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
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
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

```haskell
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 *)

• Addition of two vectors loaded from different files:
  let (x : ? vec) = loadvec "file1"
  in let (y : ? 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 : ? vec) = loadvec "file1"
  in let (y : ? 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

```ocaml
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:

```ocaml
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:

```ocaml
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:

```ocaml
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**

```ocaml
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

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

2 Porting of OCaml-GPR
Required changes
Percentages of required changes
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
• `dot x y` computes inner product of `x` and `y`.

```
val dot : x:vec → 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 (1)

- **dot** \(\sim x \ y\) computes inner product of \(x\) and \(y\).

  \[
  \text{val} \quad \text{dot} : x : \text{vec} \rightarrow \text{vec} \rightarrow \text{float}
  \]

- **axpy** \(?alpha\ \sim x \ y\) computes \(y := \alpha \times x + y\).

  \[
  \text{val} \quad \text{axpy} : ?alpha : \text{float} \rightarrow x : \text{vec} \rightarrow \text{vec} \rightarrow \text{unit}
  \]
Example of the typing (2) — Transpose flags

| ?transb:[ ‘N | ‘T | ‘C ] → mat (* B *) → mat (* C *) |

transa and transb specify no transpose (‘N), transpose (‘T) and conjugate transpose (‘C) of matrices $A$ and $B$:

- E.g., $C := \alpha AB^\top + \beta C$ when $\text{transa}=‘N$ and $\text{transb}=‘T$.  


Example of the typing (2) — Transpose flags

```
  ?transb:[ ‘N | ‘T | ‘C ] → mat (* B *) → mat (* C *)
```

transa and transb specify no transpose (‘N), transpose (‘T) and
conjugate transpose (‘C) of matrices A and B:
  • E.g., $C := \alpha AB^\top + \beta C$ 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**

\[
\text{val \ \text{gemm}} : \ ?\alpha: \text{num\_type} \rightarrow \ ?\beta: \text{num\_type} \rightarrow \ ?c: \text{mat \ (*C*)} \rightarrow \\
\ ?\text{transa: [ 'N | 'T | 'C ]} \rightarrow \text{mat \ (*A*)} \rightarrow \\
\ ?\text{transb: [ 'N | 'T | 'C ]} \rightarrow \text{mat \ (*B*)} \rightarrow \text{mat \ (*C*)}
\]

transa and transb specify no transpose (‘N), transpose (‘T) and conjugate transpose (‘C) of matrices \( A \) and \( B \):

- E.g., \( C := \alpha AB^\top + \beta C \) when transa=‘N and transb=‘T.

**Our solution**

**type \ 'a \ trans \ (* = [ 'N | 'T | 'C ] *)**

**val \ normal \ : \ (('m,'n) \ mat \ \rightarrow \ ('m,'n) \ mat) \ trans \ (* = 'N *)**

**val \ trans \ : \ (('n,'m) \ mat \ \rightarrow \ ('m,'n) \ mat) \ trans \ (* = 'T *)**

**val \ conjtr \ : \ (('n,'m) \ mat \ \rightarrow \ ('m,'n) \ mat) \ trans \ (* = 'C *)**

**val \ gemm \ : \ ... \ \rightarrow \ ?c:('m,'k) \ mat \ (*C*) \ \rightarrow \\
transa:((x,y) \ mat \ \rightarrow \ ('m,'n) \ mat) \ trans \ \rightarrow \ (x,y) \ mat \ (*A*) \ \rightarrow \\
transb:((z,w) \ mat \ \rightarrow \ ('n,'k) \ mat) \ trans \ \rightarrow \ (z,w) \ mat \ (*B*) \ \rightarrow \\
\ ('m,'k) \ mat \ (*C*)**
Example of the typing (2) — Transpose flags

\[
\text{val } \text{gemm} : \ ?\alpha:\text{num}\_\text{type} \rightarrow \ ?\beta:\text{num}\_\text{type} \rightarrow \ ?c:\text{mat} \ (*\ C\ *) \rightarrow \\
\ ?\text{transa}:[ \ 'N | 'T | 'C \ ] \rightarrow \ \text{mat} \ (*\ A\ *) \rightarrow \\
\ ?\text{transb}:[ \ 'N | 'T | 'C \ ] \rightarrow \ \text{mat} \ (*\ B\ *) \rightarrow \ \text{mat} \ (*\ C\ *)
\]

transa and transb specify no transpose (‘N), transpose (‘T) and conjugate transpose (‘C) of matrices \( A \) and \( B \):

- E.g., \( C := \alpha AB^\top + \beta C \) when transa=‘N and transb=‘T.

Our solution

\[\text{type } 'a\ \text{trans} \ (* = [ \ 'N | 'T | 'C \ ] \ *)\]

\[\text{val normal} : (('m,'n) \ \text{mat} \rightarrow ('m,'n) \ \text{mat}) \ \text{trans} \ (* = 'N *)\]

\[\text{val trans} : (('n,'m) \ \text{mat} \rightarrow ('m,'n) \ \text{mat}) \ \text{trans} \ (* = 'T *)\]

\[\text{val conjtr} : (('n,'m) \ \text{mat} \rightarrow ('m,'n) \ \text{mat}) \ \text{trans} \ (* = 'C *)\]

\[\text{val gemm} : \ldots \rightarrow ?c:('m,'k) \ \text{mat} \ (*\ C\ *) \rightarrow \\
\ ?\text{transa}:(('x,'y) \ \text{mat} \rightarrow ('m,'n) \ \text{mat}) \ \text{trans} \rightarrow ('x,'y) \ \text{mat} \ (*\ A\ *) \rightarrow \\
\ ?\text{transb}:(('z,'w) \ \text{mat} \rightarrow ('n,'k) \ \text{mat}) \ \text{trans} \rightarrow ('z,'w) \ \text{mat} \ (*\ B\ *) \rightarrow \\
\ ('m,'k) \ \text{mat} \ (*\ C\ *)\]
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).

Example of a contiguous matrix:

```
  1 2 3 4 5 6 7 8 9
```

Example of a discrete matrix (submatrix):

```
  1 4 7
  2 5 8
  3 6 9
```

Col #1 Col #2 Col #3
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).
The type of contiguous matrices $<$: The type of discrete matrices
Example of the typing (3) — Subtyping for discrete memory access

The type of contiguous matrices $\leq$: The type of discrete matrices

Our solution

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

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

<table>
<thead>
<tr>
<th>Argument Type (contravariant pos.)</th>
<th>Return Type (covariant pos.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>('m, 'n, cnt) mat</td>
<td>('m, 'n, 'cnt_or_dsc) mat</td>
</tr>
<tr>
<td>('m, 'n, 'cnt_or_dsc) mat</td>
<td>('m, 'n, dsc) mat</td>
</tr>
</tbody>
</table>
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:

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

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(contravariant pos.)</td>
<td>(covariant pos.)</td>
</tr>
<tr>
<td>Contiguous Discrete</td>
<td></td>
</tr>
<tr>
<td>('m,'n, cnt) mat</td>
<td>('m,'n,'cnt_or_dsc) mat</td>
</tr>
<tr>
<td>('m,'n,'cnt_or_dsc) mat</td>
<td>('m,'n, dsc) mat</td>
</tr>
</tbody>
</table>
Example of the typing (3) — Subtyping for discrete memory access

The type of contiguous matrices $\ll$ The type of discrete matrices

Our solution

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

- \( \text{type ('m,'n,'cnt_or_dsc) mat (* 'm-by-'n matrices *)} \)
- \( \text{type cnt (* phantom *)} \)
- \( \text{type dsc (* phantom *)} \)

<table>
<thead>
<tr>
<th>Contiguous</th>
<th>Argument Type (contravariant pos.)</th>
<th>Return Type (covariant pos.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>('m,'n,'cnt_or_dsc) mat</td>
<td>('m,'n,'cnt_or_dsc) mat</td>
</tr>
<tr>
<td></td>
<td>('m,'n,'cnt_or_dsc) mat</td>
<td>('m,'n,dsc) mat</td>
</tr>
</tbody>
</table>
The type of contiguous matrices is a subtype of the type of discrete matrices.

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

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

<table>
<thead>
<tr>
<th>Argument Type (contravariant pos.)</th>
<th>Return Type (covariant pos.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>('m,'n, cnt) mat</td>
<td>('m,'n,'cnt_or_dsc) mat</td>
</tr>
<tr>
<td>('m,'n,'cnt_or_dsc) mat</td>
<td>('m,'n, dsc) mat</td>
</tr>
</tbody>
</table>
Example of the typing (3) — Subtyping for discrete memory access

The type of contiguous matrices $\ll$: The type of discrete matrices

Our solution

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

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

<table>
<thead>
<tr>
<th>Argument Type (contravariant pos.)</th>
<th>Return Type (covariant pos.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>('m,'n, cnt) mat</td>
<td>('m,'n,'cnt_or_dsc) mat</td>
</tr>
<tr>
<td>('m,'n,'cnt_or_dsc) mat</td>
<td>('m,'n, dsc) mat</td>
</tr>
</tbody>
</table>
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

```ocaml
val calc_upper : Kernel.t \rightarrow Inputs.t \rightarrow mat
```

```ocaml
down
val calc_upper :
  ('D, _, _) Kernel.t \rightarrow
  ('D, 'n) Inputs.t \rightarrow (* 'n vectors of dimension 'D *)
  ('n, 'n, 'cnt_or_dsc) mat (* 'n-by-'n contiguous matrix *)
```
1. Our idea
   Generative phantom types
   Typing of BLAS and LAPACK functions

2. Porting of OCaml-GPR
   Required changes
   Percentages of required changes
We classified the changes under 19 categories:

- Mechanical changes (12 categories)

- Manual changes (7 categories)
We classified the changes under 19 categories:

- Mechanical changes (12 categories)

<table>
<thead>
<tr>
<th>OCaml-GPR</th>
<th>SGPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.{i,j}</td>
<td>get_dyn, set_dyn</td>
</tr>
<tr>
<td>‘N, ‘T, ‘C</td>
<td>normal, trans, conjtr</td>
</tr>
<tr>
<td>vec, mat</td>
<td>(’n,’cd) vec, (’m,’n,’cd) mat</td>
</tr>
</tbody>
</table>

- Manual changes (7 categories)
  - Next page...
Escaping generative phantom types
An example of manual changes

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

```ocaml
let vec_of_array a =  
  Vec.init (Array.length a) (fun i -> a.(i-1))
```
Escaping generative phantom types
An example of manual changes

\[
\text{val Vec.init : int } \rightarrow \text{ (int } \rightarrow \text{ float) } \rightarrow \text{ vec (* Lacaml *)}\\
\text{val Vec.init : 'n size } \rightarrow \text{ (int } \rightarrow \text{ float) } \rightarrow \text{ ('n, 'cd) vec (* SLAP *)}
\]

• This can be compiled in Lacaml, but \textbf{cannot} in SLAP.

\[
\text{let vec_of_array a =}\\
\text{ Vec.init (Array.length a) (fun i \rightarrow a.(i-1))}
\]

• Does this fix work?

\[
\text{module type SIZE = sig}\\
\text{ type n (* generative phantom type *)}\\
\text{ val value : n size}\\
\text{end}
\]

\[
\text{let vec_of_array a =}\\
\text{ let module N = (val Size.of_int_dyn (Array.length a) : SIZE) in}\\
\text{ Vec.init N.value (fun i \rightarrow a.(i-1))}
\]
### 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.

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

- Does this fix work?

```ocaml
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

```ocaml
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

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

let vec_of_array a = (* : float array \rightarrow (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 \rightarrow 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

<table>
<thead>
<tr>
<th></th>
<th>Static size checking</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. extra arguments</td>
<td>no</td>
<td>easy</td>
</tr>
<tr>
<td>2. first-class modules</td>
<td>yes</td>
<td>(slightly) hard</td>
</tr>
</tbody>
</table>

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

We used the first solution temporarily in SGPR.
1 Our idea
   Generative phantom types
   Typing of BLAS and LAPACK functions

2 Porting of OCaml-GPR
   Required changes
   Percentages of required changes
## Percentages of required changes

<table>
<thead>
<tr>
<th>Lines</th>
<th>S2I</th>
<th>SC</th>
<th>SOP</th>
<th>I2S</th>
<th>IDX</th>
<th>RF</th>
<th>IF</th>
<th>SUB</th>
<th>ETA</th>
<th>RID</th>
<th>RMDC</th>
<th>ITP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/block_diag.mli</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/block_diag.ml</td>
<td>58</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/const.ml</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/const.ml</td>
<td>141</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>lib/l.in-one.ml</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/l.in-one.ml</td>
<td>149</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>lib/l.in-ard.ml</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/l.in-ard.ml</td>
<td>188</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>lib/se.iso.ml</td>
<td>58</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/se.iso.ml</td>
<td>343</td>
<td>18</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>lib/se.iso.ml</td>
<td>105</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lib/se.iso.ml</td>
<td>680</td>
<td>43</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lib/gp.ml</td>
<td>151</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lib/gp.ml</td>
<td>2294</td>
<td>81</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>lib/interfaces.ml</td>
<td>1008</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lib/gp_utils.ml</td>
<td>229</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>app/ocaml/gp.ml</td>
<td>440</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>6064</td>
<td>6064</td>
<td>175</td>
<td>16</td>
<td>6</td>
<td>4</td>
<td>216</td>
<td>30</td>
<td>34</td>
<td>14</td>
<td>34</td>
<td>27</td>
<td>48</td>
</tr>
</tbody>
</table>

| 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 |

### Mechanical changes (12 categories): 15.39 % (933 lines)
- Could be automated

### Manual changes (7 categories): 3.61 % (219 lines)
- Required a human brain

- Total changes: 18.35 % (1113 lines out of 6064 lines)
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](https://github.com/akabe/slap)
- **Sized GPR (SGPR)**
  - [https://github.com/akabe/sgpr](https://github.com/akabe/sgpr)
- **Details of changes** (containing the table of percentages)
  - [https://akabe.github.com/sgpr/changes.pdf](https://akabe.github.com/sgpr/changes.pdf)
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
### 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”

### Porting of OCaml-GPR

Insertion of type parameters (ITA)

### Related works

Lightweight Static Capabilities
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:**

```ocaml
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:**

```ocaml
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)
```
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
Example of the typing — Side flags

Multiplication of \( k \times k \) symmetric matrix \( A \) and \( m \times n \) matrix \( B \):

\[
\text{val symm : ?side: [‘L|‘R] } \rightarrow \ ?\beta: \text{num_type} \rightarrow \ ?c: \text{mat (* C *) } \rightarrow \\
\text{?alpha: num_type } \rightarrow \ \text{mat (* A *) } \rightarrow \ \text{mat (* B *) } \rightarrow \ \text{mat (* C *)}
\]

- If side=‘L, \( C := \alpha AB + \beta C \) where \( A \) is a \( m \times m \) matrix.
- If side=‘R, \( C := \alpha BA + \beta C \) where \( A \) is a \( n \times n \) matrix.
Example of the typing — Side flags

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

\[
\text{val symm : ?side:} ['L'|'R'] \rightarrow \text{?beta:num_type} \rightarrow \text{?c:mat (* C *)} \rightarrow \\
\text{?alpha:num_type} \rightarrow \text{mat (* A *)} \rightarrow \text{mat (* B *)} \rightarrow \text{mat (* C *)}
\]

- If side='L, $C := \alpha AB + \beta C$ where $A$ is a $m \times m$ matrix.
- If side='R, $C := \alpha BA + \beta C$ where $A$ is a $n \times n$ matrix.

Our solution

\[
\text{type ('k, 'm, 'n) side (* = [ 'L | 'R ] *)}
\]
\[
\text{val left : ('m, 'm, 'n) side (* = 'L *)}
\]
\[
\text{val right : ('n, 'm, 'n) side (* = 'R *)}
\]
\[
\text{val symm : side:('k, 'm, 'n) side} \rightarrow \text{?beta:num_type} \rightarrow \\
\text{?c:('m, 'n) mat (* C *)} \rightarrow \text{?alpha:num_type} \rightarrow \\
\text{('k, 'k) mat (* A *)} \rightarrow \text{('m, 'n) mat (* B *)} \rightarrow \\
\text{('m, 'n) mat (* C *)}
\]
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 computes inner product of two vectors.

\[
\begin{align*}
\text{val} \quad \text{dot} : \ ?n: \text{int} & \rightarrow \ ?\text{ofs}x: \text{int} \rightarrow \ ?\text{inc}x: \text{int} \rightarrow x: \text{vec} \rightarrow \\
& \rightarrow \ ?\text{ofs}y: \text{int} \rightarrow \ ?\text{incy}: \text{int} \rightarrow \text{vec} (* y *) \rightarrow \text{float}
\end{align*}
\]

\[
\sum_{i=1}^{n} x[\text{ofs}x + (i - 1)\text{inc}x] \times y[\text{ofs}y + (i - 1)\text{incy}]
\]

(\text{x}[i] \text{ is the } i\text{-th element of vector } x.)
dot computes inner product of two vectors.

\[
\sum_{i=1}^{n} x[ofs + (i-1)inc] \times y[ofs + (i-1)incy]
\]

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

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

\begin{center}
\begin{tabular}{c|c|c}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{tabular}
\end{center}

Col #1  Col #2  Col #3
dot computes inner product of two vectors.

\[
\sum_{i=1}^{n} x[\text{ofs} + (i-1)\text{inc}] \times y[\text{ofs} + (i-1)\text{incy}]
\]

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

- \texttt{ofs} and \texttt{inc} are used to treat a column or a row without copy.

Examples:
- **2nd column**: \(n=3\), \(ofs=4\), \(inc=1\)
Subvectors

dot computes inner product of two vectors.

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

(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.
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.

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


### 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

---

**Appendix**

39 / 29
Our original function “submat_dyn”

All BLAS and LAPACK functions can take submatrices.

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

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

```
val lacpy : ...
→ ?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 *)`
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 →
  ?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 \( B \).

\[
\text{val lacpy : } \ldots \rightarrow \ ?m:\text{int} \rightarrow \ ?n:\text{int} \rightarrow \\
?br:\text{int} \rightarrow \ ?bc:\text{int} \rightarrow ?b:\text{mat} (\ast \ B \ast) \rightarrow \\
?ar:\text{int} \rightarrow \ ?ac:\text{int} \rightarrow \text{mat} (\ast \ A \ast) \rightarrow \text{mat} (\ast \ B \ast)
\]

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

Our solution

We defined separate function submat_dyn to return a submatrix.

\[
lacpy \sim m \sim n \sim ar \sim ac \ a (\ast \ \text{Lacaml} \ \ast)
\]

\[
\rightarrow \ lacpy (\text{submat_dyn} \sim m \sim n \sim ar \sim ac \ a) (\ast \ \text{SLAP} \ \ast)
\]
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
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
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 \( \alpha x + \beta y \) \((x, y : \text{vector}; \alpha, \beta : \text{scalar})\)

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

val axby : float \to float \to ('n, _) vec \to ('m, _) vec \to ('n, _) vec

'n and 'm should be equal!
```
Two solutions to this problem

1. To **type-annotate** `axby` by hand

```ocaml
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

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

where

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