Practical examples of writing programs and proving theorems in Idris.

Donovan Crichton

January 2020

Preliminaries

Propositional Logic

- Concerned with statements of verifiable facts.
- Used daily by programmers when reasoning about Boolean values.

Symbol	Meaning	Example
T, F	True, False	Boolean values.
$p, q, r, \ldots$	Propositions	Let $p =$ "It is raining."
$\neg$	Negation (Not)	$\neg p$
$\wedge$	Conjunction (And)	$p \wedge q$
$\vee$	Disjunction (Or)	$p \vee q$
$\rightarrow$	Implication (If)	$p \rightarrow q$
$\leftrightarrow$	Bi Implication (Iff)	$p \leftrightarrow q$
$\equiv$	Equivalence	$p \equiv q$
$\top$	Tautology	$p \vee \neg p \equiv \top$
$\bot$	Contradiction	$p \wedge \neg p \equiv \bot$

Definitions of Connectives

Conjunction (And)

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction (Or)

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Negation (Not)

p	$\neg p$
T	F
F	T

Implication (If)

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Bi Implication (Iff)

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

Logical Equivalence

p	q	$p \equiv q$
T	T	T
T	F	F
F	T	F
F	F	T

Proof Techniques

By Exhaustion

Idea: Prove by enumerating all possible cases.

Prove: $(\neg p \vee q) \leftrightarrow (p \rightarrow q)$.

p	q	$\neg p$	$\neg p \vee q$	$p \rightarrow q$	$(\neg p \vee q) \leftrightarrow (p \rightarrow q)$
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Proof Techniques

By Appeal to Lemma

Idea: Introduce pre-proven smaller proofs (called a Lemma) to prove a larger proof.

- **Lemma 1.** $p \lor \neg p \equiv \top$.
- **Lemma 2.** $(p \equiv q) \equiv (p \leftrightarrow q)$.

Prove: $(p \leftrightarrow q) \lor \neg(p \equiv q) \leftrightarrow \top$.

$$
\begin{aligned}
(p \leftrightarrow q) \lor \neg(p \equiv q) \leftrightarrow \top & \qquad \text{Premise.} \\
(p \equiv q) \lor \neg(p \equiv q) \equiv \top & \qquad \text{Lemma 2.} \\
\top \equiv \top & \qquad \text{Lemma 1.}
\end{aligned}
$$

$\square$

First Order (or Predicate) Logic

Extends propositional logic from reasoning about propositions to reasoning about sets.

Symbol	**Meaning**	**Example**
$X, Y, Z, \ldots$	Set Variables	Let $Y = \{2, 3, 4\}$.
$x, y, z, \ldots$	Member Variables	Let $z = 2$.
$P(x), Q(y), \ldots$	Predicate Variables	Let $Q(y) = y > 1$.
$\forall x \in X, P(x)$	Universal Quantifier	$\forall y \in Y, Q(y)$
$\exists x \in X, P(x)$	Existential Quantifier	$\exists z \in Y, z = 2$

Proof Techniques

Induction

- Allows us to prove that a property $P(x)$ holds $\forall x \in X$. Provided X is well-founded.
- Informally well-founded means "no infinite decreasing chains".

Prove: $\forall n \in \mathbb{N}(\exists y \in \mathbb{N}, y = n + 1)$

$$\begin{aligned} y &= 0 + 1 && \text{Base Case. } n = 0 \\ &= 1 && \square \end{aligned}$$

$$\begin{aligned} y &= (k + 1) + 1 && \text{Inductive Step. } n = k + 1 \\ &= k + 2 && \square \end{aligned}$$

Why should I care?

- Types are just sets with flavour!
- **Bool** $= \{True, False\}$
- **Int** $= \{-\infty, ..., -2, -1, 0, 1, 2, 3, ..., \infty\}$
- Mixing of flavours is not allowed!
- $\{True, -2, "Hello", 1\}$ Can really only be said to be a "thing" flavoured set.

Propositions as Types. Proofs as Programs

- The Curry-Howard-Lambeck correspondence is well known amongst Haskell programmers for the correspondence between categories and programming.
- The correspondence with logic is less often discussed.
- Holds for any language that is based on a typed lambda calculus.

Idea: A type is a proposition.

What is Truth?

Propositional Logic and Predicate Logic consider truth to be the Boolean value "True". These logics also have a notion of vacuous truth.

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

In Predicate Logic: $\forall x \in \{\} P(x)$ is also true.

- If a type is a proposition, what does it mean for it to be true?
- A type is true iff it is inhabited with a value.

Curry-Howard in Idris

Logic Term	Logic Symbol	Idris Symbol	Idris Type
Implication	$p \Rightarrow q$	`p -> q`	Arrow
Conjunction	$p \wedge q$	`(p, q)`	Pair (Product)
Disjunction	$p \vee q$	`Either p q`	Enum (Sum)
Negation	$\neg\ p$	`p -> Void`	Void Type
IFF/Eq	$p \Leftrightarrow q$, $p \equiv q$	`(p -> q, q -> p)`	Pair Arrows
Universal	$\forall\ x.\ P\ x$	`p -> Type`	Π Type
Existential	$\exists\ x.\ P\ x$	`(x ** P x)`	Σ Type
$=$	$=$	`p = q`	Type Equality
$\top$	True	`()`	Unit Type
$\bot$	False	`Void`	Uninhabited

Natural Numbers

Let $\mathbb{N}$ denote the set of natural numbers where:
1. Zero (0) is a natural number.
2. If k is a natural number, then the successor of k is also a natural number.

```
data Nat : Type where
  Z : Nat
  S : (k : Nat) -> Nat
```

- **Nat : Type** is the type constructor.
- **S** and **K** are the value constructors.

The equality GADT

```
data (=) : (x : A) -> (y : B) -> Type where
  Refl : x = x
```

- Data type to represent constructive equality.
- Allows intentional (sic) equality, not extensional equality.

Proving commutativity on addition in Idris

```
(+) : Nat -> Nat -> Nat
Z     + y = Z
(S k) + y = S (k + y)
```

$\forall x, y \in \mathbb{N}. x + y = y + x$

```
plusIsCommutative : (x, y : Nat) -> x + y = y + x
plusIsCommutative x y = ?what
```

$\frac{x,y:Nat}{what:x+y=y+x}$

```
plusIsCommutative : (x, y : Nat) -> x + y = y + x
plusIsCommutative Z y = ?t1
plusIsCommutative (S k) y = ?t2
```

$\frac{x,y:Nat}{t1:y=y+Z}$

$\frac{x,y:Nat}{t2:S(k+y)=y+(S\ k)}$

Introducing Lemmas in Idris

$$\frac{x, y : Nat}{t1 : y = y + Z}$$

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 Z = ?lemhole1
lemma1 (S k) = ?lemhole2
```

$$\frac{}{lemhole1 : Z = Z}$$

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 X = Refl
lemma1 (S k) = ?lemhole2
```

$$\frac{k : Nat}{lemhole2 : S\ k = S\ (k + Z)}$$

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 Z = Refl
lemma1 (S k) =
  let rec = lemma1 k
  in ?lemhole2
```

$$\frac{k : Nat \quad rec : k = k + Z}{lemhole2 : S\ k = S\ (k + Z)}$$

The rewrite rule

- replaces one side of an equality in the goal, with an equality sharing the same side given as an argument.

$$\frac{k{:}Nat \quad rec{:}k=k+Z}{lemhole2{:}S\ k=S\ (k+Z)}$$

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 Z = Refl
lemma1 (S k) =
  let rec = lemma1 k
  in rewrite rec in ?lemhole2
```

$$\frac{k{:}Nat \quad rec{:}k=k+Z \quad \_rewrite\_rule{:}k=k+Z}{lemhole2{:}S\ (k+Z)=S\ (k+Z)}$$

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 Z = Refl
lemma1 (S k) =
  let rec = lemma1 k
  in rewrite rec in Refl
```

Back to our original proof attempt!

```
plusIsCommutative : (x, y : Nat) -> x + y = y + x
plusIsCommutative Z y = ?t1
plusIsCommutative (S k) y = ?t2
```

$\frac{x,y : Nat}{t1 : y = y + Z}$

$\frac{x,y : Nat}{t2 : S (k+y) = y + (S\ k)}$

```
plusIsCommutative : (x, y : Nat) -> x + y = y + x
plusIsCommutative Z y = lemma1 y
plusIsCommutative (S k) y = ?t2
```

$\frac{x,y : Nat}{t2 : S (k+y) = y + (S\ k)}$

```
lemma2 : (x, y : Nat) -> S (k + y) = y + (S k)
lemma2 Z y = ?lemhole1
lemma2 (S k) y = ?lemhole2
```

$\frac{y : Nat}{lemhole1 : S\ y = S\ y}$

$\frac{k,y : Nat}{lemhole2 : S\ (S\ (k+y)) = S\ (k + (S\ y))}$

Completing the second lemma.

```
lemma2 : (x, y : Nat) -> S (k + y) = y + (S k)
lemma2 Z y = Refl
lemma2 (S k) y = ?lemhole2
```

$$\frac{k, y : Nat}{lemhole2 : S\ (S\ (k+y)) = S\ (k + (S\ y))}$$

```
lemma2 : (x, y : Nat) -> S (k + y) = y + (S k)
lemma2 Z y = Refl
lemma2 (S k) y =
  let rec = lemma2 k y
  in ?lemhole2
```

$$\frac{k, y : Nat \quad rec : S\ (k+y) = k + S\ y}{lemhole2 : S\ (S\ (k+y)) = S\ (k + (S\ y))}$$

```
lemma2 : (x, y : Nat) -> S (k + y) = y + (S k)
lemma2 Z y = Refl
lemma2 (S k) y =
  let rec = lemma2 k y
  in rewrite rec in ?lemhole2
```

$$\frac{k, y : Nat \quad rec : S\ (k+y) = k + S\ y \quad \_rewrite\_rule : S\ (k+y) = k + S\ y}{lemhole2 : S\ (k + (S\ y)) = S\ (k + (S\ y))}$$

Our completed proof

```
lemma1 : (x : Nat) -> x = x + Z
lemma1 Z = Refl
lemma1 (S k) =
  let rec = lemma1 k
  in rewrite rec in Refl

lemma2 : (x, y : Nat) -> S (k + y) = y + (S k)
lemma2 Z y = Refl
lemma2 (S k) y =
  let rec = lemma2 k y
  in rewrite rec in Refl

plusIsCommutative : (x, y : Nat) -> x + y = y + x
plusIsCommutative Z y = lemma1 y
plusIsCommutative (S k) y =
  let rec = plusIsCommutative k y
  in rewrite rec in lemma2 y k
```

Theorem Proving in Anger

Given a random number, return a 'random' element of a given list.

```
randElem : (n : Nat) -> (xs : List a) -> a
randElem x xs = index (mod n (length xs)) xs

When checking argument ok to function index
Can't find v/ of type InBounds (mod n (length xs)) xs
```

Lets check out the index function then...

```
index : (n : Nat) -> (xs : List a) ->
        {auto ok : InBounds n xs} -> a
```

Idris tries to automatically generate a value of the `InBounds n xs` type.

Propositions as (actual) Types

```
data InBounds : (k : Nat) -> (xs : List a) -> Type where
  InFirst : InBounds 0 (x :: xs)
  InLater : InBounds k xs -> InBounds (S k) (x :: xs)

data LTE : (n : Nat) -> (m : Nat) -> Type where
  LTEZero : LTE Z m
  LTESucc : LTE n m -> LTE (S n) (S m)

data NonEmpty : (xs : List a) -> Type where
  IsNonEmpty : NonEmpty (x :: xs)

randElem : (n : Nat) -> (xs : List a) -> a
randElem x xs = index (mod n (length xs)) xs
```

Over 100 lines of lemmas later...

Why are we doing this again?

Clearly proving these theorems is a lot of work just to get an index function working properly!

- A stronger guarantee of functional correctness than unit testing.
- Proofs only need to be written once, and can be quite general.
- Proofs are usually erased at compile-time in Idris unless they're explicitly needed at run-time.
- Mission critical, safety critical, or financial critical systems.
- How about those laws?...

A correct definition of Functor

```
interface GeorgeFunctor (f : Type -> Type) where
  gmap : (a -> b) -> f a -> f b
  gIdentLaw : (xs : (f a)) ->
              gmap Prelude.Basics.id xs = id xs
  gCompLaw : (xs : (f a)) -> (g : b -> c) ->
             (h : a -> b) ->
             gmap (g . h) xs = (gmap g . gmap h) xs
```

Correct List Implementations

```
implementation GeorgeFunctor List where
  gmap f [] = []
  gmap f (x :: xs) = f x :: gmap f xs

  gIdentLaw [] = Refl
  gIdentLaw (x :: xs) =
    let prf = gIdentLaw (x :: xs)
    in rewrite prf in Refl

  gCompLaw [] g h = Refl
  gCompLaw (x :: xs) g h =
    let prf = gCompLaw (x :: xs) g h
    in rewrite prf in Refl
```

Just to make sure it wasn't a fluke...

```
implementation GeorgeFunctor Maybe where
  gmap f Nothing = Nothing
  gmap f (Just x) = Just (f x)

  gIdentLaw Nothing = Refl
  gIdentLaw (Just x) = Refl

  gCompLaw Nothing g f = Refl
  gCompLaw (Just x) g f = Refl
```

In Summary...

- We can directly map proofs in mathematical logic to proofs in dependently typed functional languages.
- We can write correct definitions of category theoretic concepts such as function, monad, applicative etc.
- Writing proofs can be arduous at first, until you build up the intuition.
- Proving things gives us the strongest guarantee of correctness possible.
- More research needs to be undertaken regarding proofs around monads, particularly the IO monad.