with identifiers of SLAP:

```
trmm ~side:Common.right ~transa:Common.trans (*! RF *) ~a b
```

Table 1 shows the correspondence between flags in Lacaml and SLAP.

2.7 Insertion of flags (IF)

In Lacaml, the transpose and side flag arguments are optional, i.e., they can be omitted. When the arguments are omitted, the default values are passed. For example,

```
trmm ~a b
```

is the same as:

```
trmm ~side:'L ~transa:'N ~a b
```

In SLAP, the transpose and side flag arguments are implemented as labeled arguments (which cannot be omitted) because they represent the constraints on matrix type. This means that we had to explicitly provide the default value (`Slap.Common.normal` for the transpose flags and `Slap.Common.left` for the side flags):

```
trmm ~side:Common.left ~transa:Common.normal (*! IF *) ~a b
```

2.8 Using of subvectors and submatrices (SUB)

All BLAS and LAPACK functions support operation on subvectors or submatrices. For instance, the following code copies the `m`-by-`n` submatrix in a matrix `a`, in which element (i, j) corresponds to the $(i + ar - 1, j + ac - 1)$ element of `a`.

```
lacpy ~m ~n ~ar ~ac a
```

Since our idea is that only size equality of sizes is ensured statically, whether the function call is safe, i.e., the submatrix is *smaller* than `a`, cannot be verified statically. However, since adding dynamic checks to all BLAS and LAPACK functions is undesirable because submatrix designation is auxiliary and not essential to those functions, we defined separate functions to return a submatrix (or a subvector) of a given matrix (or vector). That is, such operations were replaced with `Slap.Vec.subvec_dyn` or `Slap.Mat.submat_dyn`, such as:

```
lacpy (Mat.submat_dyn m n ~ar ~ac a)
```

2.9 Eta-conversion (ETA)

Let `f` be a function that accepts two vectors that may have different dimensions and that returns `unit`.

```
let f x y = ...      (* f : ('m, 'cd1) vec -> ('n, 'cd2) vec -> unit *)
let g = f Vec.empty  (* g : ('_n, '_cd2) vec -> unit *)
```

The function `g` created by partial application is not polymorphic due to value restriction[1]. To recover the lost polymorphism, eta-conversion (insertion of arguments) was required:

```
let f x y = ...                     (* f : ('m, 'cd1) vec -> ('n, 'cd2) vec -> unit *)
let g y = f Vec.empty y (*! ETA *)  (* g : ('n, 'cd2) vec -> unit *)
```

2.10 Replacing of identifiers (RID)

Several identifiers needed to be replaced; e.g.,

```
open Lacaml.D
```

was replaced with

```
open Slap.D (*! RID *)
```

2.11 Removing of dynamic checks (RMDC)

The following dynamic check is not needed in SLAP because the type `dot` statically ensures the equality of the dimensions of two vectors `x` and `y`.

```
let f x y =
  if Vec.dim x <> Vec.dim y then invalid_arg "error!";
  dot x y (* dot : ('n, _) vec -> ('n, _) vec -> float *)
```

Thus, this code was rewritten as:

[1] A type parameter like `'_a` can be instantiated *only once*.

```
let f x y =
  (* if Vec.dim x <> Vec.dim y then invalid_arg "error!"; *) (*! RMDC *)
  dot x y
```

(We commented out such dynamic checks iso that we could determine the number of lines containing this kind of change.)

2.12 Insertion of type parameters (ITP)

We changed the types `vec` and `mat` on the right hand side of a type definition to `('n, 'cd) vec` and `('m, 'n, 'cd) mat`, respectively. Then the type parameters `'m`, `'n`, and `'cd` must also be added to the left hand side. Theoretically, it suffices to give fresh parameters to all `vec` and `mat`. For instance,

```
module M : sig
  type t
  val f : int -> t
end = struct
  type t = {
    n : int;
    id : mat;
  }
  let f n =
    let id = Mat.identity n in
    { n; id; }
end
```

was rewritten as:

```
module M : sig
  type ('a, 'b, 'c, 'd) t
  val f : 'a size -> ('a, 'a, 'a, _) t
end = struct
  type ('a, 'b, 'c, 'd) t = {
    n : 'a size;
    id : ('b, 'c, 'd) mat;
  }
  let f n =
    let id = Mat.identity n in
    { n; id; }
end
```

In the latter code, constraints of equality of sizes are unified automatically by the OCaml type inference engine. In practice, however, doing so introduces too many parameters in the OCaml-GPR library. We reduced the number by unifying type parameters that are known to be equal:

```
module M : sig
  type ('n, 'cnt_or_dsc) t (*! ITP *)
  val f : 'n size -> ('n, 'cnt) t (*! ITP *)
end = struct
  type ('n, 'cnt_or_dsc) t = { (*! ITP *)
    n : 'n size; (*! ITP *)
    id : ('n, 'n, 'cnt_or_dsc) mat; (*! ITP *)
  }
  let f n =
    let id = Mat.identity n in
    { n; id; }
end
```

3 Manual changes

Seven of the require kinds of changes had to be made manually. To make a finer-grained distinction in each kind of changes, we gave them reference numbers such as $[n]$.

3.1 Insertion of type annotations (ITA)

When a matrix operation is implemented by low-level index-based accesses, its size constraints cannot be inferred statically (since they are checked only at runtime): For example, consider the function `axby`, which calculates $\alpha\boldsymbol{x} + \beta\boldsymbol{y}$ with scalar values α and, β, and vectors $\boldsymbol{x}$ and $\boldsymbol{y}$:

```
open Slap.D

let axby alpha x beta y =
  let n = Vec.dim x in
  let z = Vec.create n in
  for i = 1 to Slap.Size.to_int n do
    let p = alpha *. (Vec.get_dyn x i) +. beta *. (Vec.get_dyn y i) in
    Vec.set_dyn z i p
  done;
  z
```

The dimensions of vectors `x` and `y` must be the same, but OCaml infers that they may be different:

```
val axby : float -> ('n, _) vec -> float -> ('m, _) vec -> ('n, _) vec
```

There are two ways to solve this problem. One is to type-annotate `axby` by hand:

```
let axby alpha (x : ('n, _) vec) beta (y : ('n, _) vec) =
  ...
```

The other way is to use high-level operations such as `scal` and `axpy` instead of low-level operations such as `get_dyn` and `set_dyn`:

```
let axby alpha x beta y =
  let z = copy y in (* z = y *)
  scal beta z;      (* z := beta * z *)
  axpy ~alpha ~x y; (* z := alpha * x + z *)
  z
```

We did not use the second way because we rewrote the OCaml-GPR code to make it as simple as possible. We encountered five such functions in OCaml-GPR:

- ITA[1]: `Gpr.Cov_se_iso.Eval.Inputs.weighted_eval`
- ITA[2]: `Gpr.Gpr_utils.log_det`
- ITA[3]: `Gpr.Gpr_utils.check_sparse_row_mat_sane`
- ITA[4]: `Gpr.Gpr_utils.check_sparse_col_mat_sane`
- ITA[5]: `Gpr.Gpr_utils.check_sparse_vec_sane`.

3.2 Optional arguments to labeled arguments (O2L)

The following function `f` accepts an integer and a unit:

```
open Lacaml.D

let f ?n () =
  let n' = match n with
           | None -> 10
           | Some n -> n in
  Vec.make n' 1.0
```

If the first optional argument `?n` is omitted, `10` is implicitly used as its default value.

Seemingly, the code above can be rewritten as:

```
open Slap.D

let f ?n () =
  let n' = match n with
           | None -> Slap.Size.ten (* Slap.Size.ten : ten size *)
           | Some n -> n in
  Vec.make n' 1.0
```

We expect that `?n:'n size -> unit -> ('n, _) vec` is the type for `f` because it works for all `n`, but

```
val f : ?n:ten size -> unit -> (ten, _) vec
```

is inferred. Only `Slap.Size.ten` can be passed to `?n`; other size values cannot be passed.

We thus replaced the optional argument with a (labeled) argument:

```
open Slap.D

let default_n = Slap.Size.ten (* the default value of n *)

let f ~n () = Vec.make n 1.0
```

In this case, the first argument can not be omitted. The default value `default_n` is passed explicitly.

We applied this approach to

- O2L[1]: the optional argument `n_rand_inducing` and
- O2L[2]: the optional argument `kernel`

of `Fitc_gp.Deriv_common.Optim.get_kernel_inducing`. In addition, we defined functions to compute their default values.

3.3 Escaping generative phantom types (EGPT)

Consider a function that converts an array of strings into a vector:

```
open Lacaml.D

let f a =
  Vec.init (Array.length a) (fun i -> float_of_string a.(i-1))

let main () =
  let a = [| "1"; "2"; "3" |] in
  let v = f a in
  Format.printf "%a\n" pp_vec v
```

This program can not be compiled in SLAP because `Vec.init` expects `'n size` as the first argument, and `Array.length` returns an integer. Therefore, the integer must be converted into a size value with `Size.Of_int_dyn`. The following code seems intuitively correct:

```
open Slap.D

let f a =
  let module N = Slap.Size.Of_int_dyn(struct let value = Array.length a end) in
  Vec.init N.value (fun i -> float_of_string a.(i-1))
```

However, OCaml cannot compile this code because the generative phantom type `N.n` escapes its scope.

There are three ways to handle this in SLAP. One is to insert the argument `n` for the size of the array, and remove the generative phantom type from the function:

```
open Slap.D

let f n a =
  if Slap.Size.to_int n <> Array.length a then invalid_arg "error";
  Vec.init n (fun i -> float_of_string a.(i-1))
```

```
let main () =
  let a = [| "1"; "2"; "3" |] in
  let module N = Slap.Size.Of_int_dyn(struct let value = Array.length a end) in
  let v = f N.value a in
  Format.printf "%a\n" pp_vec v
```

In this case, whether `n` is equal to the length of `a` should be *dynamically* checked.

Another way is to define a functor that returns a module containing the generative phantom type:

```
open Slap.D

module F (A : sig val value : string array end) : VEC = struct
  module N = Slap.Size.Of_int_dyn(struct let value = Array.length A.value end)
  type n = N.n (* the generative phantom type *)
  let value = (* val value : (n, _) vec *)
    Vec.init N.value (fun i -> float_of_string A.value.(i-1))
end

let main () =
  let a = [| "1"; "2"; "3" |] in
  let module V = F(struct let value = a end) in
  Format.printf "%a\n" pp_vec V.value
```

where signature `VEC` is defined as:

```
module type VEC = sig
  type n (* a generative phantom type *)
  val value : (n, _) vec
end
```

And the third way is to define `f : string array -> (?, _) vec` by using a first-class module instead of a functor:

```
open Slap.D

let f a =
  let module N = Slap.Size.Of_int_dyn(struct let value = Array.length a end) in
  let module V = struct
      type n = N.n
      let value = Vec.init N.value (fun i -> float_of_string a.(i-1))
    end in
  (module V : VEC)

let main () =
  let a = [| "1"; "2"; "3" |] in
  let module V = (val (f a) : VEC) in
  Format.printf "%a\n" pp_vec V.value
```

However, a type annotation of a module is required with the third way. We thus used the first way temporarily in SGPR as follows:

- EGPT[1]: Two `'n size` arguments were added to `Gpr.Cov_*.Eval.Inputs.create`.
- EGPT[2]: The labeled argument `n_hypers : 'n size` was added to `Gpr.Fitc_gp.Deriv_common.Optim.get_hypers_val`.
- EGPT[3]: Two `'n size` arguments were added to `read_training_samples` in app/ocaml_gpr.ml.

We plan to consider rewriting SGPR by using the third way because the first way requires dynamic checks.

3.4 Function types that depend on values of arguments (FT)

The following function `f` changes the dimension of a returned vector depending on the value of the argument `b`.

```
open Lacaml.D

let f b n = Vec.make0 (if b then n else n + 1)
```

In SLAP, the expression `(if b then n else Slap.Size.succ n)` cannot be compiled because the terms `n` and `Slap.Size.succ n` have different types (the former is `'n size` and the latter is `'n s size`). To obtain a function type that depends on the value of `b`, we rewrote the above code as:

```
open Slap.D

module M : sig
  type ('n, 'm) t
  val b_tru : ('n, 'n) t
  val b_fls : ('n, 'n s) t
  val f : ('n, 'm) t -> 'n size -> ('m, _) vec
end = struct
  type ('n, 'm) t = 'n size -> 'm size
  let b_tru n = n
  let b_fls n = Slap.Size.succ n
  let f b n = Vec.make0 (b n)
end
```

The `b_tru` and `b_fls` parameters are passed to the first argument instead of **`true`** and **`false`**, respectively. We found three such cases in OCaml-GPR:

- FT[1]: The functions `SGD.create` and `SMD.create` in the module `Fitc_gp.Deriv_common.Optim` have types dependent on the value of the argument `learn_sigma2 : bool`. We define the constants `learn_sigma` and `not_learn_sigma2` to use for the argument instead of **`true`** and **`false`**.
- FT[2]: The constraint on type parameters for the record type `('D,'d,'m) Cov_se_fat.Params.params` changes depending on the value in its field `tproj`. This is not a function, but the required technique is very similar.
- FT[3]: The same technique was applied to the argument `n` of the function `read_test_samples` (app/ocaml_gpr.ml).

3.5 Expression types that depend on values of free variables (ET)

The function `f` returns the squared norm of the product of a vector `x` and a randomly created matrix:

```
open Lacaml.D

let f b x =
  let n = Vec.dim x in
  let m = if b then n + 1 else n in
  let a = Mat.random m n in
  let y = gemv a x in
  nrm2 y
```

This code is rewritten as:

```
open Slap.D

let f b x =  (* f : bool -> ('n, _) vec -> float *)
  let n = Vec.dim x in
  let g m = (*! ET *)  (* g : 'm size -> float *)
    let a = Mat.random m n in
    let y = gemv a x in
    nrm2 y
  in (*! ET *)
  if b then g (Slap.Size.succ n) (*! SOP *) else g n (*! ET *)
```

This change is of a kind similar to `FT`, but the interface of a function is not affected by this kind of change.

We applied this kind of changes to two expressions in app/ocaml_gpr.ml:

- ET[1]: The type of the expression that returns `d` and `tproj` in the function `train` depends on the value of the argument `args.dim_red` of the function.
- ET[2]: The type of the expression that returns `inputs` in the function `test` depends on the length of an array `samples` loaded from a file.

3.6 Fitting of signatures (FS)

The above-mentioned changes (including mechanical changes) are relatively local. In contrast, the FS and DKS changes discussed here are widespread and somewhat ad hoc. Before discussing FS, we explain several important modules in the OCaml-GPR library and their relationships. OCaml-GPR supports kernel-based[2] fitting of (nonlinear) functions. Kernel functions for the fitting are defined as modules, and OCaml-GPR provides five predefined kernel functions (see OCaml-GPR documentation for details):

- `Gpr.Cov_const`
- `Gpr.Cov_lin_one`
- `Gpr.Cov_lin_ard`
- `Gpr.Cov_se_iso`
- `Gpr.Cov_se_fat`

These kernel modules cannot be used for fitting directly. To make a module suitable for fitting, a programmer needs to pass it to the functor `Gpr.Fitc_gp.Make` or `Gpr.Fitc_gp.Make_deriv`.

3.6.1 Types of kernels

Each kernel function accepts two input vectors and parameters such as scalar values, vectors, and matrices. For example, a constant kernel `Cov_const` calculates $k(\boldsymbol{x}, \boldsymbol{y}) = 1/\theta^2$ with a real constant θ, and the kernel function of `Cov_lin_ard` needs a vector of automatic relevance determination (ARD) parameters for computing the covariance of two input vectors with ARD.

The parameters are embedded in the data structure of each kernel. Thus, if the kernel requires vectors or matrices as parameters, their type parameters are added on the left hand side of the type definition of the kernel type (cf. Section 2.12). The number of type parameters depends on the kernel type because the number and the types of parameters differ from each other (e.g. the kernel type has no type parameters in `Cov_const` but two in `Cov_lin_ard`). However, all kernel modules need to be given the same signature in order to pass them to the functor `Gpr.Fitc_gp.Make` or `Gpr.Fitc_gp.Make_deriv`. Therefore, it is necessary to give all kernels the same type.

We fitted each kernel type to the type `('D, 'd, 'm) t` of the kernel `Cov_se_fat`, which has the most type parameters. It has three type parameters, but some or all of them are phantom for some modules (e.g., all of them are phantom for `Cov_const` while only `'m` is phantom for `Cov_lin_ard`.)

3.6.2 Generalization of kernel types

The kernel of `Cov_se_fat` contains dimensionality reduction from `'D`-dimensional space to `'d`-dimensional, so `'D` often differs from `'d`. In contrast, in other kernels, `'D` is always the same as `'d` because there is no input dimension reduction. Here we explain the safe generalization of the latter type `('d,'d,'m) t` to the former type `('D,'d,'m) t` through the following simple example of a signature for modules for conversion of input vectors. (Type parameters for subtyping of vectors and matrices are omitted.)

[2] A *kernel method* (or *kernel trick*) is an approach to extending a linear algorithm on the basis of inner product of the vectors to a nonlinear one by using *kernel functions*.

```
module type S = sig
  type ('D, 'd) t
  val calc_vec : ('D, 'd) t -> 'D vec -> 'd vec
  val calc_mat : ('D, 'd) t -> ('D, 'n) mat -> ('d, 'n) mat
end
```

The `calc_vec` converts a `'D`-dimensional input vector into a `'d`-dimensional output vector. Similarly, `calc_mat` converts `'n` (column) vectors. (`('D, 'n) mat` is a type of array of `'n` vectors with dimensions of `'D`.)

It is straightforward to implement conversion including dimensionality reduction (corresponding to `Cov_se_fat`). For example, the module `M0` calculates $2\boldsymbol{P}^\top\boldsymbol{x}$ with a `'D`-by-`'d` matrix $\boldsymbol{P}$ for dimensionality reduction and a `'D`-dimensional input vector $\boldsymbol{x}$:

```
module M0 : S with type ('D, 'd) t = ('D, 'd) mat = struct
  type ('D, 'd) t = ('D, 'd) mat (* a matrix for dimensionality reduction *)

  let calc_vec p x =
    let y = gemv ~trans:Common.trans p x (* y : 'd vec *)
    scal 2.0 y;
    y

  let calc_mat p x =
    let y = gemm ~transa:Common.trans p ~transb:Common.normal x (* y : ('d, 'n) mat *)
    Mat.scal 2.0 y;
    y
end
```

With a little ingenuity, we define a module for conversion that does not reduce the dimensions:

```
module M1 : sig
  include S
  val create : 'd size -> ('d, 'd) t (* constructor of ('D, 'd) t *)
end = struct
  type ('D, 'd) t = ('D, 'd) mat
  let create d = Mat.identity d (* return a d-by-d identity matrix *)

  (* The implementation of calc_vec and calc_mat is the same as M0. *)
end
```

Note that the return type of `create`, the constructor of the abstract type `('D, 'd) t`, is `('d, 'd) t`, not `('D, 'd) t`. The value that can be passed to `calc_vec` or `calc_mat` is made only by `create`. Thus, we can consider that a practical type for the first argument of `calc_vec` and `calc_mat` is `('d, 'd) t`.

This approach is inefficient because a huge identity matrix must be multiplied when `d` is large. As a more efficient approach, we used identity functions instead of the identity matrix:

```
module M2 : sig
  include S
  val create : unit -> ('d, 'd) t
end = struct
  type ('D, 'd) t = {id : 'n . ('D vec -> 'd vec) * (('D, 'n) mat -> ('d, 'n) mat)}
  let create () = {id = (fun x -> x), (fun x -> x)}

  let calc_vec {id = (id, _)} x =
    let y = id x in (* y : 'd vec *)
    scal 2 y;
    y

  let calc_mat {id = (_, id)} x =
    let y = id x in (* y : ('d, 'n) mat *)
    Mat.scal 2 y;
    y
end
```

First-class polymorphism (extension of OCaml) was used for the definition of `('D, 'd) M2.t`. There is also a similar solution using only ML types:

```
module type S3 = sig
  type ('D, 'd, 'n) t (* the type parameter 'n is added. *)
  val calc_vec : ('D, 'd, _) t -> 'D vec -> 'd vec
  val calc_mat : ('D, 'd, 'n) t -> ('D, 'n) mat -> ('d, 'n) mat
end

module M3 : sig
  include S3
  val create : unit -> ('d, 'd, 'n) t
end = struct
  type ('D, 'd, 'n) t = ('D vec -> 'd vec) * (('D, 'n) mat -> ('d, 'n) mat)
  let create () = (fun x -> x), (fun x -> x)
  let calc_vec (id, _) x = id x
  let calc_mat (_, id) x = id x
end
```

In the above code, the type parameter `'n` is inserted on the left side hand of the type definition. The implementation of `M3` is a little simpler than that of `M2`, but the behavior of `M3.calc_mat` is not the same as those of `M1.calc_mat` and `M2.calc_mat`; e.g., the function types for

```
let f1 t x y = (M1.calc_mat t x), (M1.calc_mat t y)
let f2 t x y = (M2.calc_mat t x), (M2.calc_mat t y)
let f3 t x y = (M3.calc_mat t x), (M3.calc_mat t y)
```

are

```
val f1 : ('D, 'd) M1.t     -> ('D, 'm) mat -> ('D, 'n) mat -> ('d, 'm) mat * ('d, 'n) mat
val f2 : ('D, 'd) M2.t     -> ('D, 'm) mat -> ('D, 'n) mat -> ('d, 'm) mat * ('d, 'n) mat
val f3 : ('D, 'd, 'n) M3.t -> ('D, 'n) mat -> ('D, 'n) mat -> ('d, 'n) mat * ('d, 'n) mat
```

The dimensions of the returned matrices are different for `f1` and `f2` but the same for `f3`. In other words, the polymorphism is restricted by using `M3.calc_mat`. Therefore, we used `M2`.

3.7 Default kernel size (DKS)

Each `Cov_*` module provides the function `Eval.Inputs.create_default_kernel_params` to generate the default parameter for its kernel function. The return type of the function is `('D,('D,ten) min,'m) Kernel.params` in `Cov_se_fat` but `('D,'D,'m) Kernel.params` in other `Cov_*` modules. To give `Cov_se_fat` and other `Cov_*` modules the same signature, we defined original type `'D default_kernel_size` for the second type parameter of `Kernel.params`. With this change, the function type for the default kernel parameter became:

```
val create_default_kernel_params : ... -> ('D, 'D default_kernel_size, 'm) Kernel.params
```

The original type is defined as

```
type 'D default_kernel_size = ('D, Slap.Size.ten) Slap.Size.min
```

in `Cov_se_fat` and

```
type 'D default_kernel_size = 'D
```

in other `Cov_*` modules.

4 Results

Table 2 shows the number of lines requiring changes that could be made mechanically and the corresponding percentages. The "Total" in the rightmost column is the number of lines requiring at least one changes, which is not equal to the simple summation of all changes because a line may have required more than one change. The number of lines for ITP was large (6.17 %) because OCaml-GPR is constructed of several large modules, so the definitions of signatures are long. Most of the ITP changes have been

Table 2: Number and percentage of lines requiring changes that could be made mechanically

	S2I	SC	SOP	I2S	IDX	RF	IF	SUB	ETA	RID	RMDC	ITP	**Total**
lib/block_diag.mli	0	0	0	0	0	0	0	0	0	1	0	5	6
lib/block_diag.ml	1	0	0	0	0	0	0	0	0	1	6	1	9
lib/cov_const.mli	0	0	0	0	0	0	0	0	0	0	0	5	5
lib/cov_const.ml	2	0	0	0	1	0	0	0	0	2	0	9	14
lib/cov_lin_one.mli	0	0	0	0	0	0	0	0	0	1	0	5	6
lib/cov_lin_one.ml	0	0	0	0	1	4	2	0	0	2	0	9	17
lib/cov_lin_ard.mli	0	0	0	0	0	0	0	0	0	1	0	5	6
lib/cov_lin_ard.ml	7	0	0	0	10	5	2	0	0	2	0	9	32
lib/cov_se_iso.mli	0	0	0	0	0	0	0	0	0	1	0	5	6
lib/cov_se_iso.ml	18	4	0	0	31	0	0	0	0	2	2	14	71
lib/cov_se_fat.mli	0	0	0	0	0	0	0	0	0	1	0	10	11
lib/cov_se_fat.ml	43	9	1	0	87	2	2	0	0	2	8	23	174
lib/fitc_gp.mli	0	0	0	0	0	0	0	0	0	0	0	0	0
lib/fitc_gp.ml	81	3	3	0	63	19	26	14	34	3	15	69	298
lib/interfaces.ml	0	0	0	0	0	0	0	0	0	1	0	196	197
lib/gpr_utils.ml	10	0	0	0	13	0	2	0	0	4	17	1	46
app/ocaml_gpr.ml	13	0	2	4	10	0	0	0	0	3	0	8	35
Total	175	16	6	4	216	30	34	14	34	27	48	374	933
Percentage	2.89	0.26	0.10	0.07	3.56	0.49	0.56	0.23	0.56	0.45	0.79	6.17	15.39

Table 3: Number and percentage of lines requiring changes that had to be made manual

	ITA	EGPT	O2L	FT	ET	DKS	FS	**Total**
lib/block_diag.mli	0	0	0	0	0	0	0	0
lib/block_diag.ml	0	0	0	0	0	0	0	0
lib/cov_const.mli	0	0	0	0	0	0	2	2
lib/cov_const.ml	0	1	0	0	0	1	8	10
lib/cov_lin_one.mli	0	0	0	0	0	0	2	2
lib/cov_lin_one.ml	0	0	0	0	0	1	12	13
lib/cov_lin_ard.mli	0	0	0	0	0	0	2	2
lib/cov_lin_ard.ml	0	0	0	0	0	1	8	9
lib/cov_se_iso.mli	0	0	0	0	0	0	2	2
lib/cov_se_iso.ml	2	0	0	0	0	1	6	9
lib/cov_se_fat.mli	0	0	0	4	0	0	0	4
lib/cov_se_fat.ml	0	0	0	28	0	1	1	30
lib/fitc_gp.mli	0	0	0	0	0	0	0	0
lib/fitc_gp.ml	0	28	31	16	0	0	0	68
lib/interfaces.ml	0	11	7	5	0	3	0	26
lib/gpr_utils.ml	6	0	0	0	0	0	1	7
app/ocaml_gpr.ml	0	17	0	6	16	0	0	35
Total	8	57	38	59	16	8	44	219
Percentage	0.13	0.94	0.63	0.97	0.26	0.13	0.73	3.61

made in lib/interfaces.ml, which defines all signatures used in OCaml-GPR. The number for IDX was the second largest (3.56 %) because index-based accesses are frequently used in OCaml-GPR. They are also used when they could be replaced with high-level matrix operations such as `map`, etc. It should thus be possible to reduce their number.

Table 3 shows the number and percentages of lines for which the required changes had to be made manually, and Table 4 shows the total amounts for all changes. Overall, 18.39 % of the lines required at least one change, out of which 15.42 % were mechanical and 3.61 % were manual. From these results, we conjecture in general that the number of non-trivial changes required for a user program of SLAP is small, but further investigation is necessary.

Table 4: Number and percentage of all lines requiring changes

	Lines	Mechanical	Manual	**Total**
lib/block_diag.mli	56	6	0	6
lib/block_diag.ml	58	9	0	9
lib/cov_const.mli	52	5	2	6
lib/cov_const.ml	141	14	10	16
lib/cov_lin_one.mli	56	6	2	7
lib/cov_lin_one.ml	149	17	13	26
lib/cov_lin_ard.mli	56	6	2	7
lib/cov_lin_ard.ml	188	32	9	39
lib/cov_se_iso.mli	58	6	2	7
lib/cov_se_iso.ml	343	71	9	78
lib/cov_se_fat.mli	105	11	4	15
lib/cov_se_fat.ml	680	174	30	199
lib/fitc_gp.mli	151	0	0	0
lib/fitc_gp.ml	2294	298	68	364
lib/interfaces.ml	1008	197	26	215
lib/gpr_utils.ml	229	46	7	53
app/ocaml_gpr.ml	440	35	35	66
Total	6064	933	219	1113
Percentage	100.00	15.39	3.61	18.35