2 Idris

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

- Dependent Types, Termination
- Curry-Howard Isomorphism
Functional Programming in Idris

Dependent Types, Termination

Haskell → Idris
Idris vs. Haskell: Data Types

**Idris**

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

**Haskell**

```haskell
data Bool = True | False
```

Idris notation is more general

**Idris**

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

**Haskell**

```haskell
data Nat = Z | S Nat
```

Type constructor ≈ function on types

**Idris**

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

**Haskell**

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

Type Parameter

Common result type for all constructors

(Complete) type of constructor

Only argument type of constructor

Same in Idris

Type Parameter
Idris vs. Haskell: Functions

\( \forall a \)

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

\[
\begin{align*}
[&] & \rightarrow & ys = ys \\
(x :: xs) & ++ ys = x :: (xs ++ ys)
\end{align*}
\]

\[
\begin{align*}
\text{id} : \{a : \text{Type}\} & \rightarrow a \rightarrow a \\
\text{id } x & = x \\
\text{true1} & = \text{id } \text{True} \\
\text{zero1} & = \text{id } 0
\end{align*}
\]

\[
\begin{align*}
\text{id} : (a : \text{Type}) & \rightarrow a \rightarrow a \\
\text{id } a \ x & = x \\
\text{true2} & = \text{id } \text{Bool } \text{True} \\
\text{zero2} & = \text{id } \text{Nat } 0
\end{align*}
\]

\[\text{not : Bool } \rightarrow \text{Bool} \]

\[
\begin{align*}
\text{not True} & = \text{False} \\
\text{not False} & = \text{True}
\end{align*}
\]

\[\text{not} : \text{Bool } \rightarrow \text{Bool} \]

\[
\begin{align*}
\text{not True} & = \text{False} \\
\text{not False} & = \text{True}
\end{align*}
\]

\[
\begin{align*}
\text{false} & : \text{Bool} \\
\text{true} & : \text{Bool}
\end{align*}
\]

\[\text{true} \equiv \text{id } \text{True} \]

\[\text{false} \equiv \text{id } \text{False} \]

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

\[\text{true} : \text{Bool} \\
\text{false} : \text{Bool}
\]

\[\text{false} = \text{id } \text{True} \]

\[\text{false} = \text{id } \text{Bool } \text{True} \]

\[\text{true} = \text{id } \text{False} \]

\[\text{false} = \text{id } \text{Nat } 0 \]

\[\text{true} = \text{id } \text{Bool } \text{False} \]

\[\text{false} = \text{id } \text{Nat } 1 \]

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

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

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

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

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

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

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

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

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

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

\[\text{id} : (a : \text{Type}) \rightarrow a \rightarrow a \\
\text{id } a \ x = x
\]
1. Define the Idris function (!!): for extracting the nth element from a list (use zero for first element).

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

For CS 582: All functions in Idris should be total!

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

%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:

- \( [\cdot] \) :: \( * \to * \)
- \( \text{Maybe} \) :: \( * \to * \)
- \( (\to) \) :: \( * \to * \to * \)

Use \text{"kind Maybe"} in ghci

In Idris:

- \( \text{Vect} \) : \( \text{Nat} \to \text{Type} \to \text{Type} \)
- \( \text{Segment} \) : \( \text{Nat} \to \text{Type} \)
- \( \text{List} \) : \( \text{Type} \to \text{Type} \)
- \( \text{Maybe} \) : \( \text{Type} \to \text{Type} \)
- \( (\to) \) : \( \text{Type} \to \text{Type} \to \text{Type} \)

Use \text{"t[type] Maybe"} in Idris
Dependent 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]

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

*Idris> mkSingle True 4
  4 : Nat
*Idris> mkSingle False 4
Vectors

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

- Type Parameter
- Type Index
- Constructors are overloaded for List and Vect
- Vectors of length \( n \)
- A synonym for Nil

\[\text{data \ List : Type} \rightarrow \text{Type} \quad \text{where}\]
\[[] : \text{List} \ a\]
\[(::) : a \rightarrow \text{List} \ a \rightarrow \text{List} \ a\]

\[\text{xs} = 1 \ :: \ 2 \ :: \ 7 \ :: \ []\]
\[\text{ys} = 9 \ :: \ \text{xs}\]

\[\text{Vect} \ 3 \ \text{Nat}\]
\[\text{Vect} \ 4 \ \text{Nat}\]

\[\text{List \ Nat}\]
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.

```
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

2.2 Vector Comparison
Vector Size

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

Result is computed

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

Alternative Definitions

Works even if the type signature is:
```
size : Vect m a → Nat
```
Exercises

6. Define the functions head and tail for vectors.

```
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. 

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

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

\[
\text{firstRow} : \text{Matrix} \ (S \ n) \ m \ a \to \text{Vect} \ m \ a \\
\text{firstRow} (xs \cons \_ \_ \_ \_) = xs \\
\text{firstCol} : \text{Matrix} \ n \ (S \ m) \ a \to \text{Vect} \ n \ a \\
\text{firstCol} \ xss = \text{map} \ \text{head} \ xss
\]
Paths to Understanding Idris

Dependent Types, Termination

Haskell → Idris → Constructive Mathematics

Curry-Howard Isomorphism
Proving with Idris

Idris

Constructive Mathematics

Curry-Howard Isomorphism
Reversing Vectors

rev : Vect n a → Vect n a
rev []   = []
rev (x::xs) = rev xs +++ [x]

infixr 7 +++
(++) : Vect n a → Vect m a → Vect (n + m) a
(++) [] ys = ys
(++) (x::xs) ys = x :: (xs +++ ys)

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

Type mismatch between
Vect (len + 1) a (Type of rev xs +++ [x])
and
Vect (S len) a (Expected type)

Specifically:
Type mismatch between
plus len 1
and
S len
Reversing Vectors

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

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

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

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

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

Type mismatch between

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

and

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

Specifically:

Type mismatch between

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

and

\[
S \ \text{len}
\]
Convincing the Type Checker ...

**We know (RHS):**

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

**We need (type declaration):**

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

**To do:**

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

**Prove to Idris (!):**

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

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

What does the type checker do?

Verifying type declaration

name : Type
name = expression

Is this true?

expression : Type

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

- **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. **Function: Proof of implication**
3. **Function: Proof transformer**
4. **Proof by case analysis**
5. **Proof by induction**

... later
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!
Proposition Types: Basic Propositions

Proof of Proposition A

- \text{val : A}

Witness for Proposition

- \text{expression : Type}

- \text{True : Bool}
  - "There are boolean values"

Witness for

- \text{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. **Indexed type**
5. **Singleton type**
6. **Equality type**

**Type that depends on values**

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

**Type that contains just one value**

**Type to express the equality between two values**

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

Proposition holds for any type; expressed through parametric polymorphism

**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"

- $\lambda x \Rightarrow x$

"Any proposition implies itself"

- $\lambda x \Rightarrow x : a \rightarrow a$
Proposition Types: Conjunction

Proof of Conjunction $A \land B$

(x, y) : (a, b)

Witness for

"A boolean can be obtained from a number"
and "Any proposition implies itself"
Piazza Question

2.3 Proposition types
Piazza Question

2.4 Logical connectives
What is a Proof?

Proof steps

Premise

proof step

Premise

proof step

Conclusion

Logical inference rules

Premise

Premise

proof step

Conclusion

Proposition transformations

### Logical inference rules

- **Conjunction rule**
  
  \[
  \frac{A \quad B}{A \land B}
  \]

- **Implication introduction**
  
  \[
  \frac{A}{A \implies B}
  \]

### Proposition transformations

- **Calculation**
  
  \[
  \frac{n = 3 + 1}{n = 4}
  \]

- **Substitution**
  
  \[
  \frac{n \quad n + n = m + m}{n = m}
  \]
Proofs as Transformations of Propositions

- Known
  - Premise
  - ... 
  - Premise

- To be proved
  - Proof step
  - Conclusion

- Known
  - Premise
  - ... 
  - Conclusion’

- To be proved
  - Premise
The Big Picture: Understanding Proofs in Idris

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
7. Dependent type
   - Type that depends on 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 \( A \land B \)

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

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

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

And Elimination

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

Implication elimination (Modus Ponens)

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

$\begin{array}{c}
[A] \\
\vdots \\
B \\
\hline
A \Rightarrow B
\end{array}$
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**

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!
Singleton Type = Indexed Type

Partitioning of type based on index values

Indexed type: type that is being indexed

Index type: type of index value

Nat

0 1 2 3 4 ...

Data

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

mkSingleton : (n : Nat) \rightarrow Singleton n
\quad mkSingleton n = The

Dependent type: type depends on value
Computing with Singleton Types

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

addSingletons : Singleton n \rightarrow Singleton m \rightarrow 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

\[
\text{data Singleton : Nat} \rightarrow \text{Type where}
\]
\[
\text{The : Singleton n}
\]

\[
facSingletons : \text{Singleton n} \rightarrow \text{Singleton (fac n)}
facSingletons _ = \text{The}
\]

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

\[
\text{data Singleton : Nat } \rightarrow \text{ Type where}
\]
\[
\text{The : Singleton } n
\]
\[
\text{addSingletons : Singleton } n \rightarrow \text{ Singleton } m \rightarrow \text{ Singleton } (n + m)
\]
\[
\text{addSingletons The The } = \text{ The}
\]
10. Draw the type partition defined by the `Even` data type.
Piazza Question

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

Values

- []
- [True :: [], False :: []]
- [False :: [], True :: []]
- [False :: [False :: []], True :: [False :: []]]
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

- **Type**: Proposition
- **Value**: Proof
- **Function**: Proof of implication
- **Function**: Proof transformer
Proposition Types: Equality

Equality is a binary type constructor

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

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

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

To prove for the reverse function

\[ n + S Z = S n \]

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

Example proposition and proof

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.

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

   \[
   \text{no_nonsense} : 2 = "2" \rightarrow \text{Void}
   \]

   \[
   \text{no_nonsense} \ 	ext{Refl} \ \text{impossible}
   \]

   Stay tuned …

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

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

   \[
   \text{eq\_length} \ \text{xs} \ _\ \text{Refl} = \text{Refl}
   \]
I. Draw instances of indexed types for the type $(\approx)$. 

\[
\text{data } (\approx) : a \to b \to \text{Type where} \\
\text{Refl} : x = x \\
\text{two_is_two} : 2 = 2 \\
\text{two_is_two} = \text{Refl}
\]
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
Proof By Case Analysis

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

Making forall-quantified variable explicit

Pattern matching on explicit argument
Lifting Equality

\[ Z_{is\_right\_unit} : n + Z = n \]
\[ Z_{is\_right\_unit} = ??? \]

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

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

We need some form of induction

We know (function property):

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

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

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

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

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

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

2.7 Congruence
Verifying Proofs

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

Type \(\approx\) The Proposition

Definition \(\approx\) Proof cases

\[
\begin{align*}
Z\_is\_right\_unit : n + Z &= n \\
Z\_is\_right\_unit \{n=Z\} &= \text{Refl} \\
Z\_is\_right\_unit \{n=S\ k\} &= \text{cong} (Z\_is\_right\_unit \{n=k\})
\end{align*}
\]

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

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

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

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

\[ \text{Refl} : x = x \]

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

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

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

Inductive Hypothesis

Definition of +

Definition of \( \text{cong} \) with \( S \) for \( f \)

Types must be equal

must be shown to be equal
Back To Reverse ...

To do:

1. \[ n + S \mathbb{Z} = S n \]

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

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

lemma : \( n + (S \; \mathbb{Z}) = S \; n \)
lemma \{n=Z\} = Refl
lemma \{n=S \; k\} = cong lemma \{n=k\}
Exercises

 lemma : n + (S Z) = S n
 lemma {n=Z}    = Reflect
 lemma {n=S k} = congr lemma {n=k}

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

12. Verify the \( S^k \) case of the lemma.
Proving the Type of Reverse

To do:

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

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

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

We know (substitution property):

\[
\text{x} = \text{y} \quad \Rightarrow \quad \text{P(x)} \Rightarrow \text{P(y)}
\]

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

Substitution property expressed as an Idris type

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

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

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

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

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

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

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

where \(\text{vector} : \text{Nat} \to \text{Type}\)

\[
\text{vector}\ k = \text{Vect}\ k\ a
\]

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

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

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