OCaml 2014 in Sweden
(presented at ML 2014)

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

Akinori Abe   Eijiro Sumii

Graduate School of Information Sciences
Tohoku University (Japan)

September 4

Background: Static size checking of arrays, lists, etc.

- Advanced type systems (e.g., dependent types) have been proposed:
  - Dependent ML [Xi and Pfenning, 1999], ATS [Xi]
  - sized type [Chin and Khoo, 2001]

  However, they generally require **non-trivial changes** to
  - existing languages and
  - application programs,

  or
- **tricky** type-level programming.

Our contribution

A linear algebra library interface
with static size checking
by using generative phantom types

Features

- Only using fairly standard ML types and a few OCaml extensions
  - Indeed, we implemented the interface in OCaml.
- Static sizes checking for most high-level matrix operations
  - Certain low-level operations need dynamic checks.
    (e.g., index-based accesses)
- Easy to port existing application programs
  - Most of required changes can be made mechanically.

Outline

1. **Our idea**
   - Generative phantom types
   - Typing of BLAS and LAPACK functions
2. **Porting of OCaml-GPR**
   - Required changes
   - Percentages of required changes

Outline

1. **Our idea**
   - Generative phantom types
   - Typing of BLAS and LAPACK functions

2. **Porting of OCaml-GPR**
   - Required changes
   - Percentages of required changes

Types of vectors and matrices

```
type 'n vec         (* the type of 'n-dimensional vectors *)
type ('m, 'n) mat (* the type of 'm-by-'n matrices *)
type 'n size        (* the singleton type on natural numbers *)
```

phantom type params

- `'m` and `'n` above

phantom types

- Types that `'m` and `'n` are instantiated with
  (They often has no constructor.)

Types of vectors and matrices

```
type 'n vec        (* the type of 'n-dimensional vectors *)
type ('m, 'n) mat (* the type of 'm-by-'n matrices *)
type 'n size       (* the singleton type on natural numbers *)
```

phantom type params

- `'m` and `'n` above

phantom types

- Types that `'m` and `'n` are instantiated with
  (They often has no constructor.)

How do we represent dimensions as types?

- They are probably unknown until runtime.

Outline

1. **Our idea**
   - Generative phantom types
   - Typing of BLAS and LAPACK functions
2. **Porting of OCaml-GPR**
   - Required changes
   - Percentages of required changes

A simple example

```
val loadvec : string → ? vec (* load a vector from a file *)
val add : 'n vec → 'n vec → 'n vec (* add two vectors *)
```

A simple example

```
val loadvec : string → ? vec (* load a vector from a file *)
val add : 'n vec → 'n vec → 'n vec (* add two vectors *)
```

- Addition of two vectors loaded from **different** files:

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* This should be ill-typed, i.e., ?¹ ≠ ?². *)
```

A simple example

```
val loadvec : string → ? vec (* load a vector from a file *)
val add : 'n vec → 'n vec → 'n vec (* add two vectors *)
```

- Addition of two vectors loaded from **different** files:

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* This should be ill-typed, i.e., ?¹ ≠ ?². *)
```

- Addition of two vectors loaded from the **same** file:

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file1" in
add x y (* This should be ill-typed, i.e., ?¹ ≠ ?². *)
```

The file might be changed between the two loads.

A simple example

```
val loadvec : string → ? vec (* load a vector from a file *)
val add : 'n vec → 'n vec → 'n vec (* add two vectors *)
```

- Addition of two vectors loaded from **different** files:

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* This should be ill-typed, i.e., ?¹ ≠ ?². *)
```

- Addition of two vectors loaded from the **same** file:

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file1" in
add x y (* This should be ill-typed, i.e., ?¹ ≠ ?². *)
```

The file might be changed between the two loads.

Thus, the return type of `loadvec` should be different every time it is called.

Generative phantom types

```
val loadvec : string → ? vec (* load a vector from a file *)
```

"?" is a generative phantom type:

- The function returns a value of a fresh type for each call.
- This corresponds to an existentially quantified sized type like $\exists$`n. n vec` (not a type-level natural number).
- We implemented this idea in OCaml (partly using first-class modules).

Our idea

Our idea

We represent dimensions by using (only) generative phantom types:

- Typing is simplified.
- **Only** equalities of dimensions are guaranteed.
- Practical programs can be verified!
    - We show the usability by porting an existing application program.

Outline

Typing of BLAS and LAPACK functions

BLAS & LAPACK

- The major linear algebra libraries for Fortran

Lacaml

- A BLAS & LAPACK binding in OCaml

Typing of BLAS and LAPACK functions

BLAS & LAPACK

- The major linear algebra libraries for Fortran

Lacaml

- A BLAS & LAPACK binding in OCaml

We typed Lacaml (BLAS & LAPACK) functions.

- Many high-level matrix operations are successfully typed!
- Certain functions need dynamic checks:
  - Index-based accesses (`get`, `set`)
  - Our original functions `subvec` to return a subvector and `submat` to return a submatrix
  - Several LAPACK functions (`syevr`, `orgqr`, `ormqr`)
  - Workspaces of LAPACK functions

Typing of BLAS and LAPACK functions

BLAS & LAPACK

- The major linear algebra libraries for Fortran

Lacaml

- A BLAS & LAPACK binding in OCaml

We typed Lacaml (BLAS & LAPACK) functions.

- Many high-level matrix operations are successfully typed!
- Certain functions need dynamic checks:
  - Index-based accesses (`get_dyn`, `set_dyn`)
  - Our original functions `subvec_dyn` to return a subvector and `submat_dyn` to return a submatrix
  - Several LAPACK functions (`syevr_dyn`, `orgqr_dyn`, `ormqr_dyn`)
  - Workspaces of LAPACK functions

Example of the typing (1)

- **`dot ~x y`** computes inner product of x and y.

```
val dot : x:vec → vec → float
```

↓

```
val dot : x:'n vec → 'n vec → float
```

Example of the typing (1)

- **dot ~x y** computes inner product of x and y.

```
val dot : x:vec → vec → float
```

↓

```
val dot : x:'n vec → 'n vec → float
```

- **axpy ?alpha ~x y** computes y := alpha * x + y.

```
val axpy : ?alpha:float → x:vec → vec → unit
```

↓

```
val axpy : ?alpha:float → x:'n vec → 'n vec → unit
```

Example of the typing (2) — Transpose flags

```
val gemm : ?alpha:num_type → ?beta:num_type → ?c:mat (* C *) →
  ?transa:[ `N | `T | `C ] → mat (* A *) →
  ?transb:[ `N | `T | `C ] → mat (* B *) → mat (* C *)
```

`transa` and `transb` specify no transpose (`` `N ``), transpose (`` `T ``) and conjugate transpose (`` `C ``) of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$:

- E.g., $\boldsymbol{C} := \alpha \boldsymbol{A}\boldsymbol{B}^\top + \beta \boldsymbol{C}$ when `transa`=`` `N `` and `transb`=`` `T ``.

Example of the typing (2) — Transpose flags

```
val gemm : ?alpha:num_type → ?beta:num_type → ?c:mat (* C *) →
  ?transa:[ `N | `T | `C ] → mat (* A *) →
  ?transb:[ `N | `T | `C ] → mat (* B *) → mat (* C *)
```

`transa` and `transb` specify no transpose (`` `N``), transpose (`` `T``) and conjugate transpose (`` `C``) of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$:

- E.g., $\boldsymbol{C} := \alpha \boldsymbol{A}\boldsymbol{B}^\top + \beta \boldsymbol{C}$ when `` transa=`N `` and `` transb=`T ``.

Our solution

```
type 'a trans (* = [ `N | `T | `C ] *)
val normal : (('m,'n) mat → ('m,'n) mat) trans (* = `N *)
val trans  : (('n,'m) mat → ('m,'n) mat) trans (* = `T *)
val conjtr : (('n,'m) mat → ('m,'n) mat) trans (* = `C *)

val gemm : ... → ?c:('m,'k) mat (* C *) →
  transa:(('x,'y) mat → ('m,'n) mat) trans → ('x,'y) mat (* A *) →
  transb:(('z,'w) mat → ('n,'k) mat) trans → ('z,'w) mat (* B *) →
  ('m,'k) mat (* C *)
```

Example of the typing (2) — Transpose flags

```
val gemm : ?alpha:num_type → ?beta:num_type → ?c:mat (* C *) →
  ?transa:[ `N | `T | `C ] → mat (* A *) →
  ?transb:[ `N | `T | `C ] → mat (* B *) → mat (* C *)
```

`transa` and `transb` specify no transpose (`` `N ``), transpose (`` `T ``) and conjugate transpose (`` `C ``) of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$:

- E.g., $\boldsymbol{C} := \alpha \boldsymbol{A}\boldsymbol{B}^\top + \beta \boldsymbol{C}$ when `` transa=`N `` and `` transb=`T ``.

Our solution

```
type 'a trans (* = [ `N | `T | `C ] *)
val normal : (('m,'n) mat → ('m,'n) mat) trans (* = `N *)
val trans  : (('n,'m) mat → ('m,'n) mat) trans (* = `T *)
val conjtr : (('n,'m) mat → ('m,'n) mat) trans (* = `C *)

val gemm : ... → ?c:('m,'k) mat (* C *) →
  transa:(('x,'y) mat → ('m,'n) mat) trans → ('x,'y) mat (* A *) →
  transb:(('z,'w) mat → ('n,'k) mat) trans → ('z,'w) mat (* B *) →
  ('m,'k) mat (* C *)
```

Example of the typing (2) — Transpose flags

```
val gemm : ?alpha:num_type → ?beta:num_type → ?c:mat (* C *) →
  ?transa:[ `N | `T | `C ] → mat (* A *) →
  ?transb:[ `N | `T | `C ] → mat (* B *) → mat (* C *)
```

transa and transb specify no transpose (`N), transpose (`T) and conjugate transpose (`C) of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$:

- E.g., $\boldsymbol{C} := \alpha \boldsymbol{A}\boldsymbol{B}^\top + \beta \boldsymbol{C}$ when transa=`N and transb=`T.

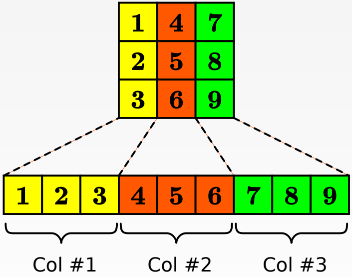
Our solution

```
type 'a trans (* = [ `N | `T | `C ] *)
val normal : (('m,'n) mat → ('m,'n) mat) trans (* = `N *)
val trans  : (('n,'m) mat → ('m,'n) mat) trans (* = `T *)
val conjtr : (('n,'m) mat → ('m,'n) mat) trans (* = `C *)

val gemm : ... → ?c:('m,'k) mat (* C *) →
  transa:(('x,'y) mat → ('m,'n) mat) trans → ('x,'y) mat (* A *) →
  transb:(('z,'w) mat → ('n,'k) mat) trans → ('z,'w) mat (* B *) →
  ('m,'k) mat (* C *)
```

Example of the typing (3) — Subtyping for discrete memory access

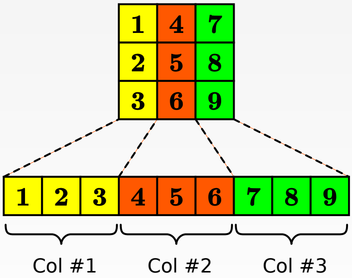
Elements in a matrix are stored in column-major order in flat, contiguous memory region.



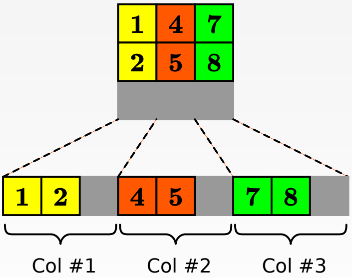
Example of the typing (3) — Subtyping for discrete memory access

Elements in a matrix are stored in column-major order in flat, contiguous memory region.

- BLAS & LAPACK functions can take (a) and (b).
- Some original functions of Lacaml can take **only** (a).



(a) Contiguous Matrix



(b) Discrete Matrix (Submatrix)

Example of the typing (3) — Subtyping for discrete memory access

The type of contiguous matrices $\quad <: \quad$ The type of discrete matrices

Example of the typing (3) — Subtyping for discrete memory access

The type of contiguous matrices $<:$ The type of discrete matrices

Our solution

We add a third parameter for "contiguous or discrete" flags:

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

	Argument Type (contravariant pos.)	Return Type (covariant pos.)
Contiguous	`('m,'n, cnt       ) mat`	`('m,'n,'cnt_or_dsc) mat`
Discrete	`('m,'n,'cnt_or_dsc) mat`	`('m,'n,        dsc) mat`

The type of contiguous matrices $<:$ The type of discrete matrices

Our solution

We add a third parameter for "contiguous or discrete" flags:

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

	Argument Type (contravariant pos.)	Return Type (covariant pos.)
Contiguous	`('m,'n, cnt ) mat`	`('m,'n,'cnt_or_dsc) mat`
Discrete	`('m,'n,'cnt_or_dsc) mat`	`('m,'n, dsc) mat`

The type of contiguous matrices $<:$ The type of discrete matrices

Our solution

We add a third parameter for "contiguous or discrete" flags:

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

	Argument Type (contravariant pos.)	Return Type (covariant pos.)
Contiguous	`('m,'n, cnt ) mat`	`('m,'n,'cnt_or_dsc) mat`
Discrete	`('m,'n,'cnt_or_dsc) mat`	`('m,'n, dsc) mat`

The type of contiguous matrices $<:$ The type of discrete matrices

Our solution

We add a third parameter for “contiguous or discrete” flags:

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

	Argument Type (contravariant pos.)	Return Type (covariant pos.)
Contiguous	`('m,'n, cnt ) mat`	`('m,'n,'cnt_or_dsc) mat`
Discrete	`('m,'n,'cnt_or_dsc) mat`	`('m,'n, dsc) mat`

The type of contiguous matrices $<:$ The type of discrete matrices

Our solution

We add a third parameter for “contiguous or discrete” flags:

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

	Argument Type (contravariant pos.)	Return Type (covariant pos.)
Contiguous	`('m,'n, cnt        ) mat`	`('m,'n,'cnt_or_dsc) mat`
Discrete	`('m,'n,'cnt_or_dsc) mat`	`('m,'n,        dsc) mat`

Outline

1. **Our idea**
   - Generative phantom types
   - Typing of BLAS and LAPACK functions

2. **Porting of OCaml-GPR**
   - Required changes
   - Percentages of required changes

Porting of OCaml-GPR

SLAP (Sized Linear Algebra Library)

- Our linear algebra library interface (a wrapper of Lacaml)
- Interface largely similar to Lacaml (to easily port existing programs)

Porting of OCaml-GPR

SLAP (Sized Linear Algebra Library)

- Our linear algebra library interface (a wrapper of Lacaml)
- Interface largely similar to Lacaml (to easily port existing programs)

OCaml-GPR (written by Markus Mottl)

- A practical machine learning library for Gaussian Process Regression
- Using Lacaml (**without** static size checking)

↓ Porting OCaml-GPR from Lacaml to SLAP

Sized GPR (SGPR)

- The ported library

A simple example

Computation of a covariant matrix of inputs given a kernel

```
val calc_upper : Kernel.t → Inputs.t → mat
```

↓

```
val calc_upper :
  ('D, _, _) Kernel.t →
  ('D, 'n) Inputs.t →      (* 'n vectors of dimension 'D *)
  ('n, 'n, 'cnt_or_dsc) mat (* 'n-by-'n contiguous matrix *)
```

Outline

Changes from OCaml-GPR to SGPR

We classified the changes under **19** categories:

- Mechanical changes (12 categories)

- Manual changes (7 categories)

Changes from OCaml-GPR to SGPR

We classified the changes under **19** categories:

- Mechanical changes (12 categories)

OCaml-GPR		SGPR
`x.{i,j}`	→	`get_dyn`, `set_dyn`
`` `N ``, `` `T ``, `` `C ``	→	`normal`, `trans`, `conjtr`
`vec`, `mat`	→	`('n,'cd) vec`, `('m,'n,'cd) mat`
⋮	⋮	⋮

- Manual changes (7 categories)
    - Next page...

Escaping generative phantom types

An example of manual changes

```
val Vec.init : int     → (int → float) →            vec (* Lacaml *)
val Vec.init : 'n size → (int → float) → ('n, 'cd) vec (* SLAP *)
```

- This can be compiled in Lacaml, but cannot in SLAP.

```
let vec_of_array a =
  Vec.init (Array.length a) (fun i → a.(i-1))
```

Escaping generative phantom types

An example of manual changes

```
val Vec.init : int     → (int → float) →            vec (* Lacaml *)
val Vec.init : 'n size → (int → float) → ('n, 'cd) vec (* SLAP *)
```

- This can be compiled in Lacaml, but cannot in SLAP.

```
let vec_of_array a =
  Vec.init (Array.length a) (fun i → a.(i-1))
```

- Does this fix work?

```
module type SIZE = sig
  type n (* generative phantom type *)
  val value : n size
end

let vec_of_array a =
  let module N = (val Size.of_int_dyn (Array.length a) : SIZE) in
  Vec.init N.value (fun i → a.(i-1))
```

Escaping generative phantom types

An example of manual changes

```
val Vec.init : int     → (int → float) →           vec (* Lacaml *)
val Vec.init : 'n size → (int → float) → ('n, 'cd) vec (* SLAP *)
```

- This can be compiled in Lacaml, but cannot in SLAP.

```
let vec_of_array a =
  Vec.init (Array.length a) (fun i → a.(i-1))
```

- Does this fix work?

```
module type SIZE = sig
  type n (* generative phantom type *)
  val value : n size
end

let vec_of_array a = (* : float array → (N.n, 'cd) vec *)
  let module N = (val Size.of_int_dyn (Array.length a) : SIZE) in
  Vec.init N.value (fun i → a.(i-1))
```

No. Generative phantom type `N.n` escapes its scope!

Two solutions to this problem (1)

1. To add extra arguments

```
let vec_of_array n a = (* : 'n size → float array → ('n, 'cd) vec *)
  assert(Size.to_int n = Array.length a);
  Vec.init n (fun i → a.(i-1))
```

- Generative phantom types are given from outside.
  - Not locally defined
- The code is simple,
- but **dynamic check** is needed.

Two solutions to this problem (2)

2. To use first-class modules

```
module type VEC = sig type n val value : (n, 'cd) vec end

let vec_of_array a = (* : float array → (module VEC) *)
  let module N = (val Size.of_int_dyn (Array.length a) : SIZE) in
  let module X = struct
    type n = N.n
    val value = Vec.init N.value (fun i → a.(i-1)) (*: (N.n,'cd) vec*)
  end in (module X : VEC)
```

- Packing
  - generative phantom type `N.n` and
  - vector of `(N.n,'cd) vec`

  as module X
- Type annotations of modules and heavy syntax

Trade-off of the two solutions

	Static size checking	Programming
1. extra arguments	no	easy
2. first-class modules	yes	(slightly) hard

- Both solutions have merits and demerits.
- In practical cases, they are in a trade-off relationship.

We used the first solution temporarily in SGPR.

Outline

Percentages of required changes

	Lines	Mechanical changes													Manual changes								Total
		S2I	SC	SOP	I2S	IDX	RF	IF	SUB	ETA	RID	RMDC	ITP	**Total**	ITA	EGPT	O2L	FT	ET	DKS	FS	**Total**	
lib/block_diag.mli	56	0	0	0	0	0	0	0	0	0	1	0	5	6	0	0	0	0	0	0	0	0	6
lib/block_diag.ml	58	1	0	0	0	0	0	0	0	0	1	6	1	9	0	0	0	0	0	0	0	0	9
lib/cov_const.mli	52	0	0	0	0	0	0	0	0	0	0	0	5	5	0	0	0	0	0	0	2	2	6
lib/cov_const.ml	141	2	0	0	0	1	0	0	0	0	2	0	9	14	0	1	0	0	0	1	8	10	16
lib/cov_lin_one.mli	56	0	0	0	0	0	0	0	0	0	1	0	5	6	0	0	0	0	0	0	2	2	7
lib/cov_lin_one.ml	149	0	0	0	0	1	4	2	0	0	2	0	9	17	0	0	0	0	0	1	12	13	26
lib/cov_lin_ard.mli	56	0	0	0	0	0	0	0	0	0	1	0	5	6	0	0	0	0	0	0	2	2	7
lib/cov_lin_ard.ml	188	7	0	0	0	10	5	2	0	0	2	0	9	32	0	0	0	0	0	1	8	9	39
lib/cov_se_iso.mli	58	0	0	0	0	0	0	0	0	0	1	0	5	6	0	0	0	0	0	0	2	2	7
lib/cov_se_iso.ml	343	18	4	0	0	31	0	0	0	0	2	2	14	71	2	0	0	0	0	1	6	9	78
lib/cov_se_fat.mli	105	0	0	0	0	0	0	0	0	0	1	0	10	11	0	0	0	4	0	0	0	4	15
lib/cov_se_fat.ml	680	43	9	1	0	87	2	2	0	0	2	8	23	174	0	0	0	28	0	1	1	30	199
lib/fitc_gp.mli	151	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lib/fitc_gp.ml	2294	81	3	3	0	63	19	26	14	34	3	15	69	298	0	28	31	16	0	0	0	68	364
lib/interfaces.ml	1008	0	0	0	0	0	0	0	0	0	1	0	196	197	0	11	7	5	0	3	0	26	215
lib/gpr_utils.ml	229	10	0	0	0	13	0	2	0	0	4	17	1	46	6	0	0	0	0	0	1	7	53
app/ocaml_gpr.ml	440	13	0	2	4	10	0	0	0	0	3	0	8	35	0	17	0	6	16	0	0	35	66
Total	6064	175	16	6	4	216	30	34	14	34	27	48	374	933	8	57	38	59	16	8	44	219	1113
Percentage	100.00	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	0.13	0.94	0.63	0.97	0.26	0.13	0.73	3.61	18.35

- Total changes: 18.35 % (1113 lines out of 6064 lines)
- Mechanical changes (12 categories): 15.39 % (933 lines)
  - Could be automated
- Manual changes (7 categories): **3.61 % (219 lines)**
  - Required a human brain

Porting of OCaml-GPR

Unfortunately (for us),
no bugs are found in Lacaml or OCaml-GPR.

Still, we believe SLAP and SGPR are useful:
An error can be detected

- earlier (i.e., at compile time instead of runtime)
- at higher level (i.e., at the caller site instead of in the call stack).

The static checking really helped during the porting!

- The OCaml typechecker showed us places requiring changes.

Conclusion

- Using generative phantom types
  - Verification of only equalities of sizes
- Only using fairly standard ML types and a few OCaml extensions
- Static sizes checking for most high-level matrix operations
- Easy to port existing application programs
  - Most of required changes can be made mechanically.

- **Sized Linear Algebra Library (SLAP)**
  - `https://github.com/akabe/slap`
- **Sized GPR (SGPR)**
  - `https://github.com/akabe/sgpr`
- **Details of changes** (containing the table of percentages)
  - `https://akabe.github.io/sgpr/changes.pdf`

APPENDIX

Appendix

Appendix

How to create vectors of the same dimension

Using a function whose type contains the same type parameter in

- the argument type and
- the return type.

Example 1. Using map:

```
val map : (float → float) → ('n, 'cnt_or_dsc) vec → ('n, 'cnt) vec
```

```
(* the dimension of y = the dimension of x *)
let y = map (fun xi → xi *. 2.0) x
```

Example 2. Using init:

```
val dim : ('n, 'cnt_or_dsc) vec → 'n size
val init : 'n size → (int → float) → ('n, 'cnt) vec
```

```
(* the dimension of z = the dimension of x *)
let z = init (dim x) (fun i → float_of_int i)
```

Example of the typing — Side flags

Multiplication of $k \times k$ symmetric matrix $\boldsymbol{A}$ and $m \times n$ matrix $\boldsymbol{B}$:

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

- If `side=`L`, $\boldsymbol{C} := \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$ where $\boldsymbol{A}$ is a $m \times m$ matrix.
- If `side=`R`, $\boldsymbol{C} := \alpha \boldsymbol{B}\boldsymbol{A} + \beta \boldsymbol{C}$ where $\boldsymbol{A}$ is a $n \times n$ matrix.

Example of the typing — Side flags

Multiplication of $k \times k$ symmetric matrix $\boldsymbol{A}$ and $m \times n$ matrix $\boldsymbol{B}$:

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

- If side=‘L, $\boldsymbol{C} := \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$ where $\boldsymbol{A}$ is a $m \times m$ matrix.
- If side=‘R, $\boldsymbol{C} := \alpha\boldsymbol{B}\boldsymbol{A} + \beta\boldsymbol{C}$ where $\boldsymbol{A}$ is a $n \times n$ matrix.

Our solution

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

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 (* C *)
```

Appendix

3 **Our idea**

- How to create vectors of the same dimension
- Side flags for square matrix multiplication
- Our original function "subvec_dyn"
- Our original function "submat_dyn"

4 **Porting of OCaml-GPR**

- Insertion of type parameters (ITA)

5 **Related works**

- Lightweight Static Capabilities

Subvectors

dot computs inner product of two vectors.

```
val dot : ?n:int → ?ofsx:int → ?incx:int → x:vec →
  ?ofsy:int → ?incy:int → vec (* y *) → float
```

$$\sum_{i=1}^{\mathtt{n}} \mathtt{x}[\mathtt{ofsx} + (i-1)\mathtt{incx}] \times \mathtt{y}[\mathtt{ofsy} + (i-1)\mathtt{incy}]$$

(x$[i]$ is the i-th element of vector x.)

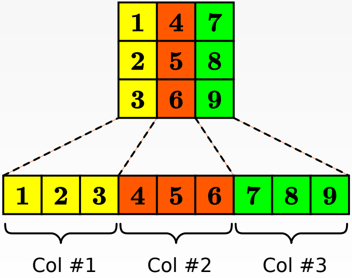
Subvectors

dot computs inner product of two vectors.

```
val dot : ?n:int → ?ofsx:int → ?incx:int → x:vec →
  ?ofsy:int → ?incy:int → vec (* y *) → float
```

$$\sum_{i=1}^{\mathtt{n}} \mathsf{x}[\mathtt{ofsx} + (i-1)\mathtt{incx}] \times \mathsf{y}[\mathtt{ofsy} + (i-1)\mathtt{incy}]$$

($\mathsf{x}[i]$ is the i-th element of vector x.)



- `ofs` and `inc` are used to treat a column or a row without copy.

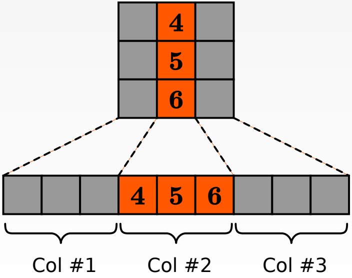
Subvectors

dot computs inner product of two vectors.

```
val dot : ?n:int → ?ofsx:int → ?incx:int → x:vec →
  ?ofsy:int → ?incy:int → vec (* y *) → float
```

$$\sum_{i=1}^{\mathtt{n}} \mathtt{x}[\mathtt{ofsx} + (i-1)\mathtt{incx}] \times \mathtt{y}[\mathtt{ofsy} + (i-1)\mathtt{incy}]$$

($\mathtt{x}[i]$ is the i-th element of vector x.)



- ofs and inc are used to treat a column or a row without copy.

Examples:

- **2nd column**: n=3, ofs=4, inc=1

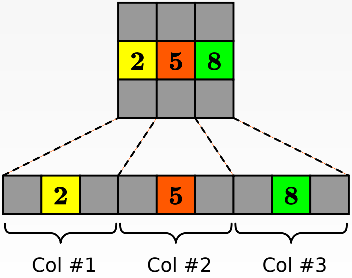
Subvectors

dot computs inner product of two vectors.

```
val dot : ?n:int → ?ofsx:int → ?incx:int → x:vec →
  ?ofsy:int → ?incy:int → vec (* y *) → float
```

$$\sum_{i=1}^{\mathtt{n}} \mathsf{x}[\mathtt{ofsx} + (i-1)\mathtt{incx}] \times \mathsf{y}[\mathtt{ofsy} + (i-1)\mathtt{incy}]$$

($\mathsf{x}[i]$ is the i-th element of vector x.)



- ofs and inc are used to treat a column or a row without copy.

Examples:

- **2nd column**: n=3, ofs=4, inc=1
- **2nd row**: n=3, ofs=2, inc=3

etc.

Our original function "subvec_dyn"

All BLAS and LAPACK functions can take subvectors (i.e., `ofs` and `inc`).

Q. Should we add dynamic checks for subvectors to all functions?

Our original function "subvec_dyn"

All BLAS and LAPACK functions can take subvectors (i.e., `ofs` and `inc`).

Q. Should we add dynamic checks for subvectors to all functions?

A. It is undesirable because subvector designation is auxiliary.

E.g., in the case of `dot`,

- **computation of inner product** is essential, but
- **subvector designation** is auxiliary.

Our original function “subvec_dyn”

All BLAS and LAPACK functions can take subvectors (i.e., `ofs` and `inc`).

Q. Should we add dynamic checks for subvectors to all functions?

A. It is undesirable because subvector designation is auxiliary.

E.g., in the case of `dot`,

- **computation of inner product** is essential, but
- **subvector designation** is auxiliary.

Our solution

We defined separate function `subvec_dyn` to return a subvector.

```
  dot ~n ~ofsx ~incx ~x y (* Lacaml *)
→ dot ~x:(subvec_dyn ~n ~ofsx ~incx x) y (* SLAP *)
```

Appendix

Our original function “submat_dyn”

All BLAS and LAPACK functions can take submatrices.

Q. Should we add dynamic checks for submatrices to all functions?

Our original function “submat_dyn”

All BLAS and LAPACK functions can take submatrices.

Q. Should we add dynamic checks for submatrices to all functions?

A. It is undesirable because submatrix designation is auxiliary.

E.g., `lacpy` copies (sub-)matrix A to (sub-)matrix B.

```
val lacpy : ... → ?m:int → ?n:int →
  ?br:int → ?bc:int → ?b:mat (* B *) →
  ?ar:int → ?ac:int → mat (* A *) → mat (* B *)
```

- **Copying** is essential, but
- **submatrix designation** is auxiliary to `lacpy`.

Our original function “submat_dyn”

All BLAS and LAPACK functions can take submatrices.

Q. Should we add dynamic checks for submatrices to all functions?

A. It is undesirable because submatrix designation is auxiliary.

E.g., `lacpy` copies (sub-)matrix A to (sub-)matrix B.

```
val lacpy : ... → ?m:int → ?n:int →
  ?br:int → ?bc:int → ?b:mat (* B *) →
  ?ar:int → ?ac:int → mat (* A *) → mat (* B *)
```

- **Copying** is essential, but
- **submatrix designation** is auxiliary to `lacpy`.

Our solution

We defined separate function `submat_dyn` to return a submatrix.

```
  lacpy ~m ~n ~ar ~ac a (* Lacaml *)
→ lacpy (submat_dyn ~m ~n ~ar ~ac a) (* SLAP *)
```

Appendix

Appendix

Insertion of type parameters (ITA)

An example of manual changes

- In many cases, size constraints are automatically inferred by OCaml.
- By using low-level operations, they cannot be probably derived.

Example

This computes $\alpha\boldsymbol{x} + \beta\boldsymbol{y}$ $(\boldsymbol{x}, \boldsymbol{y}$: vector; α, β : scalar)

```
let axby alpha beta x y =
  let n = Vec.dim x in (* Vec.dim : ('n, 'cd) vec → 'n size *)
  Vec.init n
    (fun i → alpha *. (Vec.get_dyn x i) +. beta *. (Vec.get_dyn y i))
```

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

'n and 'm should be equal!

Two solutions to this problem

1. To type-annotate axby by hand

```
let axby alpha beta (x : ('n, _) vec) (y : ('n, _) vec) =
  let n = Vec.dim x in (* Vec.dim : ('n, 'cd) vec → 'n size *)
  Vec.init n
    (fun i → alpha *. (Vec.get_dyn x i) +. beta *. (Vec.get_dyn y i))
```

We used this way to rewrite OCaml-GPR as simple as possible.

2. To use high-level matrix operations

```
let axby alpha beta x y =
  Vec.map2 (fun xi yi → alpha *. xi +. beta *. yi) x y
```

where

```
val Vec.map2 : (float→float) → ('n,_) vec → ('n,_) vec → ('n,_) vec
```

Appendix

Lightweight Static Capabilities

Kiselyov and Shan. Lightweight Static Capabilities. PLPV. 2006.

- Static checking of inequalities in OCaml
- Compatibility with Dependent ML
- CPS encoding of existential types using first-class polymorphism
- Needing CPS conversion
  - Changing structures of programs (e.g., relationship of function calls)

Our approach

- Static checking of only equalities in OCaml
- Compatibility with Lacaml (without static checking)
- Existential types using first-class modules
- Needing the conversion of "escaping generative phantom types."
  - Without significantly restructuring of programs