2 Idris

"I think you should be more explicit here in step two."

Joe Programmer

The supervisor Mr. Idris
Paths to Understanding Idris

- Haskell
- Idris
- Constructive Mathematics

- Dependent Types, Termination
- Curry-Howard Isomorphism
Functional Programming in Idris
Idris vs. Haskell: Data Types

**Idris**

```idris
data Bool : Type where
  True : Bool
  False : Bool

data Nat : Type where
  Z : Nat
  S : Nat → Nat

data List : Type → Type where
  [] : List a
  (::) : a → List a → List a
```

**Haskell**

```haskell
data Bool = True | False

data Nat = Z | S Nat

data [a] = [] | (:) a [a]
```

- **GADT notation is more general**
- **Type constructor ≈ function on types**
- **Common result type for all constructors**
- **(Complete) type of constructor**
- **Only argument type of constructor**
- **Type Parameter**
- **Same in Idris**
Idris vs. Haskell: Functions

\[
\forall \ a
\]

\(\text{infixr 7 ++}\)

\((++): \{a: \text{Type}\} \rightarrow \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a\)

\([] \quad ++ \text{ ys } = \text{ ys}\)

\((x :: \text{ xs}) \quad ++ \text{ ys } = x :: (\text{ xs } ++ \text{ ys})\)

\(\text{id}: \{\text{a:Type}\} \rightarrow \text{ a } \rightarrow \text{ a}\)

\(\text{id } x = x\)

\(\text{true1 } = \text{id True}\)

\(\text{zero1 } = \text{id } 0\)

\(\text{id}: (\text{a:Type}) \rightarrow \text{ a } \rightarrow \text{ a}\)

\(\text{id } a \ x = x\)

\(\text{true2 } = \text{id Bool True}\)

\(\text{zero2 } = \text{id Nat } 0\)

\(\text{not } :: \text{ Bool } \rightarrow \text{ Bool}\)

\(\text{not True } = \text{ False}\)

\(\text{not False } = \text{ True}\)

Implicit Argument (is optional; will be automatically inferred by type checker)

Explicit Type Argument
1. Define the Idris function `(!!)` for extracting the nth element from a list (use zero for first element).

```
(!!) : {a : Type} -> List a -> Nat -> Maybe a
[]      !! _     = Nothing
(x::_)  !! Z     = Just x
(_::xs) !! (S n) = xs !! n
```

```
(++) : {a : Type} -> List a -> List a -> List a
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

---

```
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```
Total Functions

tail : {a : Type} → List a → List a
tail [] = []
tail (_ :: xs) = xs

head : {a : Type} → List a → a
head [] = ???
head (x :: _) = x

%default total

In Haskell

tail :: [a] → [a]
tail (_:xs) = xs

head :: [a] → a
head (x:_:xs) = x

Note: This is actually not how it's done in Idris. We'll get back to this later …

We are not allowed to cheat in Idris!
Piazza Question

Type Constructors
What is a Type Constructor?

In Haskell

\[
[\ ] :: \ast \to \ast \\
\text{Maybe} :: \ast \to \ast \\
(\to) :: \ast \to \ast \to \ast
\]

Use "kind Maybe" in ghci

In Idris

A function that maps types and values to types

\[
\text{Vect} :: \text{Nat} \to \text{Type} \to \text{Type} \\
\text{Segment} :: \text{Nat} \to \text{Type}
\]

Use "t[ype] Maybe" in Idris

\[
\text{List} :: \text{Type} \to \text{Type} \\
\text{Maybe} :: \text{Type} \to \text{Type} \\
(\to) :: \text{Type} \to \text{Type} \to \text{Type}
\]
Dependent Types

**Type constructor**

Single : Bool → Type

Single True  = Nat
Single False = List Nat

**Value argument**

mkSingle : (b : Bool) → Nat → Single b

mkSingle True  x = x
mkSingle False x = [x]

**Result type depends on argument value**

*Idris> mkSingle True 4
4 : Nat

*Idris> mkSingle False 4

**Producing values of different types**

sum : (b : Bool) → Single b → Nat

sum True  x = x
sum False [] = 0
sum False (x::xs) = x + sum False xs

**Consuming values of different types**

*Idris> sum True 4
4 : Nat

*Idris> sum False [4]
Vectors

\[
data \text{ Vect} : \text{ Nat} \to \text{ Type} \to \text{ Type} \text{ where} \\
\text{ Nil} : \text{ Vect} \ Z \ a \\
(\cons) : a \to \text{ Vect} \ n \ a \to \text{ Vect} \ (S \ n) \ a
\]

Vectors of length \( n \)

\[
data \text{ List} : \text{ Type} \to \text{ Type} \text{ where} \\
\text{} : \text{ List} \ a \\
(\cons) : a \to \text{ List} \ a \to \text{ List} \ a
\]

\[
\text{ Vect} 3 \ \text{ Nat} \\
\text{ Vect} 4 \ \text{ Nat} \\
\text{ List} \ \text{ Nat} \\
\text{ xs} = 1 \ \cons \ 2 \ \cons \ 7 \ \cons \ [] \\
\text{ ys} = 9 \ \cons \ \text{ xs}
\]
4. Define the Idris functions `eqNat` and `eqList` for comparing two natural numbers and two lists of values.
5. Define the Idris function `eqVect` for comparing two vectors of values.

```idris
eqVect : Eq a => Vect n a -> Vect n a -> Bool
eqVect []      []      = True
eqVect (x::xs) (y::ys) = x==y && eqVect xs ys
```

```idris
eqList : Eq a => List a -> List a -> Bool
eqList []      []      = True
eqList (x::xs) (y::ys) = x==y && eqList xs ys
eqList _        _      = False
```
Piazza Question

Vector Comparison
Vector Size

size : List a → Nat
size [] = Z
size (_ :: xs) = S (size xs)

size : Vect n a → Nat
size {n} _ = n
Exercises

6. Define the functions **head** and **tail** for vectors.

```idris
data Vect : Nat → Type → Type where
  Nil : Vect Z a
  (::) : a → Vect n a → Vect (S n) a
```
7. Define a type for matrices with \( n \) rows and \( m \) columns using nested vectors.

8. Define the functions `firstRow` and `firstCol` for matrices.
Paths to Understanding Idris

- Dependent Types, Termination
- Curry-Howard Isomorphism
- Haskell
- Idris
- Constructive Mathematics
Proving with Idris
Reversing Vectors

\[
\text{rev} : \text{Vect } n \ a \rightarrow \text{Vect } n \ a \\
\text{rev} [] = [] \\
\text{rev} (x::xs) = \text{rev} \ xs \ +++ \ [x]
\]

Type mismatch between
\(\text{Vect } (k + 1) \ a\) (Type of \(\text{rev} \ xs\) +++ [x])
and
\(\text{Vect } (S \ k) \ a\) (Expected type)

Specifically:
Type mismatch between
plus \ k \ 1
and
\(S \ k\)

Infixr 7 +++

(+++) : Vect n a \rightarrow \text{Vect } m \ a \rightarrow \text{Vect } (n + m) \ a \\
(+++) [] \ ys = ys \\
(+++) (x::xs) \ ys = x :: (xs +++ ys)

Using ++ leads to a different type error message because of overloading

Does not typecheck
Reversing Vectors

\[
\text{rev : Vect } n \ a \to \text{ Vect } n \ a \\
\text{rev } [] = [] \\
\text{rev } (x::xs) = \text{rev } xs \ +++ [x]
\]

\[
\text{rev } xs \ +++ [x] : \text{ Vect } (n + 1) \ a
\]

\[
(\+++) : \text{ Vect } n \ a \to \text{ Vect } m \ a \to \text{ Vect } (n + m) \ a
\]

Type mismatch between \(\text{ Vect } (k + 1) \ a\) (Type of (rev xs) +++ [x]) and \(\text{ Vect } (S \ k) \ a\) (Expected type)

Specifically:
Type mismatch between 
\[
\text{ plus } k \ 1
\]
and 
\[
S \ k
\]
Convincing the Type Checker ...

\[
\text{rev} : \text{Vect } n \ a \to \text{Vect } n \ a \\
\text{rev } [] = [] \\
\text{rev } (x::xs) = \text{rev } xs +++ [x]
\]

We know (RHS):

\[
\text{rev } xs +++ [x] : \text{Vect } (n + 1) \ a
\]

We need (type declaration):

\[
\text{rev } xs +++ [x] : \text{Vect } (S \ n) \ a
\]

To do:

1. \[n + S \ Z = S \ n\]

Prove (to Idris (!)):

2. \[\text{ Vect } (n + S \ Z) \ a = \text{ Vect } (S \ n) \ a\]

3. \[\text{rev } xs +++ [x] : \text{ Vect } (n + S \ Z) \ a \Rightarrow \text{rev } xs +++ [x] : \text{ Vect } (S \ n) \ a\]
Talking to the Type Checker

What does the type checker do?
Verifying type declaration

Is this true?

How to “convince” the type checker of a fact/proposition \( P \)?
Express \( P \) as a type \( T \), and find an expression \( e \) that has type \( T \).
Talking to the Type Checker

How to “convince” the type checker of a fact/proposition $P$?

Express $P$ as a type $T$, and find an expression $e$ that has type $T$.

Witness for Proposition

expression : Type

Curry-Howard Isomorphism

The values of a type are the proofs for the proposition represented by it.
The Big Picture: Understanding Proofs in Idris

- **Dependent type**: Type that depends on values
- **Indexed type**: Type that is partitioned into non-overlapping subtypes by values (called “indexes”)
- **Singleton type**: Type that contains just one value
- **Equality type**: Type to express the equality between two values

**Curry-Howard Isomorphism**

1. **Type**: Proposition
2. **Function**: Proof of implication
3. **Function**: Proof transformer
4. **Value**: Proof

Don’t confuse this with instances of the Eq type class!
The Big Picture:
Understanding Proofs in Idris

- **Dependent type**: Type that depends on values

- **Indexed type**: Type that is partitioned into non-overlapping subtypes by values (called “indexes”)

- **Singleton type**: Type that contains just one value

- **Equality type**: Type to express the equality between two values

- **Curry-Howard Isomorphism**
  - Type: Proposition
  - Value: Proof

- **Function**: Proof of implication
- **Function**: Proof transformer

Don’t confuse this with instances of the Eq type class!

Proof by case analysis

Proof by induction

... later
**Proposition Types: Basic Propositions**

- **Proof of Proposition A**
  - `val : A`
  - **Witness for Proposition**
    - `expression : Type`

- **Witness for Proposition**
  - `true : Bool`
    - **"There are boolean values"**
  - `Z : Nat`
    - **"There are natural numbers"**
The Big Picture: Understanding Proofs in Idris

1. **Type: Proposition**
   - Value: Proof

2. **Function: Proof of implication**
3. **Function: Proof transformer**

Don’t confuse this with instances of the `Eq` type class!

- **Dependent type**: Type that depends on values
- **Indexed type**: Type that is partitioned into non-overlapping subtypes by values (called “indexes”)
- **Singleton type**: Type that contains just one value
- **Equality type**: Type to express the equality between two values

Curry-Howard Isomorphism
Proposition Types: Implication

- **Implication**
  - \( \forall x : a \rightarrow a \)
  - "Any proposition implies itself"

- **Witness for**
  - \( \forall x \rightarrow x \)

- **Nat \( \rightarrow \) Bool**
  - even : Nat \( \rightarrow \) Bool
  - "A boolean can be obtained from a number"

- **Proof of Implication A \( \Rightarrow \) B**
  - fun : A \( \rightarrow \) B
Proposition Types: Conjunction

Proof of Conjunction A \land B

(x, y) : (a, b)

(even, \ x \Rightarrow x)

Witness for

“A boolean can be obtained from a number”
and “Any proposition implies itself”

(even, \ x \Rightarrow x) : (Nat \rightarrow \text{Bool}, a \rightarrow a)
Piazza Question

Proposition types
Piazza Question

Logical connectives
What is a Proof?

Proof steps

Premise → proof step → Conclusion

Logical inference rules

\[
\begin{array}{c}
A & B \\
\hline
A \land B
\end{array}
\]

\[
\begin{array}{c}
A & A \implies B \\
\hline
B
\end{array}
\]

Proposition transformations

\[
\begin{array}{c}
 n = 3 + 1 \\
\hline
 n = 4
\end{array}
\]

\[
\begin{array}{c}
 n = m \\
\hline
 n + n = m + m
\end{array}
\]

calculation

substitution
Proofs as Transformations of Propositions

Known

To be proved

Premise

Premise

Premise

Conclusion

Premise

Premise

Conclusion'

Premise

Proof step

Proof step

Known

To be proved
The Big Picture: Understanding Proofs in Idris

1. **Type: Proposition**
   - Value: Proof

2. **Function: Proof of implication**

3. **Function: Proof transformer**

4. **Indexed type**
   - Type that is partitioned into non-overlapping subtypes by values (called “indexes”)

5. **Singleton type**
   - Type that contains just one value

6. **Equality type**
   - Type to express the equality between two values

Don’t confuse this with instances of the Eq type class!
Proof Transformers

Obtaining a proof for $A$ from a proof of the conjunction $A \land B$

$fst : (a, b) \rightarrow a$

Obtaining a proof for the conclusion of an implication given the premise

$apply : (a \rightarrow b) \rightarrow a \rightarrow b$

And Elimination

$A \land B$

$\begin{array}{c}
A \\
\hline
B
\end{array}$

Implication elimination (Modus Ponens)

$A \Rightarrow B$

$\begin{array}{c}
A \\
\hline
B
\end{array}$
Proof Transformers

Obtaining a proof for the implication $A \Rightarrow B$ given a proof for $B$ assuming $A$

\[
\forall x : \mathit{e} \rightarrow b
\]

where $x$ of type $a$ may occur free in $e$

Implication introduction

\[
\begin{align*}
[A] & \\
\vdots & \\
B & \\
\hline
A \Rightarrow B
\end{align*}
\]
Piazza Question

Proof constructors
The Big Picture: Understanding Proofs in Idris

- **Dependent type**: Type that depends on values
- **Indexed type**: Type that is partitioned into non-overlapping subtypes by values (called “indexes”)
- **Singleton type**: Type that contains just one value
- **Equality type**: Type to express the equality between two values

- **Curry-Howard Isomorphism**
  - **Type: Proposition**: Value: Proof
  - **Function: Proof of implication**
  - **Function: Proof transformer**

Don’t confuse this with instances of the Eq type class!
Singleton Type = Indexed Type

- **Nat**

  - 0, 1, 2, 3, ...

Partitioning of type based on index values

- **Indexed type:** type that is being indexed
- **Index type:** type of index value

- **The**

  - **Singleton 0**
  - Index 0
  - Value

- **Singleton 1**
  - Index 1
  - Value

- **Singleton 2**
  - Index 2
  - Value

- **Singleton 3**
  - Index 3
  - Value

- **Data Singleton : Nat → Type where**

  - The : Singleton n

- **mkSingleton : (n : Nat) → Singleton n**

  - mkSingleton n = The

Indexed type: type that is being indexed

Dependent type: type depends on value
Computing with Singleton Types

data Singleton : Nat → Type where
  The : Singleton n

addSingletons : Singleton n → Singleton m → Singleton (n + m)
addSingletons {n} {m} The The = The

Computation happens as part of type checking

The function definition itself is trivial
(Arbitrary) Computation in Types

```idris
data Singleton : Nat → Type where
  The : Singleton n

facSingletons : Singleton n → Singleton (fac n)
addSingletons _ = The

fac : Nat → Nat
fac Z = 1
fac (S n) = (S n) * fac n
```
9. Define an Idris function \texttt{lift} for lifting an arbitrary \texttt{Nat \rightarrow Nat} function to work on singletons.

```
data Singleton : Nat \rightarrow Type where
    The : Singleton n

addSingletons : Singleton n \rightarrow Singleton m \rightarrow Singleton (n + m)
addSingletons The The = The
```
Exercises

10. Draw the type partition defined by the Even data type.

data Even : Bool \to Type where
  E0  : Even True
  E1  : Even False
  ESS : Even b \to Even b
Piazza Question

Indexed types
Vectors Are Indexed Types

data Vect : Nat → Type → Type where
  Nil : Vect Z a
  (::) : a → Vect n a → Vect (S n) a

{Diagram}

Index
Z

Vect Z Bool

[]

Type

Values

Index
S
Z

Vect 1 Bool

True :: []
False :: []

Type

Values

Index
S (S Z)

Vect 2 Bool

False :: True :: []
True :: True :: []
False :: False :: []
True :: False :: []

Type

Values
The Big Picture: Understanding Proofs in Idris

- Dependent type: Type that depends on values
- Indexed type: Type that is partitioned into non-overlapping subtypes by values (called “indexes”)
- Singleton type: Type that contains just one value
- Equality type: Type to express the equality between two values
- Curry-Howard Isomorphism:
  - Type: Proposition
  - Value: Proof
  - Function: Proof of implication
  - Function: Proof transformer

Don’t confuse this with instances of the Eq type class!
Proposition Types: Equality

Equality is a binary type constructor

\[
\text{data } (=) : a \to b \to \text{Type where}
\]
\[
\text{Refl} : x = x
\]

Every element of a type is equal to itself (reflexivity)

Refl is the name of the axiom; \(a\) and \(x\) are implicit arguments of \(\text{Refl}\)

To prove for the reverse function

\[
\text{Vect } (n + S\ Z)\ a = \text{Vect } (S\ n)\ a
\]

Example proposition and proof

\[
\text{two_is_two} : 2 = 2
\]
\[
\text{two_is_two} = \text{Refl}
\]

Name of the “Theorem”

The Proposition

Proof of the proposition
Understanding Propositional Equality

Why does this example work?

*The type checker evaluates (normalizes) value expressions and thus simplifies the arithmetic expression.*

Why are theorems correct?

*The type checker determines the type of the proof and makes sure it agrees with the defined type.*

```idris
data (=) : a → b → Type where
  Refl : x = x

two_is_two : 2 = 2
  two_is_two = Refl

idiom : 2 + 2 = 4
  idiom = Refl
```
Understanding Propositional Equality

Why two different types \( a \) and \( b \)?

(1) To be able to express equality between two values of different types and show that this is impossible

\[
\text{no_nonsense : } 2 = "2" \to \text{Void}
\]
\[
\text{no_nonsense} \text{ Refl impossible}
\]

Stay tuned …

(2) To be able to express equality between two values of potentially different types and show their equality

\[
\text{eq_length : } (xs : \text{Vect} \ n \ a) \to (ys : \text{Vect} \ m \ a) \to (xs = ys) \to n = m
\]
\[
\text{eq_length} \ _xs \ _ys \ _xs \ = \ _ys \ \text{Refl} = \text{Refl}
\]

data \( (\ = \) : \ a \to b \to \text{Type} \) where

\[
\text{Refl : } x = x
\]
11. Draw instances of indexed types for the type \((=)\).
Proof By Case Analysis

neg_cancel : not (not b) = b
neg_cancel = Refl

Doesn’t work! Idris can’t reduce the LHS, because not is defined by equations for specific cases

not : Bool → Bool
not True = False
not False = True

Revealing implicit argument in type is optional

neg_cancel : {b : Bool} → not (not b) = b

Provide a proof “case-by-case”

Pattern matching on implicit argument

neg_cancel : not (not b) = b
neg_cancel {b=True} = Refl
neg_cancel {b=False} = Refl
Proof By Case Analysis

\[ \text{neg_cancel : (b : Bool) \rightarrow not (not b) = b} \]
\[ \text{neg_cancel True = Refl} \]
\[ \text{neg_cancel False = Refl} \]

Making forall-quantified variable explicit

Pattern matching on explicit argument
Lifting Equality

\( (+) : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \)

\[
\begin{align*}
Z + m &= m \\
S n + m &= S(n + m)
\end{align*}
\]

\[\text{Z\_is\_right\_unit} : n + Z = n\]
\[\text{Z\_is\_right\_unit} = ???\]

No definition matches \( n + Z \); try different cases for \( n \)

We need some form of induction

We know (function property):

\[
\text{n + Z = n} \quad \Rightarrow \quad f(n + Z) = f(n)
\]

(Plan: Use this with \( f = S \))
**Congruence**

\[
\begin{align*}
Z_{\text{is_right_unit}} : n + \mathbb{Z} &= n \\
Z_{\text{is_right_unit}} \{n=\mathbb{Z}\} &= \text{Refl} \\
Z_{\text{is_right_unit}} \{n=S \ k\} &= ???
\end{align*}
\]

We know (function property):

\[
\begin{align*}
\text{Function property expressed as an Idris type}
\end{align*}
\]

\[
\begin{align*}
\text{cong} : \{f : t \rightarrow u\} \rightarrow a = b \rightarrow f \ a = f \ b \\
\text{cong Refl} = \text{Refl}
\end{align*}
\]

\[
\begin{align*}
N_{\text{is_right_unit}} : n + \mathbb{Z} &= n \\
N_{\text{is_right_unit}} \{n=\mathbb{Z}\} &= \text{Refl} \\
N_{\text{is_right_unit}} \{n=S \ k\} &= \text{cong} (N_{\text{is_right_unit}} \{n=k\})
\end{align*}
\]
Piazza Question

Congruence
Verifying Proofs

Defined name \(\approx\) Name of the Theorem

Type \(\approx\) The Proposition

Definition \(\approx\) Proof cases

Show for each case (= equation):
Type of RHS matches declared type.

\[
\begin{align*}
Z_{\text{is_right_unit}} : n + Z &= n \\
Z_{\text{is_right_unit}} \{n=Z\} &= \text{Refl} \\
Z_{\text{is_right_unit}} \{n=S~k\} &= \text{cong} \ (Z_{\text{is_right_unit}} \{n=k\})
\end{align*}
\]

Proof Checking \(\approx\) Type Checking
Verifying Proofs

\[ Z_{\text{is_right_unit}} : n + Z = n \]
\[ Z_{\text{is_right_unit}} \{n=Z\} = \text{Ref}l \]
\[ Z_{\text{is_right_unit}} \{n=S \ k\} = \text{cong} (Z_{\text{is_right_unit}} \{n=k\}) \]

\[ (+) : \text{Nat} \to \text{Nat} \to \text{Nat} \]
\[ Z + m = m \quad \text{(1)} \]
\[ S \ n + m = S \ (n + m) \quad \text{(2)} \]

\[ \text{Ref}l : x = x \]
\[ \text{cong} : x = y \to f \ x = f \ y \]

Computed Type (substitute \( Z \) for \( x \))

Declared Type (substitute \( Z \) for \( n \))

Inductive Hypothesis

Definition of \( \text{cong} \) with \( S \) for \( f \)
Back To Reverse ...

To do:

1. \( n + S Z = S n \)

2. \( \text{Vect} (n + S Z) \ a = \text{Vect} (S n) \ a \)

3. \[ \text{rev xs +++ [x]} : \text{Vect} (n + S Z) \ a \Rightarrow \text{rev xs +++ [x]} : \text{Vect} (S n) \ a \]

1. \( \text{lemma : n + (S Z) = S n} \)
2. \( \text{lemma } \{n=Z\} = \text{Refl} \)
3. \( \text{lemma } \{n=S \ k\} = \text{cong lemma } \{n=k\} \)
Exercises

12. Verify the $S \cdot k$ case of the lemma.
Proving the Type of Reverse

To do:

1. \[ n + S \, Z = S \, n \]

2. \[ Vect \left( n + S \, Z \right) a = Vect \left( S \, n \right) a \]

3. \[ \text{rev} \, xs \; +++ \; [x] : Vect \left( n + S \, Z \right) a \Rightarrow \text{rev} \, xs \; +++ \; [x] : Vect \left( S \, n \right) a \]

We know (substitution property):

\[ x = y \quad \Rightarrow \quad P(x) \Rightarrow P(y) \]

Plan: Use this with \( P \, x = Vect \, x \, a \) as \( P(\text{lemma}) \)

Substitution property expressed as an Idris type

\[ \text{replace} : \{P : a \rightarrow \text{Type}\} \rightarrow x = y \rightarrow P \, x \rightarrow P \, y \]

\[ \text{replace} \; \text{Refl} \, p = p \]
Finally: Fixing the Definition of Reverse

\[ \text{rev} : \text{Vect } n \ a \rightarrow \text{Vect } n \ a \]
\[ \text{rev} \ [\] = [] \]
\[ \text{rev} \ (x :: xs) = \text{replace} \ \{P = \text{vector}\} \ \text{lemma} \ (\text{rev} \ xs +++ [x]) \]

where \( \text{vector} : \text{Nat} \rightarrow \text{Type} \)
\( \text{vector } k = \text{Vect } k \ a \)

The property of being a vector of length \( k \)