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

- Haskell
- Idris
- Constructive Mathematics

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

Dependent Types, Termination

Haskell ➔ Idris
Idris vs. Haskell: Data Types

**Idris**

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

  - Type constructor ≈ function on types
  - Type Parameter
  - GADT notation is more general

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

  - `Type Constructor = Nat`
  - `Type Parameter = a`

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

  - `Type constructor ≈ function on types`

**Haskell**

- `data Bool = True | False`

  - (Complete) type of constructor

- `data Nat = Z | S Nat`

  - Only argument type of constructor

- `data [a] = [] | (∷) a [a]`

  - Same in Idris

- `data List a = [] | (∷) a (List a)`
Idris vs. Haskell: Functions

\[ \forall \ a \ \text{infixr} \ 7 \ \text{++} \]
\[ (++): \{a: \text{Type}\} \to \text{List } a \to \text{List } a \to \text{List } a \]
\[ [] \ \text{++} \ ys = ys \]
\[ (x :: xs) \ \text{++} \ ys = x :: (xs \ \text{++} \ ys) \]

\[ \text{id} : \{a: \text{Type}\} \to a \to a \]
\[ \text{id } x = x \]
\[ \text{true1} = \text{id } \text{True} \]
\[ \text{zero1} = \text{id } 0 \]

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

\[ \text{id} : (a: \text{Type}) \to a \to a \]
\[ \text{id } a \ x = x \]
\[ \text{true2} = \text{id } \text{Bool } \text{True} \]
\[ \text{zero2} = \text{id } \text{Nat } 0 \]
1. Define the Idris function `(!!)` for extracting the nth element from a list (use zero for first element).
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∷_ ) = 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?

A function that maps types to types

In Haskell

```
[ ]    :: * → *
Maybe :: * → *
(→)   :: * → * → *
```

Use "kind Maybe" in ghci

A function that maps types \textit{and} values to types

In Idris

```
Vect    : Nat → Type → Type
Segment : Nat → Type
List    : Type → Type
Maybe   : Type → Type
(→)     : Type → Type → Type
```

Use "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

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

- **Type Index**
- **Type Parameter**

Constructors are overloaded for `List` and `Vect`

- Vectors of length `n`

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

A synonym for `Nil`

```
x = 1 :: 2 :: 7 :: []
y = 9 :: x
```

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

`ys = 9 :: x`
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

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
size _ _ = n

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

Result is computed
Result is read off the type
Implicit Argument
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

Haskell → Idris

Curry-Howard Isomorphism

Idris → Constructive Mathematics
Proving with Idris

Idris

Constructive Mathematics

Curry-Howard Isomorphism
Reversing Vectors

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

Infixr 7 +++

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

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

Specifically:

Type mismatch between
\[
\text{plus } k \ 1
\]
and
\[
S \ k
\]
Reversing Vectors

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

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

Type mismatch between
\[
\text{Vect } (S \ k) \text{ a}
\]
and
\[
\text{Vect } (k + 1) \text{ a}
\]

Specifically:

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

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

We know (RHS):

```
rev xs +++ [x] : Vect (n + 1) a
```

We need (type declaration):

```
rev xs +++ [x] : Vect (S n) a
```

To do:

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

Prove (to Idris (!)): 

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

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

4. \( rev xs +++ [x] : Vect (S n) a \)
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 \( T \).

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

- 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

**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!
Proposition Types: Basic Propositions

**Proof of Proposition A**

- val : A

**Witness for Proposition**

- expression : Type

**Witness for**

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

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

**Curry-Howard Isomorphism**

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

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

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

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

Witness for

even : Nat → Bool
“A boolean can be obtained from a number”

Witness for

\x ⇒ x
“Any proposition implies itself”

Proposition holds for any type; expressed through parametric polymorphism

fun : A → B
Proof of Implication A ⇒ B

\x ⇒ x : a → a
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”

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

Proposition types
Piazza Question

Logical connectives
What is a Proof?

Premise

proof step

Premise

proof step

Conclusion

Proof steps

Logical inference rules

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

Proposition transformations

\[ \frac{n = 3 + 1}{n = 4} \quad \frac{n = m}{n + n = m + m} \]

calculation

substitution
Proofs as Transformations of Propositions

Known

Premise

Premise

Premise

To be proved

Conclusion

proof step

Premise

Premise

Premise

Known

To be proved

Premise

Conclusion’

Premise

proof step
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

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

1. Value: Proof
2. Dependent type
Type that depends on values
3. Singleton type
Type that contains just one value
Proof Transformers

Obtaining a proof for \( A \) from a proof of the conjunction \( A \land B \):

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

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):

\[
\frac{A \Rightarrow B}{A} \quad \frac{A}{B}
\]
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 $e$

[\[A\]
: 
B
\]

$\Rightarrow A \Rightarrow B$

Implication introduction
Piazza Question

Proof transformers
The Big Picture: Understanding Proofs in Idris

Value:
- Proof

Type: Proposition

Function: Proof of implication

Function: Proof transformer

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

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

Dependent type

Indexed type

Singleton type

Equality type

Curry-Howard Isomorphism

Idris
Singleton Type = Indexed Type

- **Nat**
- **Singleton**
- **Index**
- **Value**

**Partitioning of type based on index values**

**Indexed type:** type that is being indexed

**Index type:** type of index value

**data Singleton : Nat → Type where**

The : Singleton n

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

mkSingleton n = The

**Dependent type:** type depends on value
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

The function definition itself is trivial
Computation happens as part of type checking
```
(Arbitrary) Computation in Types

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

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

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

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

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

two_is_two : 2 = 2
\[
two\_is\_two = \text{Refl}
\]
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

\[
\text{no}_\text{nonsense} : \text{2 = "2" \to Void} \\
\text{no}_\text{nonsense} \text{ Refl impossible}
\]

Stay tuned …

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

\[
\text{eq}_\text{length} : (\text{xs : Vect n a}) \to (\text{ys : Vect m a}) \to (\text{xs = ys}) \to n = m \\
\text{eq}_\text{length} \text{ xs _ Refl = Refl}
\]
11. Draw instances of indexed types for the type (=).

data (=) : a → b → Type where
  Refl : x = x
  two_is_two : 2 = 2
  two_is_two = Refl
Proof By Case Analysis

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

**Pattern matching on implicit argument**

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

Provide a proof “case-by-case”

```idris
neg_cancel : {b : Bool} → not (not b) = b
neg_cancel {b=True} = Refl
neg_cancel {b=False} = Refl
```

**Revealing implicit argument in type is optional**

```idris
not : Bool → Bool
not True = False
not False = True
```
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

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

\[ (\text{Plan: Use this with } f = S) \]
Z_is_right_unit : n + Z = n
Z_is_right_unit {n=Z} = Refl
Z_is_right_unit {n=S k} = ???

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

cong : \( \{ f : t \to u \} \to a = b \to f a = f b \)
cong Refl = Refl

cong {f=S} (Z_is_right_unit {n=k})
Piazza Question

Congruence
Verifying Proofs

 Defined name ≈ Name of the Theorem

\[\text{Z_is_right_unit : } n + \mathbb{Z} = n\]
\[\text{Z_is_right_unit } \{n=\mathbb{Z}\} = \text{Refl}\]
\[\text{Z_is_right_unit } \{n=S\ k\} = \text{cong } (\text{Z_is_right_unit } \{n=k\})\]

Type ≈ The Proposition

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

Definition ≈ Proof cases

Proof Checking ≈ Type Checking
Verifying Proofs

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

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

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

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

\[
\begin{align*}
Z_{\text{is\_right\_unit}} \{n=S\ k\} &= \text{cong} (Z_{\text{is\_right\_unit}} \{n=k\}) \sim S\ k + Z = S\ k \\
\text{cong} : x = y &\to f\ x = f\ y
\end{align*}
\]
Back To Reverse ...

To do:

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

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

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

To do:

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

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

```
replace : {P : a → Type} → x = y → P x → P y

lemma : n + (S Z) = S n

rev : Vect n a → Vect n a
rev [] = []
rev (x :: xs) = replace {P = vector} lemma (rev xs +++ [x])
   where vector : Nat → Type
         vector k = Vect k a
```

The property of being a vector of length $k$