"I think you should be more explicit here in step two."
Paths to Understanding Idris

Dependent Types, Termination

Haskell → Idris → Constructive Mathematics

Curry-Howard Isomorphism
Functional Programming in Idris

Dependent Types, Termination

Haskell → Idris
Idris vs. Haskell: Data Types

**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**

---

**data Bool : Type**
- **data Bool = True | False**

---

**data Nat : Type**
- **data Nat = Z | S Nat**

---

**data List : Type**
- **data [a] = [] | (∷) a [a]**

---

**data List a : Type**
- **data List a = [] | (∷) a (List a)**

---

**GADT notation is more general**

**Type constructor ≈ function on types**

**Type Parameter**

**Common result type for all constructors**

**(Complete) type of constructor**

**Only argument type of constructor**

**Same in Idris**
Idris vs. Haskell: Functions

**Idris**

\[ \forall a \]

\[ \text{not} :: \text{Bool} \rightarrow \text{Bool} \]
\[ \text{not True} = \text{False} \]
\[ \text{not False} = \text{True} \]

\[ \text{id} : \{a : \text{Type}\} \rightarrow a \rightarrow a \]
\[ \text{id} \ a \ x = x \]

\[ \text{true1} = \text{id} \ \text{True} \]
\[ \text{zero1} = \text{id} \ 0 \]

**Haskell**

\[ \text{not} :: \text{Bool} \rightarrow \text{Bool} \]
\[ \text{not True} = \text{False} \]
\[ \text{not False} = \text{True} \]

\[ \text{id} : (a:\text{Type}) \rightarrow a \rightarrow a \]
\[ \text{id} \ a \ x = x \]

\[ \text{true2} = \text{id} \ \text{Bool} \ \text{True} \]
\[ \text{zero2} = \text{id} \ \text{Nat} \ 0 \]

**Implicit Argument**

\[ \text{infixr 7 ++} \]

\[ (++) : \{a : \text{Type}\} \rightarrow \text{List a} \rightarrow \text{List a} \rightarrow \text{List a} \]
\[ [] \ + y s = y s \]
\[ (x :: x s) \ + y s = x :: (x s \ + y s) \]

**Explicit Type Argument**

\[ \forall a \]

\[ (++) : \{a : \text{Type}\} \rightarrow \text{List a} \rightarrow \text{List a} \rightarrow \text{List a} \]
\[ x :: x s \ + y s = x :: (x s \ + y s) \]
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
[ ]     !! n     = Nothing
(x :: xs) !! Z     = Just x
(_ :: xs) !! (S n) = xs !! n
```

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

```
(+++) : {a : Type} → List a → List a → List a
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```
Total Functions

Add at the top of all Idris modules

%default total

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

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

In Haskell

tail :: [a] → [a]
tail (_:xs) = xs

head :: [a] → a
head (x:_ = x
Piazza Question

2.1 Type Constructors
What is a Type Constructor?

A function that maps types to types

In Haskell

[] :: * → *
Maybe :: * → *
(->) :: * → * → *

Use ":kind Maybe" in ghci

A function that maps types and values to types

In Idris

List : Type → Type
Maybe : Type → Type
(->) : Type → Type → Type

Use ":t[ype] Maybe" in Idris

Vect : Nat → Type → Type
Segment : Nat → Type
Dependent Types

Type constructor

Value argument

Result type depends on argument value

Producing values of different types

Consuming values of different types

Single : Bool → Type
Single True  = Nat
Single False = List Nat

mkSingle : (b : Bool) → Nat → Single b
mkSingle True  x = x
mkSingle False x = [x]

Idris> mkSingle True 4
4 : Nat

Idris> mkSingle False 4

sum : (b : Bool) → Single b → Nat
sum True x       = x
sum False []      = 0
sum False (x::xs) = x + sum False xs
Vectors

\begin{align*}
\text{data} \ & \text{Vect} : \text{Nat} \to \text{Type} \to \text{Type} \ \\
& \text{where} \\
\text{Nil} : \text{Vect} \ Z \ a \\
(\:::) : a \to \text{Vect} \ n \ a \to \text{Vect} \ (S \ n) \ a \\
\end{align*}

Constructors are overloaded for List and Vect

Vectors of length n

A synonym for Nil

\begin{align*}
\text{data} \ & \text{List} : \text{Type} \to \text{Type} \ \\
& \text{where} \\
\text{[]} : \text{List} \ a \\
(\:::) : a \to \text{List} \ a \to \text{List} \ a \\
\end{align*}

xs = 1 :: 2 :: 7 :: []

ys = 9 :: xs

Vect 3 Nat

Vect 4 Nat

List Nat
Exercises

4. Define the Idris functions `eqNat` and `eqList` for comparing two natural numbers and two lists of values.

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

eqNat : Nat → Nat → Bool
eqNat Z     Z     = True
eqNat (S n) (S m) = eqNat n m
eqNat _     _     = False

eqList : Eq a => List a → List a → Bool
eqList []      []      = True
eqList (x::xs) (y::ys) = x==y && eqList xs ys
eqList _        _      = False
```
5. Define the Idris function `eqVect` for comparing two vectors of values.

\[
eq_{\text{List}} : \text{Eq a} \Rightarrow \text{List a} \rightarrow \text{List a} \rightarrow \text{Bool}
\]
\[
eq_{\text{List}} [] \hspace{1em} [] = \text{True}
\]
\[
eq_{\text{List}} (x::xs) \hspace{1em} (y::ys) = x==y \land \eqList xs ys
\]
\[
eq_{\text{List}} \_ \hspace{1em} \_ = \text{False}
\]
Piazza Question

2.2 Vector Comparison
Vector Size

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

size : Vect n a → Nat
size {n} _ = n

Alternative Definitions

size {n} _ = n
size {a} {n} _ = n
size {n} {a} _ = n

Implicit Argument

Result is computed

Result is read off the type
6. Define the functions `head` and `tail` for vectors.
7. Define a type for matrices with \( n \) rows and \( m \) columns using nested vectors.

\[
\text{Matrix} : \text{Nat} \to \text{Nat} \to \text{Type} \\
\text{Matrix} \; n \; m \; a = \text{Vect} \; n \; (\text{Vect} \; m \; a)
\]

8. Define the functions \text{firstRow} and \text{firstCol} for matrices.

\[
\text{firstRow} : \text{Matrix} \; (\text{S} \; n) \; m \; a \\
\quad \text{firstRow} \; (\text{xs} \; :: \; \_ ) = \text{xs}
\]

\[
\text{firstCol} : \text{Matrix} \; n \; (\text{S} \; m) \; a \\
\quad \text{firstCol} \; \text{xss} = \text{map} \; \text{head} \; \text{xss}
\]
Paths to Understanding Idris

- Haskell
- Idris
- Constructive Mathematics

Dependent Types, Termination

Curry-Howard Isomorphism
Proving with Idris

Idris

Constructive Mathematics

Curry-Howard Isomorphism
Reversing Vectors

\[
\text{rev : Vect } n \ a \rightarrow \text{Vect } n \ a
\]

\[
\text{rev } \text{[]} \quad = \quad \text{[]}
\]

\[
\text{rev } (x::xs) \quad = \quad \text{rev } \text{xs ++ } [x]
\]

**infixr 7 ++**

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

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

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

**Does not typecheck**

Type mismatch between

\[
\text{Vect } (\text{len } + 1) \ a \quad (\text{Type of } \text{rev } \text{xs ++ } [x])
\]

and

\[
\text{Vect } (\text{S len}) \ a \quad (\text{Expected type})
\]

Specifically:

Type mismatch between

\[
\text{plus } \text{len } 1
\]

and

\[
\text{S len}
\]
Reversing Vectors

\[
\text{rev} : \text{Vect} \ n \ a \to \text{Vect} \ n \ a
\]

\[
\text{rev} \ [\] = []
\]

\[
\text{rev} \ (x::xs) = \text{rev} \ xs ++ [x]
\]

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

Type mismatch between

\[
\text{Vect} \ (\text{len} + 1) \ a \quad \text{(Type of \ rev \ xs ++ [x])}
\]

and

\[
\text{Vect} \ (S \text{len}) \ a \quad \text{(Expected type)}
\]

Specifically:

Type mismatch between

\[
\text{plus} \ \text{len} \ 1
\]

and

\[
S \text{len}
\]
We know (RHS):

\[ \text{rev } [\ ] = [\ ] \]

\[ \text{rev } (x :: xs) = \text{rev } xs ++ [x] \]

We need (type declaration):

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

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

To do:

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

What does the type checker do?

Verifying type declaration

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 $T$.

**Curry-Howard Isomorphism**

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

Curry-Howard Isomorphism

1. Type: Proposition
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 by case analysis
Proof by induction
... later

Value: Proof
Type: Proposition
Dependent type
Type that depends on 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

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

**Curry-Howard Isomorphism**

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

- **Proof of Proposition A**
  - val : A

- **Witness for Proposition**
  - expression : Type

- **True**
  - True : Bool
  - "There are boolean values"

- **Z**
  - Z : Nat
  - "There are natural numbers"
The Big Picture: Understanding Proofs in Idris

Curry-Howard Isomorphism

1. Type: Proposition
2. Function: Proof of implication
3. Function: Proof transformer
4. Index type
5. Singleton type
6. Equality type

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

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

**Proof of Implication** \(A \Rightarrow B\)

\(\text{fun} : A \rightarrow B\)

**Witness for**

\(\text{even} : \text{Nat} \rightarrow \text{Bool}\)

“A boolean can be obtained from a number”

**Witness for**

\(\forall x \Rightarrow x\)

“Any proposition implies itself”

\(\forall x \Rightarrow x : a \rightarrow a\)

Proposition holds for any type; expressed through parametric polymorphism.
Proposition Types: Conjunction

Proof of Conjunction $A \land B$

\[(x, y) : (a, b)\]

\[(\text{even}, \lambda x \Rightarrow x)\]

Witness for

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

\[(\text{even}, \lambda x \Rightarrow x) : (\text{Nat} \to \text{Bool}, a \to a)\]
Piazza Question

2.3 Proposition types
Piazza Question

2.4 Logical connectives
What is a Proof?

Premise → proof step → Premise → proof step → Conclusion

Proof steps

Logical inference rules

```
A  B
---
A ∧ B

A  A ⇒ B
---
B
```

Proposition transformations

```
n = 3 + 1
---
n = 4
```

```
n + n = m + m
```

calculation

substitution
Proofs as Transformations of Propositions

Known

Premise

Premise

To be proved

proof step

Premise

Conclusion

Premise

Conclusion'

Premise

proof step

Known

To be proved
The Big Picture: Understanding Proofs in Idris

Curry-Howard Isomorphism

1. Type: Proposition
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. Equality type
   - Type to express the equality between two values
6. Singleton type
   - Type that contains just one value

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

- Value: Proof
- Type: Proposition
Proof Transformers

**Obtaining a proof for A from a proof of the conjunction A ∧ B**

\[ \text{fst} : (a,b) \rightarrow a \]

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

\[ \text{apply} : (a \rightarrow b) \rightarrow a \rightarrow b \]

---

**And Elimination**

\[ \frac{A \land B}{A} \]

**Implication elimination (Modus Ponens)**

\[ \frac{A \Rightarrow B \quad A}{B} \]
Proof Transformers

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

$\forall x \Rightarrow e : a \rightarrow b$

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

Implication introduction
Piazza Question

2.5 Proof transformers
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**

- **Indexed type**: type that is being indexed
- **Index type**: type of index value
- **Partitioning of type based on index values**

**Data Definition**

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

**Function Definition**

```idris
mkSingleton : (n : Nat) → Singleton n
mkSingleton n = The
```
Computing with Singleton Types

```idris
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

```
addSingletons : Singleton n → Singleton m → Singleton (n + m)
addSingletons The The = The
```

The function definition itself is trivial

```
addSingletons : Singleton n → Singleton m → Singleton (n + m)
addSingletons _ _ = The
```
(Arbitrary) Computation in Types

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

fac : Nat → Nat
fac Z     = 1
fac (S n) = (S n) * fac n
```

```
facSingletons : Singleton n → Singleton (fac n)
facSingletons _ = The
```
9. Define an Idris function `lift` for lifting an arbitrary `Nat → Nat` function to work on singletons.

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

addSingletons : Singleton n → Singleton m → Singleton (n + m)
addSingletons The The = The
```
10. Draw the type partition defined by the **Even** data type.

```idris
data Even : Bool → Type where  
E0  : Even True  
E1  : Even False  
ESS : Even b → Even b
```
2.6 Indexed types
Vectors Are Indexed Types

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

**Index**

- Z
- S Z
- S (S Z)

**Type**

- Vect Z Bool
- Vect 1 Bool
- Vect 2 Bool

**Values**

- []
- True :: []
- False :: []
- False :: True :: []
- True :: True :: []
- False :: False :: []
- True :: False :: []
The Big Picture: Understanding Proofs in Idris

Curry-Howard Isomorphism

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

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

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

Equality is a binary type constructor

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

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

To prove for the reverse function

n + S Z = S n
Vect (n + S Z) a = Vect (S n) a

Example proposition and proof

two_is_two : 2 = 2
two_is_two = Refl

Refl is the name of the axiom;
a and x are implicit arguments of Refl

The Proposition

Name of the “Theorem”

Proof of the proposition
Understanding Propositional Equality

<table>
<thead>
<tr>
<th>data (=) : a → b → Type where</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refl : x = x</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>obvious : 2 + 2 = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>obvious = Refl</td>
</tr>
</tbody>
</table>
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

```idris
no_nonsense : 2 = "2" → Void
no_nonsense Refl impossible
```

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

```idris
eq_length : (xs : Vect n a) → (ys : Vect m a) → (xs = ys) → n = m
eq_length xs _ Refl = Refl
```
11. Draw instances of indexed types for the type (=).
Proof By Case Analysis

Proof that negation cancels:

\[ \neg (\neg b) = b \]

\[ \text{negCancelable} : \forall b. (\neg (\neg b)) = b \]

- \( \text{negCancelable} \{ b = \text{True} \} = \text{Refl} \)
- \( \text{negCancelable} \{ b = \text{False} \} = \text{Refl} \)

Idris can't reduce the LHS, because \( \neg \) is defined by equations for specific cases:

\[ \neg : \text{Bool} \to \text{Bool} \]

- \( \neg \text{True} = \text{False} \)
- \( \neg \text{False} = \text{True} \)

Provide a proof "case-by-case":

- \( \text{negCancelable} : \{ b : \text{Bool} \} \to \neg (\neg b) = b \)

Revealing implicit argument in type is optional.

Pattern matching on implicit argument.
Proof By Case Analysis

Making forall-quantified variable explicit

neg_cancel : (b : Bool) → not (not b) = b
neg_cancel True  = Refl
neg_cancel False = Refl

Pattern matching on explicit argument
Lifting Equality

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

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

We need some form of induction

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

We know (function property):

\[ n + Z = n \Rightarrow f(n + Z) = f(n) \]

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

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

We know (function property):
\[ n + Z = n \Rightarrow f(n + Z) = f(n) \]

Function property expressed as an Idris type:
\[ \text{cong} : \{f : t \rightarrow u\} \rightarrow a = b \rightarrow f \ a = f \ b \]
\[ \text{cong Refl} = \text{Refl} \]

\[ \text{Z\_is\_right\_unit} : n + Z = n \]
\[ \text{Z\_is\_right\_unit} \{n=Z\} = \text{Refl} \]
\[ \text{Z\_is\_right\_unit} \{n=S \ k\} = \text{cong} \ (\text{Z\_is\_right\_unit} \{n=k\}) \]
\[ \text{cong} \{f=S\} \text{Z\_is\_right\_unit} \]

\[ (+) : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \]
\[ Z \ + \ m = m \]
\[ S \ n + m = S \ (n + m) \]
Piazza Question

2.7 Congruence
## Verifying Proofs

### Defined name

Name of the Theorem

### Type

The Proposition

### Definition

Proof cases

<table>
<thead>
<tr>
<th>Defined name</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>n + Z = n</td>
<td>Z is right unit</td>
<td>n + Z = n</td>
</tr>
<tr>
<td>n = Z</td>
<td>Z is right unit</td>
<td>n = Z</td>
</tr>
<tr>
<td>n = S k</td>
<td>Z is right unit</td>
<td>n = S k</td>
</tr>
</tbody>
</table>

**Show for each case (= equation):**

Type of RHS matches declared type.

**Proof Checking**

Type Checking
Verifying Proofs

\[ Z\_is\_right\_unit : n + Z = n \]
\[ Z\_is\_right\_unit \{n=Z\} = Refl \]
\[ Z\_is\_right\_unit \{n=S k\} = cong (Z\_is\_right\_unit \{n=k\}) \]

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

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

---

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

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

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

\[ Z\_is\_right\_unit \{n=S k\} \]
\[ = \text{cong} (Z\_is\_right\_unit \{n=k\}) \sim S \ k + Z = S \ k \]
\[ = \text{cong} (p : k + Z = k) \sim S \ (k + Z) = S \ k \]
\[ = q : S \ (k + Z) = S \ k \]

**Inductive Hypothesis**

- Definition of \( Z\_is\_right\_unit \) with \( S \) for \( f \)

**Substituting \( S \ k \) for \( n \)**

**Types must be equal**

- \( Z \) = \( Z \)
- \( Z + Z = Z \)
- \( S \ k + Z = S \ k \)
- \( S \ (k + Z) = S \ k \)

**Definition of +**

- \( Z + m = m \)
- \( S \ n + m = S \ (n + m) \)

**Definition of 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. \( \text{lemma} : n + (S Z) = S n \)
   
   \( \text{lemma} \ \{n=Z\} = \text{Refl} \)

   \( \text{lemma} \ \{n=S \ k\} = \text{cong lemma} \ \{n=k\} \)
Exercises

lemma : n + (S Z) = S n
lemma {n=Z} = Refl
lemma {n=S k} = cong lemma {n=k}

(+) : Nat → Nat → Nat
Z + m = m
S n + m = S (n + m)

cong : x = y → f x = f y

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

To do:

1. \[ \text{lemma} : n + (S \, Z) = S \, n \]
   \[ n + S \, Z = S \, n \]

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

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

We know (substitution property):

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

Plan: Use this with \( P \, x = \text{Vect} \, x \, a \) 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]) \\
\text{where} \ \text{vector} : \text{Nat} \rightarrow \text{Type} \\
\text{vector } k = \text{Vect } k \ a
\]

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