単純かつ実用的な静的サイズ検査つき線形代数演算ライブラリの試作と検証

PPL 2014
平成 26 年 3 月 5 日（水）

阿部晃典 [1]　住井英二郎 [2]

[1] 東北大学 工学部 情報知能システム総合学科
abe@kb.ecei.tohoku.ac.jp

[2] 東北大学 大学院 情報科学研究科
sumii@ecei.tohoku.ac.jp

行列演算のサイズ検査

多くの数値計算言語や線形代数演算ライブラリは
行列演算のサイズの整合性を静的に 保証しない
(MatLab, BLAS, LAPACK, *etc.*)

- サイズの合わない行列演算は実行時エラーや不具合の原因（例外やメモリ破壊など）

行列演算のサイズ検査

多くの数値計算言語や線形代数演算ライブラリは
行列演算のサイズの整合性を静的に保証しない
(MatLab, BLAS, LAPACK, *etc.*)

- サイズの合わない行列演算は実行時エラーや不具合の原因（例外やメモリ破壊など）

行列演算の静的サイズ検査が実現できれば，

- バグの早期発見
- 実行時サイズ検査の除去

が期待できる

既存研究

自然数上の依存型を用いた既存研究：

- Dependent ML [Xi and Pfenning, 1999], ATS [Xi]
- sized type [Chin and Khoo, 2001]

型システムやアプリケーションプログラムに
自明でない変更が必要！

既存研究

自然数上の依存型を用いた既存研究：

- Dependent ML [Xi and Pfenning, 1999], ATS [Xi]
- sized type [Chin and Khoo, 2001]

型システムやアプリケーションプログラムに
自明でない変更が必要！

【疑問】
こんな静的サイズ検査を実現できないか？

- 既存のメジャーなプログラミング言語で実現できる
- アプリケーションプログラムに簡単に導入できる

本研究の貢献

生成的 (generative) な幽霊型 (phantom type) により
静的サイズ検査を行う線形代数演算ライブラリ

【特徴】

- ほぼ標準的な ML の型システムとモジュールシステムで実現可能
  - 関数型プログラミング言語 OCaml でライブラリを試作
- ほとんどの高水準な行列演算（行列積など）の静的サイズ検査が可能
  - （添字アクセスなど）一部の低水準な演算などは実行時検査を伴う
- 既存のアプリケーションプログラムへの変更がほとんど不要

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証

型レベル自然数による静的なサイズ情報の表現

```
module M : sig
  type z       (* ゼロ                   --- 幽霊型 (phantom type) *)
  type 'n s    (* 後者(successor) --- 幽霊型 (phantom type) *)










end
```

型レベル自然数による静的なサイズ情報の表現

```
module M : sig
  type z       (* ゼロ              --- 幽霊型 (phantom type) *)
  type 'n s    (* 後者(successor) --- 幽霊型 (phantom type) *)
  type 'n vec (* 'n 次元ベクトルの型 *)





  type ('m, 'n) mat (* 'm 行 'n 列の行列の型 *)



end
```

型レベル自然数による静的なサイズ情報の表現

```
module M : sig
  type z      (* ゼロ              --- 幽霊型 (phantom type) *)
  type 'n s   (* 後者(successor) --- 幽霊型 (phantom type) *)
  type 'n vec (* 'n 次元ベクトルの型 *)
  val empty : z vec
  val cons  : float -> 'n vec -> 'n s vec
  val add   : 'n vec -> 'n vec -> 'n vec (* ベクトルの加算 *)
  ...

  type ('m, 'n) mat (* 'm 行 'n 列の行列の型 *)
  val add_mat : ('m, 'n) mat -> ('m, 'n) mat -> ('m, 'n) mat
  val mul_mat : ('m, 'n) mat -> ('n, 'k) mat -> ('m, 'k) mat
  ...
end
```

型レベル自然数による静的サイズ検査

```
# open M

# let a = cons 1.0 (cons 2.0 empty)
val a : z s s vec

# let b = cons 3.0 a
val b : z s s s vec

# let c = cons 4.0 (cons 5.0 (cons 6.0 empty))
val c : z s s s vec

# add b c (* 型が付く (b と c は同じサイズ) *)
- : z s s s vec

# add a b (* 型エラー (a と b は異なるサイズ) *)
```

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証

生成的な幽霊型が必要になる場合

```
val loadvec : string -> ? vec (* ファイルからベクトルを読み込む *)
```

生成的な幽霊型が必要になる場合

```
val loadvec : string -> ? vec (* ファイルからベクトルを読み込む *)
```

- 異なるファイルから読み込んだベクトルの加算

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* 型エラーを起こすべき (?¹ ≠ ?²) *)
```

生成的な幽霊型が必要になる場合

```
val loadvec : string -> ? vec (* ファイルからベクトルを読み込む *)
```

- 異なるファイルから読み込んだベクトルの加算

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* 型エラーを起こすべき (?¹ ≠ ?²) *)
```

- 同じファイルから読み込んだベクトルの加算

```
let (x : ?¹ vec) = loadvec "file" in
let (y : ?² vec) = loadvec "file" in
add x y (* 型エラーを起こすべき (?¹ ≠ ?²) *)
```

読み込み中にファイルが書き換えられるかもしれない

生成的な幽霊型が必要になる場合

```
val loadvec : string -> ? vec (* ファイルからベクトルを読み込む *)
```

- 異なるファイルから読み込んだベクトルの加算

```
let (x : ?¹ vec) = loadvec "file1" in
let (y : ?² vec) = loadvec "file2" in
add x y (* 型エラーを起こすべき (?¹ ≠ ?²) *)
```

- 同じファイルから読み込んだベクトルの加算

```
let (x : ?¹ vec) = loadvec "file" in
let (y : ?² vec) = loadvec "file" in
add x y (* 型エラーを起こすべき (?¹ ≠ ?²) *)
```

読み込み中にファイルが書き換えられるかもしれない

したがって，loadvec の戻り値の型は（引数の値に関係なく）
「実行する度に異なる型」になるべき

生成的な幽霊型

```
val loadvec : string -> ? vec
```

? は生成的な幽霊型：

- 関数は呼ばれる度にフレッシュな型の値を返す
  - 生成的な型 — 値が作られる度に，新たに与えられる型
- ∃n. n vec のような存在限量化されたサイズつき型を表現
  - コンパイル時に実行時のサイズは不明
  - 他のいずれのベクトルとも異なるサイズと仮定

生成的な幽霊型の実現

OCaml では,「戻り値の型が生成的な関数」は書けない

- これ以降は，そのような関数を書ける仮想的な言語で説明
- 本研究ではファンクターを用いて実現

```
module F (S : sig end) : sig
  type t              (* 生成的な幽霊型に相当 *)
  val value : t vec (* ? vec に相当 *)
end = struct
  type t
  let value = ...
end

module M1 = F(struct end)
module M2 = F(struct end)
add M1.value M2.value (* 型エラー (M1.t ≠ M2.t) *)
```

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証

生成的な幽霊型のみによる静的サイズ検査

型レベル自然数は演算が大変 ➔ 生成的な幽霊型のみを使う

- `val cons : float -> 'n vec -> 'n s vec`
- `val tl : 'n s vec -> 'n vec`

生成的な幽霊型 のみ による静的サイズ検査

型レベル自然数は 演算が大変 ➔ 生成的な幽霊型 のみ を使う

- `val cons : float -> 'n vec -> ~~'n s~~ vec`
  ?
- `val tl_dyn : ~~'n s~~ vec -> ~~'n~~ vec`
  'n ?

  ※ _dyn は実行時検査を伴う関数

サイズの等しさしか検査しない！ ➔ 型付けが大幅に単純化

生成的な幽霊型のみによる静的サイズ検査

型レベル自然数は演算が大変 ➔ 生成的な幽霊型のみを使う

- `val cons    : float -> 'n vec -> ~~'n s~~ vec`
  (`'n s` → `?`)
- `val tl_dyn : ~~'n s~~ vec -> ~~'n~~ vec`
  (`'n s` → `'n`, `'n` → `?`)

  ※ _dyn は実行時検査を伴う関数

サイズの等しさしか検査しない！ ➔ 型付けが大幅に単純化

【疑問】具体的サイズ情報が失われても困らない？

➔ ライブラリを試作して，実際のアプリケーションプログラムで検証

生成的な幽霊型のみによる静的サイズ検査

型レベル自然数は演算が大変 ➔ 生成的な幽霊型のみを使う

- `val cons   : float -> 'n vec -> ~~'n s~~ vec`
  (`'n s` → `?`)
- `val tl_dyn : ~~'n s~~ vec -> ~~'n~~ vec`
  (`'n s` → `'n`, `'n` → `?`)

  ※ _dyn は実行時検査を伴う関数

サイズの等しさしか検査しない！ ➔ 型付けが大幅に単純化

【疑問】具体的サイズ情報が失われても困らない？

➔ ライブラリを試作して，実際のアプリケーションプログラムで検証

- 結論： 実用上は問題ない
- 理由： cons や tl_dyn のような関数の使用頻度は低い

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証
    - BLAS と LAPACK への静的サイズ検査の導入
    - 検証方法
    - 検証結果

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア 1：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア 2：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証
   - BLAS と LAPACK への静的サイズ検査の導入
   - 検証方法
   - 検証結果

BLAS と LAPACK

BLAS，LAPACK：

- Fortran の線形代数演算ライブラリ（様々な言語に移植されている）
- 高速 & 汎用的
- 数値計算で多用

本研究では BLAS と LAPACK に静的サイズ検査を導入し，
線形代数演算ライブラリを OCaml で試作した

- 多くの高水準な行列演算に本研究の方法で型付けできた
  - この後，型付けの例を紹介
- 一部，大小比較などを必要とする仕様が存在
  - 添字アクセスなど，一部の低水準な操作
  - オフセットとインクリメント幅
  - LAPACK 関数の作業領域サイズ
  - 大小関係に基づく仕様を含む関数 (orgqr, ormqr, syevr)

型付けの例 (1) — ベクトルの内積・加算

ベクトルの内積　※ 説明に無関係な引数は省略

```
val dot : x:vec -> vec -> float
```

```
val dot : x:'n vec -> 'n vec -> float
```

型付けの例 (1) — ベクトルの内積・加算

ベクトルの内積

※ 説明に無関係な引数は省略

```
val dot : x:vec -> vec -> float
```

```
val dot : x:'n vec -> 'n vec -> float
```

ベクトルのスカラー倍と加算 ($\boldsymbol{y} \leftarrow \alpha\boldsymbol{x} + \boldsymbol{y}$)

```
val axpy : ?alpha:float -> x:vec (* x *) -> vec (* y *) ->unit
```

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

型付けの例 (2) — 転置フラグ

一般行列同士の積

```
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 *)
```

$\boldsymbol{A}$ と $\boldsymbol{B}$ に非転置 (‘N)，転置 (‘T)，共役転置 (‘C) を指定できる：

- 例）transa=‘N，transb=‘T のとき，$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^\top + \beta \boldsymbol{C}$

型付けの例 (2) — 転置フラグ

一般行列同士の積

```
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 *)
```

A と B に非転置 (`N)，転置 (`T)，共役転置 (`C) を指定できる：

- 例）transa=`N，transb=`T のとき，$C \leftarrow \alpha A B^\top + \beta C$

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

val gemm : ... -> ?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 *)
```

型付けの例 (3) — 乗算方向フラグ

$k \times k$ 対称行列 $\boldsymbol{A}$ と $m \times n$ 一般行列 $\boldsymbol{B}$ の積

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

- side=‘L のとき, $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$ （ただし, $\boldsymbol{A}$ は $m \times m$ 行列）
- side=‘R のとき, $\boldsymbol{C} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A} + \beta \boldsymbol{C}$ （ただし, $\boldsymbol{A}$ は $n \times n$ 行列）

型付けの例 (3) — 乗算方向フラグ

$k \times k$ 対称行列 $\boldsymbol{A}$ と $m \times n$ 一般行列 $\boldsymbol{B}$ の積

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

- side=‘L のとき，$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$ （ただし，$\boldsymbol{A}$ は $m \times m$ 行列）
- side=‘R のとき，$\boldsymbol{C} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A} + \beta \boldsymbol{C}$ （ただし，$\boldsymbol{A}$ は $n \times n$ 行列）

```
type ('k, 'm, 'n) side (* = [ ‘N | ‘T | ‘C ] *)
val left  : ('m, 'm, 'n) trans (* = ‘L *)
val right : ('n, 'm, 'n) trans (* = ‘R *)

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

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア 1：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア 2：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証
    - BLAS と LAPACK への静的サイズ検査の導入
    - 検証方法
    - 検証結果

検証方法

※ 形式的検証ではありません

Lacaml — BLAS と LAPACK の OCaml バインディング
（静的サイズ検査なし）

Slap — 我々が試作した線形代数演算ライブラリ
（静的サイズ検査あり，Lacaml 互換インタフェース）

以下のアルゴリズムを Lacaml と Slap で実装してコードを比較
(`https://github.com/akabe/slap-lacaml`)

- LU 分解
- QR 分解
- 単純パーセプトロン
- K-means クラスタリング
- 主成分分析

アウトライン

1. 背景：型レベル自然数による静的サイズ検査
2. 本研究のアイディア１：生成的な幽霊型による静的サイズ検査
3. 本研究のアイディア２：生成的な幽霊型「のみ」による静的サイズ検査
4. 実装と検証
   - BLAS と LAPACK への静的サイズ検査の導入
   - 検証方法
   - 検証結果

検証結果

今回，検証に用いたアルゴリズムでは，
比較的小さい書き換えで済んだ：

- 添字アクセスの記法の変更：`x.{i,j}` ➔ `get_dyn`, `set_dyn`
- 転置フラグなどの書き換えが必要
- `int` 型と `'n size` 型を区別する必要がある

【結果】

- 各行列演算のサイズの整合性を等しさのみで検査する方法で，
  アプリケーションプログラム全体を検査することができた
  ※ 一部の低水準な操作や大小関係を除く
- プログラム記述の柔軟性の低下は問題にならなかった
- OCaml の型推論の恩恵を受けられる
  - 生成的幽霊型を実現するモジュールに関する型注釈を
    ユーザが書く必要はなかった

まとめ

- 生成的な幽霊型によるサイズ表現
  - ∃n. n vec のような存在限量化されたサイズつき型を表現
  - サイズの等しさのみを検査
- BLAS と LAPACK への型付け
  - 多くの高水準な行列演算を型付けできた
  - 一部，大小比較などが必要な仕様が存在
- アプリケーションプログラムの移植によるライブラリの有効性の検証
  - プログラムへの変更がほぼ不要

【課題】

- ライブラリの内部実装の静的サイズ検査
- 生成的な幽霊型を含む型システムの定式化
- 大小比較や自然数上の演算を含むサイズ検査の実現

など

アウトライン

5. Appendix
   - 型レベル自然数による静的サイズ検査のシグネチャ
   - サイズを第一級の値として扱うための単一型
   - 自然数上の演算を表す型演算子
   - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
   - プログラムの記述の柔軟性の低下
   - トップレベルの生成的幽霊型のファンクターによる実現
   - エスケープする生成的な幽霊型の実現
   - オフセットとインクリメント幅

5. Appendix
    - 型レベル自然数による静的サイズ検査のシグネチャ
    - サイズを第一級の値として扱うための単一型
    - 自然数上の演算を表す型演算子
    - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
    - プログラムの記述の柔軟性の低下
    - トップレベルの生成的幽霊型のファンクターによる実現
    - エスケープする生成的な幽霊型の実現
    - オフセットとインクリメント幅

型レベル自然数による静的なサイズ情報の表現

```
module Vec : sig
  type z       (* ゼロ *)
  type 'n s    (* 後者 (successor) *)
  type 'n vec (* ベクトル型 *)
  val empty : z vec
  val cons  : float -> 'n vec -> 'n s vec
  val hd    : 'n s vec -> float
  val tl    : 'n s vec -> 'n vec
  val dim   : 'n vec -> int (* ベクトルの次元を返す *)
  val add   : 'n vec -> 'n vec -> 'n vec (* ベクトルの加算 *)
end
```

アウトライン

5. Appendix
   - 型レベル自然数による静的サイズ検査のシグネチャ
   - サイズを第一級の値として扱うための単一型
   - 自然数上の演算を表す型演算子
   - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
   - プログラムの記述の柔軟性の低下
   - トップレベルの生成的幽霊型のファンクターによる実現
   - エスケープする生成的な幽霊型の実現
   - オフセットとインクリメント幅

リストとの変換

`of_list` には

```
val of_list : float list -> 'n vec
```

という型を与えたい．
戻り値の 'n には，リストのサイズに対応する型を代入したい

リストとの変換

of_list には

```
val of_list : float list -> 'n vec
```

という型を与えたい.
戻り値の 'n には, リストのサイズに対応する型を代入したい

サイズを表す型を受け取る引数を加えてみる：

```
val of_list_dyn : float list -> 'n vec -> 'n vec
let of_list_dyn l v =
  if List.length l = dim v then l else invalid_arg "..."
```

※ ***_dyn は実行時検査を伴う関数

リストとの変換

of_list には

```
val of_list : float list -> 'n vec
```

という型を与えたい．
戻り値の 'n には，リストのサイズに対応する型を代入したい

サイズを表す型を受け取る引数を加えてみる：

```
val of_list_dyn : float list -> 'n vec -> 'n vec
let of_list_dyn l v =
  if List.length l = dim v then l else invalid_arg "..."
```

※ ***_dyn は実行時検査を伴う関数

第 2 引数で必要とされているのはサイズ情報（ベクトルではない）

単一型 (singleton type)

サイズ自体を表す値とその型 `'n size` を導入する：

```
module Vec : sig
  type 'n size
  val size_to_int : 'n size -> int
  val dim         : 'n vec -> 'n size
  val of_list_dyn : float list -> 'n size -> 'n vec
  ...
end = struct
  type 'n size = int
  let size_to_int n   = n
  let dim             = List.length
  let of_list_dyn l n = if List.length l = n then l else ...
  ...
end
```

`'n size` は単一型 — α `size` 型の式の評価結果 = 型 α に対応する自然数

第一級サイズ値が有用である別な例

$m \times n$ 行列や $k \times l$ 行列から $m \times k$ 行列や $n \times k$ 行列を作ることを考える.

- 'n size 型を導入しない場合：

```
val of_list_dyn1 : float list list -> ('m, 'n) mat ->
                   ('k, 'l) mat -> ('m, 'k) mat
val of_list_dyn2 : float list list -> ('m, 'n) mat ->
                   ('k, 'l) mat -> ('n, 'k) mat
...
```

パラメータの全ての組合せが必要

- 'n size 型を導入する場合：

```
val dim1        : ('m, 'n) mat -> 'm size
val dim2        : ('m, 'n) mat -> 'n size
val of_list_dyn : float list list -> 'm size -> 'n size ->
                 ('m, 'n) mat
```

of_list_dyn は 1 つで十分

アウトライン

5. Appendix
   - 型レベル自然数による静的サイズ検査のシグネチャ
   - サイズを第一級の値として扱うための単一型
   - 自然数上の演算を表す型演算子
   - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
   - プログラムの記述の柔軟性の低下
   - トップレベルの生成的幽霊型のファンクターによる実現
   - エスケープする生成的な幽霊型の実現
   - オフセットとインクリメント幅

自然数上の演算を表す型演算子

- 生成的な幽霊型による append の型付け：

```
val append : 'm vec -> 'n vec -> ? vec
```

  - 自然数上の演算の結果をフレッシュな自然数と見なしている

自然数上の演算を表す型演算子

- 生成的な幽霊型による append の型付け：

```
val append : 'm vec -> 'n vec -> ? vec
```

  - 自然数上の演算の結果をフレッシュな自然数と見なしている

- もっと簡単に表現できる：

```
val append : 'm vec -> 'n vec -> ('m, 'n) add vec
```

  - 加算は決定的 (deterministic) な演算
  - 加算を単なる自由代数 (free algebra) と考える

これなら，OCaml の型システムで直接的に実現できる！

アウトライン

5 Appendix
- 型レベル自然数による静的サイズ検査のシグネチャ
- サイズを第一級の値として扱うための単一型
- 自然数上の演算を表す型演算子
- 生成的な幽霊型のみの型付けによる自然数の情報の喪失
- プログラムの記述の柔軟性の低下
- トップレベルの生成的幽霊型のファンクターによる実現
- エスケープする生成的な幽霊型の実現
- オフセットとインクリメント幅

自然数の情報が失われる

1. 大小関係を表現できない

2. 自然数上の計算を表現できない

自然数の情報が失われる

1. 大小関係を表現できない

【型レベル自然数】

```
val hd : ’n s vec -> float
val tl : ’n s vec -> ’n vec
```

2. 自然数上の計算を表現できない

自然数の情報が失われる

1. 大小関係を表現できない

【型レベル自然数】

```
val hd : 'n s vec -> float
val tl : 'n s vec -> 'n vec
```

【生成的な幽霊型】

```
val hd_dyn : 'n vec -> float
val tl_dyn : 'n vec -> ? vec
```

2. 自然数上の計算を表現できない

自然数の情報が失われる

1. 大小関係を表現できない

【型レベル自然数】

```
val hd : 'n s vec -> float
val tl : 'n s vec -> 'n vec
```

【生成的な幽霊型】

```
val hd_dyn : 'n vec -> float
val tl_dyn : 'n vec -> ? vec
```

2. 自然数上の計算を表現できない

```
val cons : float -> 'n vec -> ? vec

let (x : ?¹ vec) = ...
let (y : ?² vec) = cons 42.0 x
let (z : ?³ vec) = tl_dyn x
```

- ベクトル x と z のサイズは等しい
- しかし，型は異なる ($?^1 \neq ?^3$)

生成的な幽霊型のみに基づく静的サイズ検査

生成的な幽霊型 のみ で型付けを大幅に単純化

- 自然数の情報が失われる
- サイズの等しさしか検査しない

- 「型が等しければサイズが等しい」という健全性は失われないものと思われる
- アプリケーションプログラムの型付けには支障がない

5 Appendix
- 型レベル自然数による静的サイズ検査のシグネチャ
- サイズを第一級の値として扱うための単一型
- 自然数上の演算を表す型演算子
- 生成的な幽霊型のみの型付けによる自然数の情報の喪失
- プログラムの記述の柔軟性の低下
- トップレベルの生成的幽霊型のファンクターによる実現
- エスケープする生成的な幽霊型の実現
- オフセットとインクリメント幅

プログラムの記述の柔軟性の低下

- hd_dyn, tl_dyn は型エラーになりやすい：

```
let fold_left f e v =
  if int_of_size (dim v) = 0 then e else
  fold_left f (f e (hd_dyn v)) (tl_dyn v)
```

【理由】fold_left に型が付いたと仮定する．
ある型 t について v : t vec とすると，

- フレッシュな型 u について tl_dyn v : u vec なので，t = u
- しかし，フレッシュの定義より，t ≠ u

プログラムの記述の柔軟性の低下

- hd_dyn, tl_dyn は型エラーになりやすい：

```
let fold_left f e v =
  if int_of_size (dim v) = 0 then e else
  fold_left f (f e (hd_dyn v)) (tl_dyn v)
```

【理由】fold_left に型が付いたと仮定する.
ある型 t について v : t vec とすると,

  - フレッシュな型 u について tl_dyn v : u vec なので, t = u
  - しかし, フレッシュの定義より, t ≠ u

- get_dyn などで添字アクセスすると型エラーにならない：

```
let fold_left f e v =
  let rec loop e i =
    if i = int_of_size (dim v) then e else
    loop (f e (get_dyn v i)) (i + 1) in
  loop e 0
```

プログラムの記述の柔軟性の低下は問題にならない？

【予想】
`hd_dyn` や `tl_dyn` はあまり使わない？
【理由】

- `tl_dyn` 等の構造の分解を伴う操作はライブラリ側で提供
  - `map` や `fold_left` などのイテレータ関数
  - ブロック行列などを必要とする処理 など

  ➔ ユーザ側での `tl_dyn` 等の使用頻度が減る
- 要素へのアクセスについては，添字を使うことが多い

実際のアプリケーションプログラムで検証した結果，
「プログラムの記述の柔軟性の低下」は問題にならなかった．

アウトライン

5 Appendix
- 型レベル自然数による静的サイズ検査のシグネチャ
- サイズを第一級の値として扱うための単一型
- 自然数上の演算を表す型演算子
- 生成的な幽霊型のみの型付けによる自然数の情報の喪失
- プログラムの記述の柔軟性の低下
- トップレベルの生成的幽霊型のファンクターによる実現
- エスケープする生成的な幽霊型の実現
- オフセットとインクリメント幅

適用的ファンクターと生成的ファンクター

ファンクター — モジュールからモジュールへの写像

例）型 t 上の順序関係を受け取り，
順序集合のモジュールを返すファンクター

```
module MakeOrdSet
  (X : sig type t val cmp : t -> t -> int end) : sig
    type t (* 順序集合の型 *)
    type elt = X.t

    val empty : t
    val insert : elt -> t -> t
    ...
  end = struct ... end
```

適用的ファンクターと生成的ファンクター

```
module MakeOrdSet (X : sig type t val cmp : t -> t -> int end)
  : sig type t ... end

module OrdInt = struct type t = int val cmp = ... end
module M1 = MakeOrdSet(OrdInt)
module M2 = MakeOrdSet(OrdInt)
```

【疑問】M1 と M2 は同じモジュールと見なすべき？ (M1.t = M2.t ?)

適用的ファンクターと生成的ファンクター

```
module MakeOrdSet (X : sig type t val cmp : t -> t -> int end)
  : sig type t ... end

module OrdInt = struct type t = int val cmp = ... end
module M1 = MakeOrdSet(OrdInt)
module M2 = MakeOrdSet(OrdInt)
```

【疑問】M1 と M2 は同じモジュールと見なすべき？ (M1.t = M2.t ?)

- OCaml では M1.t = M2.t
  - モジュール上の式は全く同じなので，M1.t = M2.t と見なしても健全
  - 適用的 (applicative) ファンクターという

適用的ファンクターと生成的ファンクター

```
module MakeOrdSet (X : sig type t val cmp : t -> t -> int end)
  : sig type t ... end

module OrdInt = struct type t = int val cmp = ... end
module M1 = MakeOrdSet(OrdInt)
module M2 = MakeOrdSet(OrdInt)
```

【疑問】M1 と M2 は同じモジュールと見なすべき？ (M1.t = M2.t ?)

- OCaml では M1.t = M2.t
  - モジュール上の式は全く同じなので，M1.t = M2.t と見なしても健全
  - 適用的 (applicative) ファンクターという
- SML では M1.t ≠ M2.t
  - M1 と M2 は同じ用途とは限らず，意味は違うかもしれない
    - 例）M1 は金額の集合，M2 は年齢の集合
  - 生成的 (generative) ファンクターという

OCaml のファンクターの振る舞い

```
module F (S : sig end) : sig type t end = struct type t end
module X = sig end
module Y = X

module A = F(X)
module B = F(X)
module C = F(Y)
module D = F(struct end)
module E = F(struct end)
```

OCaml ではファンクターの引数が同じ名前で束縛されている場合に限り同じモジュールと見なされる：

- A.t = B.t
- A.t ≠ C.t ≠ D.t ≠ E.t

つまり，名前で束縛 しなければ，生成的な幽霊型を作れる！
（ただし，トップレベル）

ローカルスコープを持つ生成的幽霊型の利用例

リストからベクトルへの変換：

```
module Vec : sig
  type 'n vec
  module Of_list (X : sig val value : float list end) :
    sig type t val value : t vec end
end = struct
  type 'n vec = float list
  module Of_list (X : sig val value : float list end) =
    struct type t let value = X.value end
end

open Vec

let foo () =
  let module X = Of_list(struct let value = [1.;2.;3.] end) in
  let module Y = Of_list(struct let value = [4.;5.] end) in
  Vec.add X.value Y.value (* 型エラー (X.t ≠ Y.t) *)
```

5. Appendix
   - 型レベル自然数による静的サイズ検査のシグネチャ
   - サイズを第一級の値として扱うための単一型
   - 自然数上の演算を表す型演算子
   - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
   - プログラムの記述の柔軟性の低下
   - トップレベルの生成的幽霊型のファンクターによる実現
   - エスケープする生成的な幽霊型の実現
   - オフセットとインクリメント幅

エスケープする生成的な幽霊型の実現

```
module type S = sig
  type t             (* ? (生成的な幽霊型) *)
  val value : t vec (* val value : ? vec *)
end
module F (X : sig end) : S = struct ... end
```

生成的な幽霊型を持つベクトルは直接返せない（コンパイルエラー）：

```
let f () =
  let module Y = F(struct end) in Y.value
```

生成的な幽霊型を含むモジュールごと返せば良い：

```
let g () =
  let module Y = F(struct end) in (Y : S)

let main () =
  let module Y' = (val g () : S) in
  ...
```

アウトライン

5. Appendix
   - 型レベル自然数による静的サイズ検査のシグネチャ
   - サイズを第一級の値として扱うための単一型
   - 自然数上の演算を表す型演算子
   - 生成的な幽霊型のみの型付けによる自然数の情報の喪失
   - プログラムの記述の柔軟性の低下
   - トップレベルの生成的幽霊型のファンクターによる実現
   - エスケープする生成的な幽霊型の実現
   - オフセットとインクリメント幅

オフセットとインクリメント幅

ベクトルの内積 (Lacaml)：

```
val dot : ?n:int ->
          ?ofsx:int -> ?incx:int -> x:vec ->
          ?ofsy:int -> ?incy:int -> vec -> float
```

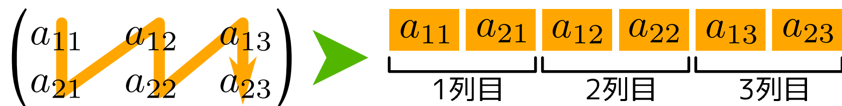
dot の計算式：

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

$\mathtt{x}[k]$ — ベクトル x の k 番目の要素 $(k \geq 1)$

オフセットとインクリメント幅の使い所

BLAS, LAPACK では行列を column-major 配置の 1 次元配列で表現



オフセット・インクリメント幅を指定すると，
列・行などをコピーせずにベクトルとして扱える．

例）$m \times n$ 行列に対して，

- i 番目の列：ofsx $= mi$, incx $= 1$, n $= m$
- i 番目の行：ofsx $= i$, incx $= m$, n $= n$

  etc...

※ x には行列を格納した 1 次元配列を与える．

オフセットとインクリメント幅の検査

オーバーランが起きないためには

$$1 \le \mathtt{ofsx} \le \mathtt{dim}(\mathtt{x}) \wedge 1 \le \mathtt{ofsx} + (\mathtt{n} - 1)\mathtt{incx} \le \mathtt{dim}(\mathtt{x})$$

が成立する必要がある.

これは明らかに等しさの判定だけでは検査できないので,
オフセット・インクリメント幅を用いない形式で実装：

```
val dot : x:'n vec -> 'n vec -> float
```