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

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

概 要　線形代数演算（行列演算）は数値計算の中心的分野の一つであり，幅広く応用されている．様々なプログラミング言語上の線形代数演算ライブラリが存在するが，それらのほとんどは，行列（およびベクトル）のサイズの整合性を静的に保証しておらず，実行時エラー（および不具合）の原因となっている．配列等のサイズの整合性を静的に保証する既存研究としては，自然数上の依存型を用いる Dependent ML [1] や sized type [2] 等があるが，これらは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を必要とする．それに対し，我々は自然数上の依存型ではなく，生成的 (generative) な幽霊型 (phantom type) をパラメータとする多相型を用いることにより，ほぼ標準的な ML 型とモジュールシステムのみで，サイズの整合性を静的に保証する線形代数演算ライブラリを提案する．そのようなライブラリを関数型プログラミング言語 OCaml で試作し，線形代数および機械学習のいくつかのアルゴリズムを記述したところ，アプリケーションプログラムへの変更はほぼ不要であることが確認できた．

1　序論

1.1　背景

線形代数演算，特に行列演算は数値計算の中心的分野の一つであり，画像処理，信号処理，機械学習等をはじめとした多くの分野で利用されている．それらを実装するために，行列演算を組み込み機能としてサポートした数値計算言語や，様々なプログラミング言語上の線形代数演算ライブラリが存在する．数値計算言語としては MatLab や統計処理言語 S が有名であり，関数型プログラミング言語においても Scala に埋め込まれた機械学習用のドメイン固有言語 OptiML [3, 4] が存在する．一方，線形代数演算ライブラリとしては BLAS，LAPACK などが有名である．

しかし，数値計算言語や線形代数演算ライブラリの多くは，ベクトルおよび行列のサイズの整合性を静的に保証していない．そのため，例えば 3×5 行列と 5×3 行列の加算など，サイズの合わない行列演算は，例外やメモリ破壊など，実行時エラーおよび不具合の原因となる．

サイズに関する不整合が起こらないことを静的に保証できれば，バグの早期発見の他に，実行時サイズ検査の除去による速度向上も期待できる．配列等のサイズの整合性を静的に保証する既存研究としては，自然数上の依存型を用いる Dependent ML [1] や ATS [5]，sized type [2] 等が存在する．しかし，これらは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を必要とする．

1.2　本研究の貢献

本研究では，自然数上の依存型のかわりに，生成的 (generative) な幽霊型 (phantom type) をパラメータとする多相型を用いることにより，以下のような特徴を備えた線形代数演算ライブラリを提案する．

- ほぼ標準的な ML の型システムとモジュールシステムのみで実現できる．
- 行列（およびベクトル）の加算や乗算をはじめ，ほとんどの高水準な線形代数演算について，サイズの整合性が静的に保証される．（行と列を指定した要素アクセスなど，一部の低水準な操作は実行時の検査が必要となる．）
- 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である．

さらに，提案の有効性を確認するため，実際に関数型プログラミング言語 OCaml 上でライブラリを試作する．また，線形代数および機械学習の代表的アルゴリズムを実装した，いくつかのプログラムを既存の（サイズの整合性を静的に保証していない）ライブラリから移植する．

1.3 本論文の構成

これ以降の本論文の構成は次の通りである．2 節では本研究の提案する静的サイズ検査の方法を，例を通して概説する．3 節では OCaml 上で試作した，具体的なライブラリの実装について詳しく述べる．4 節ではそのライブラリを用いてアプリケーションプログラムを記述した結果を考察する．5 節では関連研究を議論し，6 節では結論として今後の課題を述べる．

2 本研究のアイディア

2.1 型レベル自然数による静的サイズ検査

本節では次節の説明の準備として，行列やベクトルのサイズを（生成的な幽霊型ではなく）型レベルの自然数で表すことにより静的に検査する基本的な方法を説明し，その問題点を述べる．

`'n vec` をサイズ（次元）が `'n` のベクトルの型，`('m,'n) mat` を `'m` 行 `'n` 列の行列の型とし，型パラメータ `'m`，`'n` にサイズを表す型を代入することにする．ベクトルと行列におけるサイズ情報の表現は，パラメータの個数以外は同じであるため，これ以降はベクトルを例として説明する．

2.1.1 型による自然数の表現

まず，ベクトルや行列のサイズを表す自然数を，型で表現する方法について考える．ここではゼロに対応する型 `z` と後者 (successor) に対応する型 `'n s` の 2 つを用いることにする．例えば，`z s s vec` は 2 次元ベクトルの型であり，`z s s s vec` は 3 次元ベクトルの型である（ML の型演算子適用は左結合的であることに注意する）．

ベクトル型 `'n vec` の実装は，リストや配列などベクトルを表現しうるデータ構造であればどのようなものでも良く，各要素は mutable でも immutable でも構わない．ここでは具体的に説明する便宜上，次のように実装されているものとする．

```
type 'n vec = float list
```

しかし，`'n vec` と `float list`（あるいは他の実装）が等しいことがライブラリの外側（ユーザ）から見えてしまうと，型パラメータ `'n` は型定義の右辺には現れないため，型パラメータに代入されたサイズ情報が型検査器に無視されてしまう．サイズの異なるベクトルの型（例えば `z s s vec` と `z s s s vec`）が等しいものとして扱われてはサイズの整合性を検査できないので，`'n vec` の実装は（ライブラリを実装するモジュールのシグネチャ等で）隠蔽する必要がある．

サイズを表現するための 2 つの型 `z`，`'n s` はシグネチャで隠蔽した幽霊型として定義する．本論文における幽霊型とは，型定義の右辺に現れない型変数，およびそのような型変数に代入される（しばしば値を持たない）型 [6] である．尚，そのような型変数を持つ型全体 [7] や，GADT (generalized algebraic data type) を幽霊型と呼ぶ人 [8] もいる．

ソースコード 1 にこれまで議論した `'n vec` と `z`，`'n s` の型の定義に加えて，ベクトルに関するいくつかの関数の定義を示す．

ソースコード 1. 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
  type 'n s
  type 'n vec
  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               (* ベクトルのサイズを調べる *)
  val add   : 'n vec -> 'n vec -> 'n vec  (* ベクトルの加算 *)
end = struct
  type z
  type 'n s
  type 'n vec = float list
  let empty    = []
  let cons x l = x :: l
  let hd       = List.hd
  let tl       = List.tl
  let dim      = List.length
  let add      = List.map2 (+.)
end
```

ソースコード 1 において重要なことは，モジュールの実装ではなく，シグネチャにおける型注釈である．型注釈はサイズに関する情報を含んでおり，`empty` は 0 次元ベクトルであること，`cons` は実数と n 次元ベクトルを受け取って $n+1$ 次元ベクトルを返すことを表現している．また，`add` は 2 つの実ベクトルの加算を行う関数であり，n 次元ベクトルを 2 つ受け取って n 次元ベクトルを返す．

ベクトルに `cons` を適用する度に `'n vec` の型パラメータ `'n` に `s` が追加されるため，次のようにベクトルのサイズ情報が型に反映される．

```
let a = Vec.cons 1.0 (Vec.cons 2.0 Vec.empty)                (* : z s s vec *)
let b = Vec.cons 3.0 a                                       (* : z s s s vec *)
let c = Vec.cons 4.0 (Vec.cons 5.0 (Vec.cons 6.0 Vec.empty)) (* : z s s s vec *)
```

内部の実装はシグネチャで隠蔽しているため，型検査器は `a` と `b` を違う型として扱う．そのため，式 `Vec.add a b` のように異なるサイズのベクトルを加算しようとすると，型が合わず静的エラーとなる．それに対して `b` と `c` のように同じサイズのベクトルは同じ型になるので，式 `Vec.add b c` は型エラーにはならない．

2.1.2 サイズを第一級の値として扱うための単一型

前節ではベクトルのサイズの自然数を型で表したが，サイズを（第一級の）値として扱いたい場合もある．例えばリストをベクトルに変換する関数 `of_list` を考えると，`float list -> 'n vec` のような型を与えたい．すると，戻り値の型 `'n vec` の型パラメータ `'n` には，引数のリストのサイズに対応する型を代入しなければならないが，リストの型にはサイズの情報が含まれていないため，それは不可能であるように思われる．

そこで次のように，サイズを表す型`'n` を与えるために，`'n vec` 型の引数を加えることを考えてみる．

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

実際に戻り値になるのは第一引数のリストであり，第二引数は戻り値の型に含めるサイズ情報`'n` だけが用いられる．

ただし，`'n` の表すサイズと，戻り値のベクトルの実際のサイズは等しくなければならない．そのため `of_list_dyn` の実装では，次のように戻り値のサイズと第二引数のサイズが等しいことを実行時に保証する必要がある．

```
let of_list_dyn l v =
  if List.length l = dim v then l else invalid_arg "Vec.of_list_dyn"
```

本論文では，このような実行時検査を伴う関数の名前には，慣習として接尾辞_dynをつけることとする．

of_list_dynの第二引数で必要とされている情報はあくまでサイズだけでありベクトルではないので，ベクトルではなくサイズそのものを表現する値を第二引数として用いる方が自然である．そこでサイズそのものを表す値とその型'n sizeをソースコード2のように導入する．

ソースコード 2. 単一型の導入（ソースコード1との差分のみ）

```
1  module Vec : sig
2    type 'n size
3    val int_of_size : 'n size -> int (* 'n size 型の値を整数値に変換する関数 *)
4    val dim         : 'n vec -> 'n size
5    val of_list_dyn : float list -> 'n size -> 'n vec
6    ...
7  end = struct
8    type 'n size = int
9    let int_of_size n   = n
10   let dim             = List.length
11   let of_list_dyn l n = if List.length l = n then l else invalid_arg "Vec.of_list_dyn"
12   ...
13 end
```

'n sizeの型パラメータ'nには，zや'n sで表現されたサイズ情報が代入され，α size型の式の評価結果は必ず型 α に対応する自然数になる．'n sizeは単一集合に対応する型であり，単一型(singleton type)と呼ばれる．'n sizeの導入により，of_list_dynの第二引数には，直接ベクトルを渡す代わりに，dimで取得したサイズ情報を渡すことになる．

'n size型を持つ第一級のサイズ値が有用な別の例として，$m \times n$ 行列と $k \times l$ 行列があり，これらのサイズ情報をもとに $m \times k$ や $n \times k$ の行列を作ることを考える．'n sizeを使わない場合

```
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
...
```

というように全ての組み合わせを用意する必要がある．しかし，'n sizeを導入すれば，行列に対しても次のようなサイズを調べる関数を用意することで，列と行のサイズ情報を分解できる．

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

したがって，次のような関数が1つあれば十分である．

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

2.1.3 問題点

このように，'n vecや'n sizeの型パラメータにサイズの情報を埋め込むことで，行列演算のサイズの整合性の検査を型検査として行うことができる．しかし，この方法には次の2つの問題点が存在する．

1. サイズの整合性が保証されるのはモジュールの外側，つまり線形代数演算ライブラリのユーザが記述したアプリケーションプログラムのみである．モジュールの内側（実装）では'n vec = float listや'n size = intという定義が見えているので，サイズの整合性は検査されない．したがって，ライブラリの開発者は型注釈に違反しないように関数（や定数）を実装しなければなら

ない．ライブラリの内部実装の静的サイズ検査については，依存型や GADT で実現できるが，本研究では対象としない．

2. すべての行列やベクトルのサイズが静的に決定されるプログラムでは，本節のような型付けによりサイズの整合性を静的に検査できるが，一般に行列やベクトルのサイズは実行時まで決まらないことも多い．このような場合，本節の方法では型付けすることができない．この問題の解決については次節で述べる．

2.2 生成的な幽霊型による静的サイズ検査

本節では，行列などのサイズが動的に決まる場合の型付けについて考える．ただし，動的にサイズが決まる場合についても，個々のベクトルや行列において実行時に一度決定されたサイズは，それ以後変化しないものとする．即ち，各ベクトル・行列はサイズに関して immutable になるように実装されているものとする．各ベクトル・行列のサイズが実行時に変化すると，要素のコピー等が必要になりオーバーヘッドが大きいので避けられるのが一般的であり，これは妥当な仮定である．

2.2.1 静的に不明なサイズを表す生成的な幽霊型

動的にサイズが決まるベクトルを返す関数の具体例として，ファイルからベクトルを読み込む関数 `loadvec` を考える．この関数は文字列（ファイル名）を受け取り，何らかのサイズのベクトルを返すので，型は `string -> ? vec` となる．今興味があるのは `?` に当てはまる型がどのように表されるのかということである．ここで，`loadvec` が存在したと仮定して，次のようなコードについて考えてみる．

```
1 let (x : ?[1] vec) = loadvec "file1" in
2 let (y : ?[2] vec) = loadvec "file2" in
3 add x y
```

異なるファイルから読み込まれたベクトル `x`，`y` のサイズが等しいとは限らないので，3 行目は型エラーを起こすべきである．さらに，1 行目と 2 行目のファイル名が同一である場合でも，1 行目を実行してから 2 行目を実行するまでの間にファイルの内容が更新される可能性がある．この場合も `x` と `y` のサイズが等しいとは限らないので，やはり 3 行目は型エラーを起こすべきである．したがって，`loadvec` の戻り値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である．そこで，`loadvec` の戻り値の型 `'n vec` の型パラメータ `'n` に代入されるのは生成的な幽霊型であり，関数は呼ばれる度にフレッシュな型の値を返すものとみなす.（このように，値が作られるたびに，新たに与えられる型のことを生成的な型という.）

生成的な幽霊型によるサイズの表現は，$\exists$`n. n vec` のような存在限量化されたサイズつき型に相当する．動的にサイズが決まる場合，コンパイル時には実行時のベクトルのサイズを知ることができないため，具体的なサイズの情報を捨て，他のいずれのベクトルとも異なるサイズであると仮定している．

ただし，OCaml では戻り値の型が生成的であるような関数を書くことはできない．これは OCaml の言語仕様上の制限であり，解決策については 3.1 節で議論する．ここではさしあたり OCaml に戻り値の型が生成的であるような関数を書く機能を追加した仮想的なプログラミング言語で説明する．生成的な幽霊型は `?` で表し，一つのプログラム例の中に異なる複数の `?` が存在する場合は，区別のために `?`[1] のように添字を書くことにする．

2.2.2 生成的な幽霊型のみに基づく静的サイズ検査

前節では動的にしかわからないベクトルや行列のサイズを，生成的な幽霊型により表した．逆に，サイズが静的にわかる場合であっても，具体的な自然数の情報を捨ててしまい，生成的な幽霊型により保守的にサイズを表すことにより，型付けを単純化することが考えられる．本節ではそのよう

な単純化について述べる（この単純化によりアプリケーションプログラム例の型付けに支障がないことは 4 節で検証する）.

型 z, 'n s の代わりに生成的な幽霊型を使うと，例えば cons の型注釈は次のように書くことができる.

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

上の cons がソースコード 1 のものと違う点は，戻り値のベクトルのサイズが第二引数のベクトルのサイズよりも 1 つ大きいという情報を含んでおらず，ただ単に新たなサイズであることしか表現していないことである.

生成的な幽霊型は具体的なサイズの値を隠してしまうため，サイズ情報の表現に生成的な幽霊型を用いると，ベクトルや行列のサイズについて自然数としての情報が失われる．この事実は次の 2 つの点に表れている.

1. 生成的な幽霊型は大きさに関する情報を持たないので大小関係が失われる．例えば，hd と tl の型を生成的な幽霊型を使って表すと，それぞれ次の hd_dyn と tl_dyn のようになる.

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

   ソースコード 1 では 1 以上のサイズのベクトルを受け取ることを型注釈で表していたが，上の型注釈ではそのような情報を含んでいない.

2. 型の上で自然数の計算に相当する操作を行うことができない．任意のサイズのベクトル x について式 tl_dyn (cons 42.0 x) の評価結果のベクトルは x と同じサイズであるが，両者は同じ型ではない．即ち，$n+1-1$ の計算結果が n に等しいことを型の上で判定することはできない．計算の結果は常にフレッシュな型で表現されるため，どのような計算を行なっても既に存在する型に等しくなることはない.

したがって，型の上ではサイズの等しさしか検査できなくなる．しかし，型の上で自然数を表現できなくなっても，モジュールの実装が型注釈に違反しないと仮定すれば,「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考えている．ただし，その逆は成り立たず，実行時のサイズが等しいとしても型の上で等しいとは限らない.

別の例として，m 次元ベクトルと n 次元ベクトルを結合する関数 append は $(m+n)$ 次元ベクトルを返す．この振る舞いを型注釈として表現するためには OCaml で型上の加算を実現する必要があるが，我々の方法では自然数上の計算を表現することはできない．しかし，append は単に新たなサイズのベクトルを返すと考えて，'m vec -> 'n vec -> ? vec という型注釈で，その振る舞いを近似することができる.

2.2.3 自然数上の演算を表す型演算子

上述の通り，自然数上の演算の結果はすべてまったく新しい（フレッシュな）自然数である，とみなした型付けも可能だが，例えば append の型に含まれる加算等の決定的 (deterministic) な演算については，単なる自由代数 (free algebra) と考えることにより，次のように表現することもできる.

```
type ('m, 'n) add (* 隠蔽された幽霊型 *)
val append : 'm vec -> 'n vec -> ('m, 'n) add vec
```

生成的な幽霊型を ('m,'n) add で置き換えただけであるが，生成的な幽霊型を使っていないので，この型注釈は通常の OCaml の型システムで直接的に実現できる．('m,'n) add は他のいかなる型とも異なる型であることを表現しており，字面上の違いのみが重要である．append において戻り値の型が生成的でなくても良いのは，引数のベクトルのサイズ（つまり引数の型）が決まると，戻り値のベクトルのサイズはだだ 1 つに決定されるためである．同様に，cons や tl_dyn の型注釈も生成的な幽霊型を用いずに記述できる.

3 実装

3.1 生成的な幽霊型の実現

3.1.1 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

ML のモジュールシステムにおけるファンクターとは，モジュールからモジュールへの写像である．例えば，型 t 上の順序関係を定めたモジュールを受け取って，順序集合に関する操作を提供するモジュールを返すファンクター MakeOrdSet について考えてみる．

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

次のように整数上の順序関係 OrdInt から 2 つのモジュール M1 と M2 を作ったとする．

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

M1 と M2 を互換性のあるモジュールと見なすべきか，すなわち M1.t = M2.t として扱うべきかという問題について，異なる 2 つの考え方が存在する．

- OCaml では M1.t = M2.t と見なす．M1 と M2 を生成したモジュール上の式は全く同じであるから，M1.t と M2.t を同じ型と見なしても健全である．この場合ファンクターが純粋関数的に振る舞うので適用的 (applicative) ファンクターという．
- 一方，SML では M1.t ≠ M2.t と見なす．なぜならば，プログラム中で M1 と M2 を同じ用途で使っているとは限らず，プログラマは 2 つのモジュールに異なる意味を与えているかもしれない．したがって，M1.t と M2.t が混同して使用されるのを防ぐために M1.t ≠ M2.t として扱う．この場合フレッシュなモジュールが生成されるので生成的 (generative) ファンクターという．

ただし，OCaml においてファンクターにより得られたモジュールに互換性があると見なされるのは，引数のモジュールを名前に束縛していて，なおかつその名前が一致する時に限られる．例えば，次のようなプログラムを考えてみる．

```
1 module F (S : sig end) : sig type t end = struct type t end
2 module X = sig end
3 module Y = X
4 module A = F(X)
5 module B = F(X)
6 module C = F(Y)
7 module D = F(struct end)
8 module E = F(struct end)
```

A と B は同じ名前のモジュール X に F を適用して得られたので，A.t = B.t である．しかし，C は X とは異なる名前の引数 Y に F を適用して得られたので，A.t ≠ C.t である．また，D と E に至っては引数を名前に束縛していないので，A.t ≠ D.t ≠ E.t である．このように，OCaml のファンクターは引数を名前に束縛しなければ生成的な結果を返すので，この振る舞いを利用して生成的な幽霊型を作ることができる．

3.1.2 ローカルスコープを持つ生成的幽霊型のローカルモジュールによる実現

OCaml では，モジュールを局所的に束縛するローカルモジュールの機能がある．ローカルモジュールを使うことで，次のようにローカルなスコープを持つ生成的な幽霊型を作ることができる．

```
let main () =
  let module M = F(struct end) in
  let module N = F(struct end) in
  ... (* M.t ≠ N.t *)
```

OCaml におけるファンクターの振る舞いとローカルモジュールの機能を利用することで，生成的な幽霊型でサイズを表現したベクトルを，ローカルに作ることができるようになる．この具体例として，リストからベクトルへ変換するプログラムをソースコード 3 に示す．

ソースコード **3.** リストからベクトルへの変換（ソースコード 2 との差分のみ）

```
module Vec : sig
  ...
  module Of_list (X : sig val value : float list end) :
    sig type t val value : t vec end
end = struct
  ...
  module Of_list (X : sig val value : float list end) =
    struct type t let value = X.value end
end

let foo () =
  let module X = Vec.Of_list(struct let value = [1.; 2.; 3.; 4.] end) in
  let module Y = Vec.Of_list(struct let value = [5.; 6.; 7.] end) in
  Vec.add X.value Y.value (* 型エラー *)
```

`foo` 関数の中で作られている `X.t` と `Y.t` は共に生成的な幽霊型であり `X.t` ≠ `Y.t` である．したがって，式 `Vec.add X.value Y.value` は型エラーになる．`Of_list` には動的にサイズが決定されるリストも渡すことができるので，ファイルから読み込んだリストなどを渡しても安全である．

リストからベクトルへ変換する方法としては，2.1.2 節で `of_list_dyn` 関数も挙げたので，`Of_list` ファンクターとの違いについて述べておく．`of_list_dyn` は既存のサイズ情報からベクトルを作る場合に使用する．しかし，渡されたリストの実際の長さと，引数に指定されたサイズに，矛盾が生じる場合は実行時エラーになる．一方，`Of_list` は新しくサイズ情報を生成するので，サイズに関して実行時エラーを発生させることはない[*1]．ただし後述するが，生成的な幽霊型を含むベクトルを戻り値にする場合は少し手間が生じる．どちらも有用であるので，我々のライブラリでは両方とも実装している．

3.1.3 参考：第一級モジュールによる生成的幽霊型の実現

第一級モジュール (first-class module) は OCaml 3.12 から導入された機能であり，モジュールを値として扱うことができる．ファンクターとローカルモジュールの代わりに，第一級モジュールの機能を用いて，生成的な幽霊型を ∃`t`．`t vec` のような存在型として実現することもできる．具体例として第一級モジュールを用いて，リストをベクトルに変換する処理を記述したものをソースコード 4 に示す．

ソースコード **4.** 第一級モジュールを用いたリストからベクトルへの変換

```
module Vec : sig
  ...
  module type S = sig type t val value : t vec end
  val of_list : float list -> (module S)
end = struct
  ...
  module type S = sig type t val value : t vec end
```

[*1] ただし根本的にデータ構造が不正である場合，例えばリストから行列への変換でリストが矩形でない場合などは実行時エラーを発生させる．

```
8   let of_list (l : float list) =
9     let module M = struct type t let value = l end in
10    (module M : S)
11 end
12
13 let foo () =
14   let module X = (val Vec.of_list [1.; 2.; 3.; 4.] : Vec.S) in
15   let module Y = (val Vec.of_list [5.; 6.; 7.]     : Vec.S) in
16   Vec.add X.value Y.value (* 型エラー *)
```

ソースコード 4 はソースコード 3 と同じ処理を実現している．ここで，Of_list ファンクターや of_list 関数を呼び出す側（ライブラリのユーザ）の視点で，両者の特徴について比較してみる．

ソースコード 3 ではファンクターの呼び出し側で，引数となるリストを含むモジュール（**struct ... end**）を作る必要があるが，ソースコード 4 では関数に直接リストを渡すことができる．したがって，引数の点では，ソースコード 4 の方がより簡潔である．

一方，ソースコード 3 ではファンクターの戻り値を直接束縛することができるが，ソースコード 4 では戻り値のモジュールの型注釈を記述する必要がある．したがって，戻り値の点では，ソースコード 3 の方がより簡潔である．

本研究では型注釈を減らすため，ファンクターを用いた方を採用する．

3.1.4 エスケープする生成的幽霊型の実現

生成的な幽霊型を直接戻り値にするような関数は OCaml では記述することができない．例えば，ライブラリのユーザが次のような関数 f を書くとコンパイルエラーになる．

```
let f l = let module X = Vec.Of_list(struct let value = l end) in X.value
```

これは OCaml の言語仕様による制限である．この問題に対して，ユーザは 2 通りの対応が考えられる．

第一の方法は，戻り値のベクトルのサイズ情報を，関数の引数として受け取るやり方である．このアプローチでは元の関数の戻り値の型が ? vec である場合に，戻り値の型を 'n vec に書き換え，'n size や 'n vec などの型の引数を追加する．この引数は戻り値に含めるべき生成的な幽霊型を受け取るためのものであり，生成的な幽霊型は関数の外側へ追い出される．これは 2.1.2 節で of_list_dyn に 'n size 型の引数を導入したときの考え方と同じである．この方法ではユーザが型注釈を書く必要がなく OCaml の型推論の恩恵が得られる．

第二の方法はユーザが前節のような，存在型を表す第一級モジュールを用いる方法である．このアプローチではライブラリで作られた生成的な幽霊型を，ユーザ関数の戻り値に含めることができるが，モジュールの型（シグネチャ）は型推論されないので，機械的なものではあるが型注釈を書く必要がある．

3.2 BLAS と LAPACK への静的サイズ検査の導入

本研究のアイディアの有効性を確認するため，我々は BLAS と LAPACK に対し，これまで述べてきた方法で静的サイズ検査を導入した線形代数演算ライブラリを試作した．BLAS (Basic Linear Algebra Subprograms) [9] はベクトルや行列の加算や積などの基本的な線形代数処理を提供するライブラリであり，LAPACK (Linear Algebra PACKage) [10] は最小二乗法や LU 分解，固有値分解など，より高度な数値解析ルーチンを提供するライブラリである．これら 2 つのライブラリは最適化されたサードパーティ製の実装が数多く存在し，インタフェースも汎用的であることから数値計算で多用される．サードパーティ実装としては，オープンソースのものでは ATLAS が，移植性の高い最適化 BLAS 実装として有名である．また，Intel Math Kernel Library や AMD Core Math

Library など，最適化された BLAS および LAPACK の実装をハードウェアベンダが提供していることもある．

これらは元々 Fortran で実装されていたライブラリだが様々な言語に移植されており，OCaml にも Lacaml [11] という BLAS と LAPACK のバインディングが存在する．本研究では Lacaml にこれまで説明してきた方法で型注釈を付け，静的サイズ検査を導入した．本節では BLAS，LAPACK のインタフェースに型注釈を付ける上で工夫が必要になった点について述べる．

3.2.1 行列の転置フラグ

BLAS，LAPACK では，演算対象の行列を転置するか否かを表すフラグを，引数として指定する．例えば，一般行列の積を計算する gemm 関数の型を次に示す[*2]．

```
val gemm : ?m:int -> ?n:int -> ?k:int ->
      ?beta:num_type -> ?cr:int -> ?cc:int -> ?c:mat (* C *) ->
      ?transa:[‘N|‘T|‘C] -> ?alpha:num_type -> ?ar:int -> ?ac:int -> mat (* A *) ->
      ?transb:[‘N|‘T|‘C] -> ?br:int -> ?bc:int -> mat (* B *) -> mat (* C *)
```

gemm は基本的には $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$ という演算を行うが，transa と transb でそれぞれ $\boldsymbol{A}$ と $\boldsymbol{B}$ の非転置 (‘N)，転置 (‘T)，共役転置 (‘C) を指定することができる．例えば，transa=‘T, transb=‘N なら $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^\top\boldsymbol{B} + \beta\boldsymbol{C}$ という演算を行い，transa=‘C, transb=‘T なら $\boldsymbol{C} \leftarrow \alpha\overline{\boldsymbol{A}}^\top\boldsymbol{B}^\top + \beta\boldsymbol{C}$ という演算を行う．

$m \times n$ 行列の転置は $n \times m$ 行列というようにフラグの値に依存して行列のサイズ（型）が変化する．したがって，引数として渡されたフラグの値を型に反映させる必要がある．最も安直な方法は引数 transa，transb の値ごとに別々の関数を作ることである．

```
val gemm_NN : ...-> ?c:('m,'k) mat ->...-> ('m,'n) mat ->...-> ('n,'k) mat -> ('m,'k) mat
val gemm_TN : ...-> ?c:('m,'k) mat ->...-> ('n,'m) mat ->...-> ('n,'k) mat -> ('m,'k) mat
val gemm_TC : ...-> ?c:('m,'k) mat ->...-> ('n,'m) mat ->...-> ('k,'n) mat -> ('m,'k) mat
...
```

この方法は簡単で柔軟性が高いがフラグの組合せの数だけ関数ができてしまい煩雑である．

他に，転置などの操作に伴う型の変化を表現した関数型をパラメータとする，多相型を用いる方法がある．この方法は BLAS のインタフェースにより近く，フラグを引数として与えることができる．転置フラグの型と値は次のように表現される．

```
type 'a trans = [‘N | ‘T | ‘C] (* 実装はシグネチャで隠す *)
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 *)
```

この場合，gemm に次のような型注釈を書き，transa，transb に normal，trans，conjtr のいずれかを渡すことで，フラグに依存したサイズの変化を型に反映させる．

```
val gemm : ... -> ?c:('m,'k) mat -> ... ->
  transa:(('w,'x) mat -> ('m,'n) mat) trans -> ('w,'x) mat -> ... ->
  transb:(('y,'z) mat -> ('n,'k) mat) trans -> ('y,'z) mat -> ('m,'k) mat
```

実際には 4 つの型パラメータを用いて行列の変換前のサイズと変換後のサイズを表現しているので，('m,'n,'k,'l) trans のように 4 つの型パラメータを用いても同じ事ができる．しかし，関数型を用いた方が転置という変換を表している意図がプログラマに伝わりやすいと思われる．

[*2] ?x:t は OCaml のオプション引数（省略可能な引数）である．

3.2.2 行列の乗算方向フラグ

BLAS，LAPACK には行列の乗算方向をフラグとして受け取る関数もある．例えば，次の対称行列 $\boldsymbol{A}$ と一般行列 $\boldsymbol{B}$ の積を計算する symm 関数などである．

```
val symm : ?m:int -> ?n:int -> ?side:[‘L|‘R] -> ?up:bool ->
  ?beta:num_type -> ?cr:int -> ?cc:int -> ?c:mat (* C *) ->
  ?alpha:num_type -> ?ar:int -> ?ac:int -> mat (* A *) ->
  ?br:int -> ?bc:int -> mat (* B *) -> mat (* C *)
```

この関数は side=‘L のときは $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$ を計算し，side=‘R のときは $\boldsymbol{C} \leftarrow \alpha\boldsymbol{B}\boldsymbol{A} + \beta\boldsymbol{C}$ を計算する．$\boldsymbol{B}$ と $\boldsymbol{C}$ を $m \times n$ 行列とすると，前者の場合 $\boldsymbol{A}$ は $m \times m$ 行列，後者では $n \times n$ 行列の型を付ける必要がある．

乗算方向フラグによる影響を型で表現することは転置フラグと同様の方法で実現できる．BLAS，LAPACK において乗算方向フラグは，上の symm 関数のように，一般行列と正方行列の積の方向を指定する際に使用される．そこで，乗算方向フラグを次のように実装する．

```
type ('k, 'm, 'n) side = [‘L | ‘R] (* 実装はシグネチャで隠す *)
val left  : ('m, 'm, 'n) side (* = ‘L *)
val right : ('n, 'm, 'n) side (* = ‘R *)
```

('k, 'm, 'n) side の型パラメータは 'k × 'k 行列 $\boldsymbol{A}$ と 'm × 'n 行列 $\boldsymbol{B}$ の積の計算における，行列のサイズに対応している．$\boldsymbol{A}$ を左から掛ける場合，すなわち $\boldsymbol{A}\boldsymbol{B}$ を計算する場合は 'k = 'm であるべきなので，left の型は ('m, 'm, 'n) side になっている．同様に $\boldsymbol{A}$ を右から掛ける場合，すなわち $\boldsymbol{B}\boldsymbol{A}$ を計算する場合は 'k = 'n であるべきなので，right の型は ('n, 'm, 'n) side になっている．これを用いて symm は次のように書ける．

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

3.3 未解決の問題

BLAS と LAPACK が提供している多くの機能については，本研究の方法で静的サイズ検査を導入することができたが，サイズの等しさだけでは表現できない仕様がいくつか存在した．本節ではサイズの等しさだけでなく，自然数上の大小比較や算術演算などを用いなければ表現できないような BLAS，LAPACK の仕様について述べる．

3.3.1 オフセットとインクリメント幅

次の関数は BLAS で定義されているベクトルの加算とスカラー倍を計算するルーチンである（型は Fortran 形式ではなく Lacaml のものである[*3]）．

```
val axpy : ?n:int -> ?alpha:num_type -> ?ofsx:int -> ?incx:int -> x:vec ->
           ?ofsy:int -> ?incy:int -> vec (* y *) -> unit
```

m 次元ベクトル x の i ($i \in [1, m]$) 番目の成分を $\mathsf{x}(i)$ と表記し，破壊的代入を行う演算子を $\leftarrow$ とすると，axpy は $i = 1, \ldots,$ n について次のような計算を行う．

$$\mathsf{y}(\mathtt{ofsy} + (i-1)\mathtt{incy}) \quad \leftarrow \quad \mathtt{alpha} \cdot \mathsf{x}(\mathtt{ofsx} + (i-1)\mathtt{incx}) + \mathsf{y}(\mathtt{ofsy} + (i-1)\mathtt{incy})$$

このように BLAS，LAPACK では計算に使用する成分のオフセット (ofsx, ofsy) とインクリメント幅 (incx, incy) を指定することで，ベクトルのデータを部分的に利用した計算を行うことができる．

[*3] num_type = float の関数と num_type = Complex.t の関数が両方用意されている．

この機能は行列の行や列をコピーすることなく計算に利用する場合に有用である．BLAS，LAPACK では行列を column-major 配置の一次元配列として表現している．例えば，

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

という行列はアドレスの小さい方から $a_{11},\quad a_{21},\quad a_{12},\quad a_{22},\quad a_{13},\quad a_{23}$ という順序で一次元配列に格納される．したがって，$m \times n$ 行列の第 i $(i \geq 1)$ 列を 1 つのベクトル x として扱いたい場合は $\mathtt{ofsx} = mi$, $\mathtt{incx} = 1$, $\mathtt{n} = m$ として，x に column-major 配置の一次元配列を指定すれば良い．また，同様に第 i $(i \geq 1)$ 行は $\mathtt{ofsx} = i$, $\mathtt{incx} = m$, $\mathtt{n} = n$ と表せる．もう少しトリッキーな例では $\mathtt{incx} = 0$ として全ての成分が同じ値のベクトルを作ったり，$\mathtt{incx} = -1$ としてベクトルの成分に逆順にアクセスしたり，$\mathtt{ofsx} = 1$, $\mathtt{incx} = m + 1$, $\mathtt{n} = \min(m, n)$ として対角成分だけを取り出すといったことができる．このように，行列の一部をコピーせずにベクトルとして扱うことで，メモリ確保やコピーのコストを減らすことができる．

オフセットとインクリメント幅が指定された時に，計算ルーチンの処理中にオーバーランが起きないことを検査するためには $1 \leq \mathtt{ofsx} \leq \mathtt{dim(x)} \wedge 1 \leq \mathtt{ofsx} + (\mathtt{n} - 1)\mathtt{incx} \leq \mathtt{dim(x)}$ という不等式が成立することを示す必要がある．これは明らかにサイズの等しさだけでは表せない．

現在の試作ライブラリでは，次のようにオフセットとインクリメント幅を使わない形で実装している．

```
val axpy : alpha:num_type -> x:'n vec -> 'n vec (* y *) -> unit
```

しかし，行列の行や列の取得などは有用なので，ベクトルの内部実装にオフセットやインクリメント幅の情報を埋め込むことで，安全な一部の組み合わせについて，対応した．

3.3.2 ユーザ定義の作業領域

LAPACK の計算ルーチンは作業領域を必要とする．Lacaml では作業領域サイズの最小値と最適値を求める関数を用意しており，例えば orgqr という QR 分解に用いる計算ルーチンの作業領域サイズについては

```
val orgqr_min_lwork : n:int-> int
val orgqr_opt_lwork : ?m:int-> ?n:int-> ?k:int-> tau:vec-> ?ar:int-> ?ac:int-> mat-> int
```

という関数でそれぞれ最小値と最適値が得られる．

orgqr に限らず，作業領域サイズの最小値は引数となる行列やベクトルのサイズ（およびフラグ）のみに依存して決定される．したがって，次のように orgqr の作業領域サイズを表すサイズを表す型'n orgqr_min_lwork をライブラリの中に新しく定義して，

```
type 'n orgqr_min_lwork (* 隠蔽された幽霊型 *)
val orgqr_min_lwork : n:'n size -> 'n orgqr_min_lwork size
```

とすれば実現することができる．

しかし，この方法では，ユーザが自分で計算ルーチンを定義し，その作業領域サイズを求める関数を作るたびに，ライブラリの内部に型と関数を追加する必要がある．そこで我々は，サイズに関する基本的な演算と，その演算に対応する幽霊型をライブラリ側から提供すれば，その組み合わせでユーザ定義関数の作業領域サイズを表現可能だと予想している．例えば，$m \times n$ 行列を処理するユーザ定義ルーチンにおいて，作業領域サイズの最小値が $\max(m, n)$ で与えられるなら，

```
type ('m, 'n) max (* 隠蔽された幽霊型 *)
val max : 'm size -> 'n size -> ('m,'n) max size
```

をライブラリ側に実装し，ユーザ側では

```
type ('m,'n) my_min_lwork = ('m,'n) max
val my_min_lwork : 'm size -> 'n size -> ('m,'n) my_min_lwork size
```

などと記述すれば良い．しかし，この方法でユーザ定義の作業領域のサイズを十分に表現できるか，という点については検証が必要である．

3.3.3 最適な大きさの作業領域

作業領域サイズの最適値は 'n size 型では表現できない．なぜならば，最適な作業領域サイズは引数のベクトルや行列のサイズだけでなく成分値にも依存するため，引数の型が全く同じでも戻り値が等しくないことがあるためである．しかし，LAPACK の計算ルーチンは作業領域サイズが最小値よりも大きければ，与えられたメモリを作業領域として使用することができる．作業領域については「サイズが丁度 n」であることよりも「サイズが n 以上」であることのほうが重要である．これはある種の大小比較により実現可能ではないかと予想している．

3.3.4 大小関係に基づく仕様を含む関数

orgqr という QR 分解に使う計算ルーチンの仕様の検査はサイズの等しさだけでは実現できなかった．QR 分解とは直交行列 $\boldsymbol{Q}$ と上三角行列 $\boldsymbol{R}$ を用いて，一般行列 $\boldsymbol{A}$ を $\boldsymbol{A} = \boldsymbol{QR}$ の形に分解する操作である．LAPACK では geqrf と orgqr という 2 つの関数を用いて QR 分解を行う．orgqr は geqrf の計算結果の行列（$m \times n$ 行列）と k を引数として受け取り，求めた $\boldsymbol{Q}$ に含まれる k 個の基底ベクトルを計算する．このとき，$m \geq n \geq k$ を満たす必要があり，この仕様の検査には大小比較が必要である．

同様に，固有値分解に用いる計算ルーチン syevr と QR 分解に関連した処理を行う計算ルーチン ormqr についても仕様の検査には大小比較が必要である．

4 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検証

4.1 検証の目的

本研究におけるベクトルや行列は型の上でサイズを表現しているため，型にサイズ情報を含まない一般的なリストや配列と比較すると型に厳しく，プログラムの記述における柔軟性が低下する可能性がある．例えば，次の関数 fold_left は実行時にサイズに関するエラーを起こさないが，型エラーとなる．

```
let rec 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)
```

型エラーの理由はおおよそ以下の通りである．fold_left に型が付いたと仮定して，その第一引数 v の型を，ある型 t について t vec とする．プログラム末尾の tl_dyn の戻り値の型は，フレッシュな型 u について u vec であるから，t = u である．しかし，フレッシュの定義より t ≠ u であるため，上の fold_left は型付けできない．

ベクトルの要素に添字でアクセスする関数 get_dyn : 'n vec -> int -> float をライブラリ（Vec モジュール）に実装しておけば，fold_left を次のように実装することができる．

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

この場合，tl_dyn のようなサイズを変更する処理を行っていないため，型エラーにはならない．同様に map などの関数についても，再帰の中で tl_dyn などを用いると型が付かず，代わりに get_dyn などを用いると型が付く．ライブラリのユーザがこれらの関数を独自に定義する場合は，hd_dyn や tl_dyn のかわりに，get_dyn や set_dyn を用いる必要がある．

一般に，hd_dyn や tl_dyn のように，引数と戻り値のベクトル（もしくは行列）のサイズが異なるような関数は，たとえプログラムが実行時にサイズに関する不整合を起こさなくても，型エラーになりやすい．しかし，ライブラリ側で map や fold_left などの一通りの汎用的な関数を提供しておけば，ユーザがそれらを独自に定義する必要はなくなるため，tl_dyn などを用いる必要は少なくなると思われる．加えて，一般にベクトルや行列の要素へのアクセスは添字を用いることが多く，hd_dyn や tl_dyn により構造を分解するような処理を行う頻度は低いと予想される．以上のような予想が正しいか，検証する必要がある．

4.2 検証結果

検証のために，我々はこれまでに述べた方法で静的サイズ検査機能を実装した線形代数演算ライブラリを OCaml で試作した．本節では線形代数および機械学習の代表的アルゴリズムを Lacaml と試作ライブラリで実装してソースコードを比較する．比較のために，試作ライブラリは Lacaml と互換性のあるインタフェースになっている．実装したアルゴリズムは LU 分解，QR 分解，単純パーセプトロン，K-means クラスタリング，主成分分析の 5 種類である．今回検証に用いた試作ライブラリと，実装したアルゴリズムのサンプルコードは https://github.com/akabe/ からダウンロードできる．

検証の結果，いずれアルゴリズムについても，アプリケーションプログラムへの変更はほぼ不要であることが確認できた．今回実装したアルゴリズムにおいて，書き換えが必要な箇所は以下の通りであった．

- Lacaml のベクトルと行列は Bigarray で実装されているので，x.{i,j} という記法で添字アクセスができる．しかし，本研究における静的サイズ検査の実装では，行列などデータ構造がシグネチャで隠蔽されるので，ライブラリに実装した get_dyn，set_dyn 関数を使って要素にアクセスする必要がある．
- of_list_dyn などの関数に 'n size 型の値を渡す必要がある．
- 必要に応じてサイズ情報を int に変換する必要がある．例えば，**for** ループの回数にサイズ情報を指定する場合などに，この作業が必要になる．
- 転置フラグなどの書き換えが必要である．

今回検証に用いたアルゴリズムでは，上述の比較的小さい書き換えで済んだ．

このように，今回実装したアルゴリズムについては，一つ一つの行列演算のサイズの整合性を等しさのみの判定で検査する方法で，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．また，型に厳しく，プログラムの記述の柔軟性が低下する点については，全く問題にはならなかった．加えて，生成的な幽霊型がエスケープするケースはなく，OCaml の型推論の恩恵を受けることができ，ライブラリのユーザが型注釈を書く必要はなかった．

5 関連研究

配列等のサイズの整合性を静的に保証する既存研究としては，自然数上の依存型を用いる Dependent ML [1] や sized type [2] 等が存在する．特に，Dependent ML の後継となるプログラミング言語 ATS [5] には BLAS や LAPACK のバインディングが存在する．本研究で提案する静的サイズ検査付き線形代数演算ライブラリでは，サイズの等しさしか判定することができないが，これらの

既存手法では大小比較などを含むより複雑な仕様を表現することができ，添字アクセスの境界検査なども行うことができる．また，本研究の方法ではライブラリ（モジュール）の内部実装はサイズ検査されず，ライブラリのユーザの間違いしか防ぐことができないが，これらの手法では内部実装まで含めてサイズの整合性を保証することができる（ただし，今現在，ATS の BLAS は C 言語のライブラリの単なるラッパーなので，内部実装は検査されていない）．しかし，これらの手法は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を必要とする．それに対して，我々の手法は単純かつ実用的であり，ほぼ標準的な ML 型とモジュールシステムで実現することができた．

文献 [12] では Coq 上で幽霊型を用いて行列演算のサイズ検査を実装している．Coq に基づいているため，本研究よりも多くの型注釈を書く必要がある．

Haskell などではサイズを型で表現するアプローチを実装している例もある．例えば，[13] では GADT を使ってベクトルや行列のサイズを表現している．この方法はライブラリの内部実装のサイズ整合性を保証することができる．一方．本研究はライブラリの外部インタフェースを対象としており，両者を組み合わせることも可能と思われる．

静的なサイズ検査を実装している実用的な線形代数演算ライブラリには，C++ で実装された Eigen [14] や uBlas [15] が存在する．しかし，C++ の型システムは健全ではないため，根本的にはサイズの整合性を保証することができない．

6 結論

我々は生成的な幽霊型をパラメータとする多相型を用いることにより，ほぼ標準的な ML 型とモジュールシステムのみで，サイズの整合性を静的に保証する線形代数演算ライブラリを提案した．我々のライブラリは，一つ一つの行列演算のサイズの整合性を，等しさのみの検査によって実現する．この方法により，既存のアプリケーションプログラムのサイズ検査を行うことができ，プログラム記述の柔軟性の低下は実用上問題にならないことがわかった．また，既存手法 [1, 2, 5] のように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を加える必要がないこともわかった．本研究のアイディアは単純かつ実用的であり，BLAS，LAPACK という主要な線形代数演算ライブラリのインターフェースに対し，ほとんどの高水準な行列演算におけるサイズの整合性を静的に保証することができた．

本研究を進める中で，次のような課題が存在することもわかった．

- 本研究の方法では，ライブラリの内部実装に関してサイズ検査が行われない．そのため，内部実装がサイズ情報を含む型注釈と矛盾している場合は，サイズの整合性を保証することはできない．ライブラリ自体の信頼性を保証するためには，内部実装まで検査されるべきである．
- 2 節では，OCaml に生成的な幽霊型を戻り値の型に含むような関数を記述する機能を加えた，仮想的なプログラミング言語を用いて，本研究のアイディアを説明した．そのような言語が存在すれば，本研究のアイディアをより直接的に実装できるが，そのためには生成型を含む型システムの定式化が望まれる．
- QR 分解など一部の関数の仕様（3.3.4 節）や作業領域のサイズの表現（3.3.3 節）などの実現には大小比較が必要である．また，オフセットとインクリメント幅の指定（3.3.1 節）や `tl_dyn` あるいは `get_dyn` のような一部の低水準な関数の仕様の検査も，等しさの比較だけでは不十分であり，自然数上の大小比較や算術演算が必要である．
- 3.3.2 節では，基本的な演算をライブラリ側で提供すれば，ユーザ定義関数の作業領域サイズを表現できるという予想を立てたが，それで十分かどうか調査する必要がある．

参考文献

[1] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 214–227, San Antonio, January 1999.

[2] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computing (HOSC)*, Vol. 14, No. 2–3, pp. 261–300, September 2001.

[3] Stanford University's Pervasive Parallelism Laboratory (PPL). OptiML. `http://stanford-ppl.github.io/Delite/optiml/`.

[4] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *ICML '11: Proceedings of the 28th Intl. Conference on Machine Learning*, June 2011.

[5] Hongwei Xi. The ATS programming language. `http://www.ats-lang.org/`.

[6] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electr. Notes Theor. Comput. Sci.*, Vol. 59, No. 1, pp. 36–52, 2001.

[7] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *IN PROCEEDINGS OF THE 2ND CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES*, pp. 109–122. ACM Press, 1999.

[8] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pp. 245–262. Palgrave Macmillan, 2003.

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

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

[11] Markus Mottl and Christophe Troestler. Lacaml – linear algebra for ocaml. `https://bitbucket.org/mmottl/lacaml`.

[12] Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding kleene algebras. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, Vol. 6172 of *Lecture Notes in Computer Science*, pp. 163–178. Springer, 2010.

[13] hyone. Length indexed matrix and indexed functor. `https://gist.github.com/hyone/3990929`.

[14] Eigen. `http://eigen.tuxfamily.org/`.

[15] uBlas. `http://www.boost.org/doc/libs/1_55_0/libs/numeric/ublas/doc/`.