# Expressive Linear Algebra in Haskell

Henning Thielemann

2019-08-21

1 Motivation

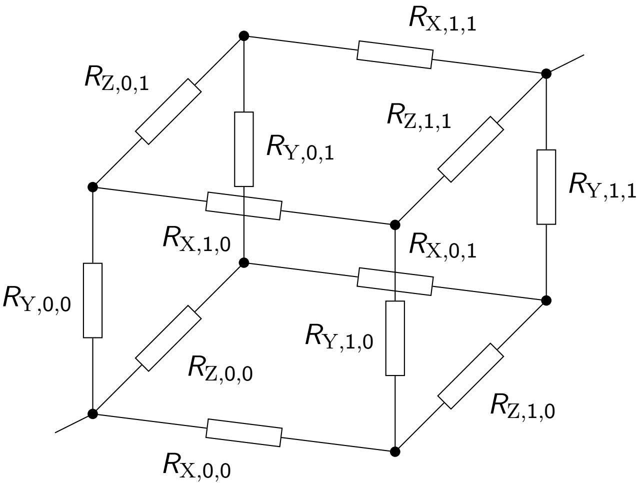
2 Solution

3 More realistic problem

4 More features

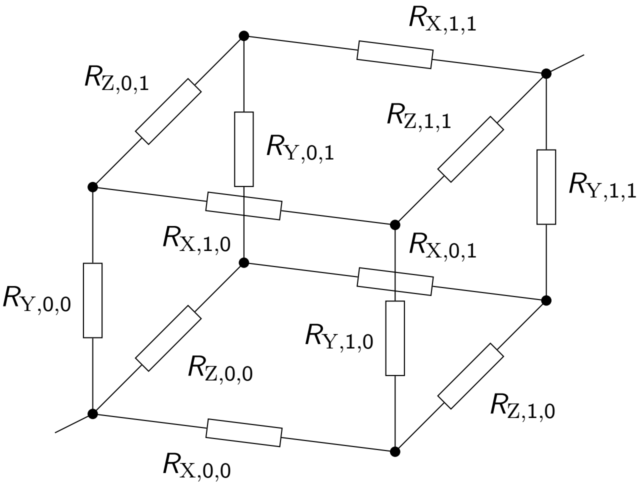
5 Closing

# Resistor cube



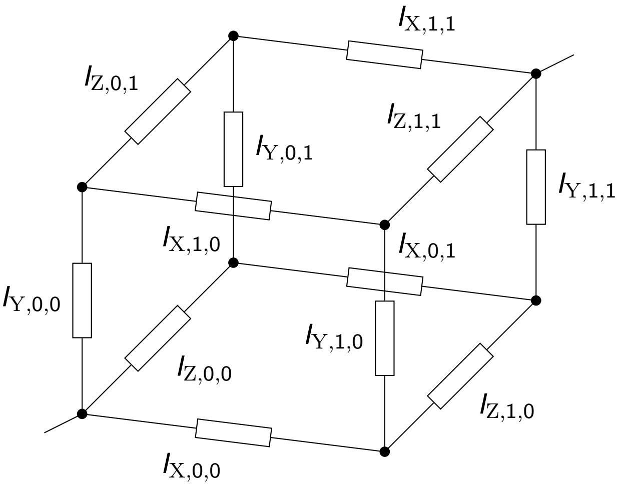

Wanted:
Total resistance

# Ohm's law



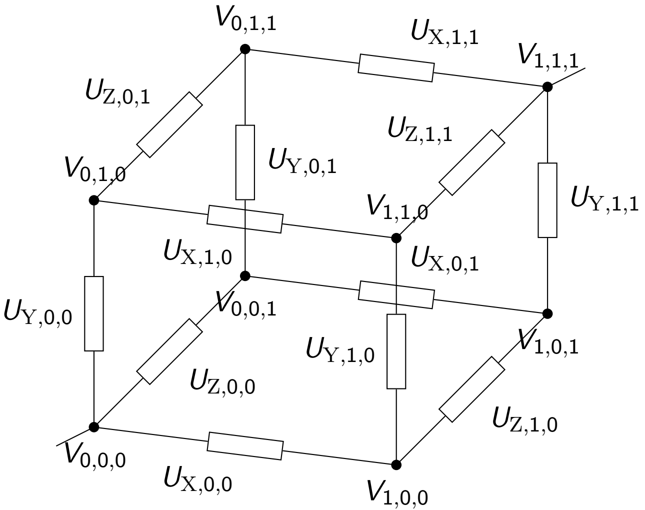

$$R_{\mathrm{X},0,0} = \frac{U_{\mathrm{X},0,0}}{I_{\mathrm{X},0,0}},$$
$$R_{\mathrm{Y},1,0} = \frac{U_{\mathrm{Y},1,0}}{I_{\mathrm{Y},1,0}},$$
$$\ldots$$

# Kirchhoff's current law




$$0 = -I_{X,0,0} + I_{Y,1,0} + I_{Z,1,0},$$
$$0 = -I_{X,0,1} + I_{Y,1,1} - I_{Z,1,0},$$
$$\ldots$$

# Kirchhoff's voltage law




$$U_{X,0,0} = -V_{0,0,0} + V_{1,0,0},$$
$$U_{Y,1,0} = -V_{1,0,0} + V_{1,1,0},$$

$$
\left(\begin{array}{c|cccccccccccc|cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
\hline
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)
\cdot
\left(\begin{array}{c}
I \\ \hline I \\ I \\ I \\ I \\ I \\ I \\ I \\ I \\ I \\ I \\ I \\ I \\ \hline V \\ V \\ V \\ V \\ V \\ V \\ V \\ V
\end{array}\right)
=
\left(\begin{array}{c}
0 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1\mathrm{A}
\end{array}\right)
$$

$$V_{0,0,0} = 0 \qquad I_{\mathrm{total}} + I_{\mathrm{X},0,0} + I_{\mathrm{Y},0,0} + I_{\mathrm{Z},0,0} = 0$$

$$R_{\mathrm{X},0,0} \cdot I_{\mathrm{X},0,0} + V_{0,0,0} - V_{1,0,0} = 0 \qquad -I_{\mathrm{X},1,1} - I_{\mathrm{Y},1,1} - I_{\mathrm{Z},1,1} = 1\mathrm{A}$$

# Solution

$$R_{\text{total}} = \frac{U_{\text{total}}}{I_{\text{total}}} \quad (1)$$

$$= \frac{V_{1,1,1} - V_{0,0,0}}{I_{\text{total}}} \quad (2)$$

# Matrix blocks

$$A = \begin{pmatrix} 0 & 0 & u_{0,0,0}^T \\ 0 & \mathbf{R} & \mathbf{V} \\ u_{0,0,0} & \mathbf{I} & 0 \end{pmatrix}$$

- **R**: encodes Ohm's law
- **V**: encodes Kirchhoff's voltage law
- **I**: encodes Kirchhoff's current law

# Matrix symmetry

$$A = \begin{pmatrix} 0 & 0 & u_{0,0,0}^T \\ 0 & \mathbf{R} & \mathbf{V} \\ u_{0,0,0} & \mathbf{I} & 0 \end{pmatrix}$$

$$\mathbf{R} = \begin{pmatrix} R_{\mathrm{X},0,0} & & & \\ & R_{\mathrm{X},0,1} & & \\ & & \ddots & \\ & & & R_{\mathrm{Z},1,1} \end{pmatrix}$$

$$\mathbf{I} = \mathbf{V}^T$$

# Matrix features

Matrix $A$ is

- symmetric,
- composed from blocks,
- and every block has a special sub-structure.

Can we represent this with Haskell's type system?

1 Motivation

2 Solution

3 More realistic problem

4 More features

5 Closing

# Solution of simultaneous linear equations

Specialised:

```
(#\|) ::
   (Shape height, Eq height, Floating a) =>
   Matrix.Symmetric height a ->
   Vector height a -> Vector height a
```

Generic:

```
(#\|) ::
   (Solve typ, HeightOf typ ~ height,
    Eq height, Floating a) =>
   Matrix typ a -> Vector height a -> Vector height a
```

- Infix operator reads as: "Matrix divides column vector".
- Implemented by LAPACK's SPTRS

# Array shapes and indices

`comfort-array`

- Index type is a type function of the array shape.

```
class Shape.C shape where
  size :: shape -> Int

class Shape.C shape => Shape.Indexed shape where
  type Index shape
```

Read a single element:

```
(!) :: Array sh a -> Index sh -> a
```

# Zero-based indexing shape

```
newtype Shape.ZeroBased n = ZeroBased n

instance (Integral n) => Shape.Indexed (ZeroBased n) where
  type Index (ZeroBased n) = n

(!) :: Array (ZeroBased Int) a -> Int -> a

array ! 0
```

- Classical zero-based indexing scheme as in `hmatrix`
- In contrast to `array` lower bound is statically fixed to zero

# Enumeration shape

```
data Shape.Enumeration enum = Enumeration

instance (Enum enum, Bounded enum) =>
    Shape.Indexed (Enumeration enum) where
  type Index (Enumeration enum) = enum

(!) :: Array (Enumeration Ordering) a -> Ordering -> a

array ! compare x y
```

- Shape statically determined by an **Enum** type

# Cartesian product shape

```
instance
  (Shape.Indexed sha, Shape.Indexed shb) =>
    Shape.Indexed (sha, shb) where
  type Index (sha, shb) = (Index sha, Index shb)

type E = Enumeration
(!) ::
  Array (E Ordering, E Bool) a -> (Ordering, Bool) -> a

array ! (compare x y, odd x)
```

- Represents a two-dimensional array of rectangular shape

# Sum shape

```
instance
  (Shape.Indexed sha, Shape.Indexed shb) =>
    Shape.Indexed (sha:+:shb) where
  type Index (sha:+:shb) = Either (Index sha) (Index shb)

type E = Enumeration
(!) ::
  Array (E Ordering :+: E Bool) a ->
  Either Ordering Bool -> a

array ! Right False
```

- Useful for block matrices

# Array shapes for corners and edges

```
data Coord = C0 | C1
  deriving (Eq, Ord, Show, Enum, Bounded)
data Dim = DX | DY | DZ
  deriving (Eq, Ord, Show, Enum, Bounded)
type Corner = (Coord,Coord,Coord)
type Edge = (Dim,Coord,Coord)

type CoordSh = Shape.Enumeration Coord
type DimSh = Shape.Enumeration Dim
type CornerShape = (CoordSh, CoordSh, CoordSh)
type EdgeShape   = (DimSh, CoordSh, CoordSh)
type BlockShape  = ():+:EdgeShape:+:CornerShape
```

# Achievement

- no need to write index flattening function yourself
- consistent structure of matrix and vectors
- no fight over zero- vs. one-based index counting
- no off-by-one errors

# Voltage matrix

```
voltageMatrix :: Matrix EdgeShape CornerShape a
voltageMatrix =
   Matrix.fromRowArray cornerShape $
   fmap
      (\e ->
         Array.fromAssociations cornerShape 0
            [(edgeCorner e C0,  1),
             (edgeCorner e C1, -1)]) $
   BoxedArray.indices edgeShape

edgeCorner :: Edge -> Coord -> Corner
edgeCorner (ed,e0,e1) coord =
   case ed of
      DX -> (coord,e0,e1)
      DY -> (e0,coord,e1)
      DZ -> (e0,e1,coord)
```

# Stacking symmetric matrices

```
#%%%# ::
  (Matrix.Symmetric sha a, Matrix.General sha shb a) ->
                              Matrix.Symmetric shb a ->
  Matrix.Symmetric (sha:+:shb) a
```

$$\texttt{(a,b) \#\%\%\%\# c} \quad \sim \quad \begin{pmatrix} A & B \\ B^T & C \end{pmatrix}$$

# Complete symmetric matrix

```
fullMatrix ::
  Vector EdgeShape a ->
  Matrix.Symmetric (():+:EdgeShape:+:CornerShape) a
fullMatrix resistances =
  let symmetricZero = Matrix.zero . MatrixShape.symmetric
  in (symmetricZero (),
      Matrix.singleRow $
       Vector.unit (edgeShape:+:cornerShape)
                        (Right sourceCorner))
     #%%#
     (diagonal resistances, voltageMatrix)
     #%%#
     symmetricZero cornerShape
```

# Result

```
sourceCorner, destCorner :: Corner
sourceCorner = (C0,C0,C0)
destCorner   = (C1,C1,C1)

totalResistance :: Double
totalResistance =
  let matrix = fullMatrix resistances
      ix = Right (Right destCorner)
      solutionVector =
        matrix #\| Vector.unit (Symmetric.size matrix) ix
  in - solutionVector ! ix
```

# Answer

For $R_{\mathrm{X},0,0} = R_{\mathrm{X},0,1} = \cdots = R_{\mathrm{Z},1,1}$:

$$R_{\mathrm{total}} = \frac{5}{6} \cdot R_{\mathrm{X},0,0}$$

# Advantage

Matrix symmetry

- Specialized solvers
- Halved space usage
- Algebraic properties encoded in types, e.g.
  `eigenvalues :: Hermitian sh a -> Vector sh (RealOf a)`

Complete example: `resistor-cube`

# However

- Pretty artificial exercise
- The `lapack` bindings still have something to offer for the general problem.

1 Motivation

2 Solution

3 More realistic problem

4 More features

5 Closing

# Set shape

```
import Data.Set (Set)

instance Ord ix => Shape.Indexed (Set ix) where
  type Index (Set ix) = ix

(!) :: Array (Set ix) a -> ix -> a
```

- **Array** (Set ix) a is isomorphic to Map ix a
- logarithmic **lookup**
- no **insert** or **delete**
- instead fast Vector.add etc.
- caveat: size compatibility check means comparision of sets
- use as Matrix dimension

# Symmetric matrix for general problem

```
fullMatrix ::
   (Graph.Edge edge, Ord node) =>
   Graph edge node a () -> node ->
   Matrix.Symmetric (() :+: Set (edge node) :+: Set node) a
fullMatrix graph source = ...
```

- Graph structure: `comfort-graph`
- Complete implementation: `linear-circuit`

# Sparsity

- Matrix $A$ is sparse
- Most big matrix problems are sparse
- LAPACK has no specialised algorithms for this case
- so we don't currently provide ones, as well

# Block structure

- LAPACK ignores the block structure
- We could optimize using blockwise inversion

1 Motivation

2 Solution

3 More realistic problem

4 More features

5 Closing

# More features

- conversion between `Vector` and `Matrix` preserves shape structure
  i.e. matrices can be vectors in an equation system
- `lapack-ffi`: auto-generated via `lapack-ffi-tools`
- Test framework for generating consistent and inconsistent matrix dimensions using `unique-logic-tf`
- faster `comfort-array` processing via LLVM and `knead`
- support for `hyper` (strongly hyped interactive notebook editor)

# Closed-world classes

- `lapack`:
  all functions support all element types that LAPACK supports,
  i.e. **`Float`**, **`Double`**, **`Complex Float`**, **`Complex Double`**
- closed-world classes:
  - add methods without extending class
  - you cannot add more types
- plain Haskell 98

# Closed-world class for **Float** and **Double**

```
class (Floating a, Prelude.RealFloat a) => Real a where
  switchReal :: f Float -> f Double -> f a

instance Real Float  where switchReal f _ = f
instance Real Double where switchReal _ f = f
```

```
type ASUM_ a = Ptr CInt -> Ptr a -> Ptr CInt -> IO a
newtype ASUM a = ASUM {getASUM :: ASUM_ a}

asum :: Real a => ASUM_ a
asum = getASUM $ switchReal (ASUM S.asum) (ASUM D.asum)
```

# Closed-world class for all LAPACK number types

```
class (Prelude.Fractional a) => Floating a where
  switchFloating ::
    f Float -> f Double ->
    f (Complex Float) -> f (Complex Double) ->
    f a

instance Floating Float where
  switchFloating f _ _ _ = f
instance Floating Double where
  switchFloating _ f _ _ = f
instance (Real a) => Floating (Complex a) where
  switchFloating _ _ fz fc =
    getCompose $ switchReal (Compose fz) (Compose fc)
```

# Matrix type tags for triangle-based matrices

```
type Diagonal  sh a = Triangular Empty  Empty  sh a
type Lower     sh a = Triangular Filled Empty  sh a
type Upper     sh a = Triangular Empty  Filled sh a
type Symmetric sh a = Triangular Filled Filled sh a

diagonal :: Vector sh a -> Triangular lo up sh a
transpose ::
  Triangular lo up sh a -> Triangular up lo sh a
```

GHC can infer:

- diagonal matrix is also lower triangular and symmetric
- transposition of transposition maintains original shape
- transposition of lower triangular matrix is upper triangular
- transposition of diagonal or symmetric matrix maintains shape

# Matrix type tags for triangle-based matrices

```
class Content c where
instance Content Empty where
instance Content Filled where

class (Content lo, Content up) => DiagUpLo lo up where
instance DiagUpLo Empty  Empty  where
instance DiagUpLo Empty  Filled where
instance DiagUpLo Filled Empty  where

square :: (Content lo, Content up) =>
  Triangular lo up sh a -> Triangular lo up sh a

multiply :: (DiagUpLo lo up, DiagUpLo up lo) =>
  Triangular lo up sh a ->
  Triangular lo up sh a -> Triangular lo up sh a
```

# Matrix type tags for triangle-based matrices

```
square ::
  (Content lo, Content up) =>
  Triangular lo up sh a -> Triangular lo up sh a

multiply ::
  (DiagUpLo lo up, DiagUpLo up lo) =>
  Triangular lo up sh a ->
  Triangular lo up sh a -> Triangular lo up sh a
```

GHC can infer:

- if multiplication preserves triangular shape,
  then multiplication of transposed matrices does that, too
- e.g. `multiply (transpose a) (transpose b)`
  is accepted without intervention
- `square` preserves shape wherever `multiply` does (superclass relation)

# Matrix size relations

```
type Square  height width a = Matrix.Full Small Small height width a
type Tall    height width a = Matrix.Full Big   Small height width a
type Wide    height width a = Matrix.Full Small Big   height width a
type General height width a = Matrix.Full Big   Big   height width a
```

Relations:

- Square: `height == width`
- Tall: `size height >= size width`
- Wide: `size height <= size width`
- General: arbitrary `height`, `width`

all relations are transitive

# Matrix size relations

```
transpose ::
  Matrix.Full vert horiz height width a ->
  Matrix.Full horiz vert width height a

multiply ::
  Matrix.Full vert horiz height width a ->
  Matrix.Full vert horiz height width a ->
  Matrix.Full vert horiz height width a
```

GHC can infer:

- transposition of transposition maintains original shape
- transposition of square matrix is square
- transposition of tall matrix is wide
- square times square is square
- tall times tall is tall

# Matrix size relations

```
transpose ::
  Matrix.Full vert horiz height width a ->
  Matrix.Full horiz vert width height a

multiply ::
  Matrix.Full vert horiz height width a ->
  Matrix.Full vert horiz height width a ->
  Matrix.Full vert horiz height width a
```

ugly:

- you must explicitly relax size relations

1 Motivation

2 Solution

3 More realistic problem

4 More features

5 Closing

# LAPACK+BLAS

The `lapack` binding is based on LAPACK+BLAS library.
Advantages:

- standard solution, ubiquitous availability
- many implementations optimized for usage of caches and vector units (OpenBLAS, ATLAS, MKL)
- many advanced functions:
  - linear solvers,
  - least squares and minimum norm solvers,
  - eigenvalues and
  - singular value decompositions

  of many common matrix types

# LAPACK+BLAS

Disadvantages:

- restricted to **`Float`**, **`Double`**, **`Complex Float`**, **`Complex Double`**
- i.e. no QuadDouble, no finite fields, no interval arithmetics
- no sparse matrices
- no batch processing,
  i.e. optimized simultaneous solution of many equally sized problems
- missing basic functions:
  matrix transpose, minimum and maximum, product

# Existing Alternatives

Alternatives in Haskell:

- `hmatrix`:
  matrices with dynamic and static sizes of natural numbers
- `repa`, `accelerate`:
  for efficiency reasons restricted to cubic shapes
- `array`:
  flexible indexing schemes but shape and index types coincide

# Conclusion

- There is more to static checks on matrix computations than type-level natural numbers.
- Special matrix types:
  better documentation + optimized algorithms
- Transpositions treatable with simple type tricks –
  Can be generalized to operations on (small) permutations.