Functional Programming
&
Formal Proof

2015级软件工程 薛明淇
vinaleux@gmail.com

Today's Topics

- Sample language: Idris
- Basic functional programming concepts
- A glimpse into logic systems
- Rudimentary reasoning and proofs
- Reflections in the real world

DO ASK QUESTIONS PLEASE

Just Enough Functional Programming

by explaining terms

type

```
Int
Nat
Bool
Nat → Bool
```

function

pure function

```
plus : (Nat, Nat) → Nat
```

higher-order function

```
sort : (NatList, (Nat, Nat) → Bool) → NatList
```

currying

```
plus : Nat → (Nat → Nat)

sort : Nat list → ((Nat → (Nat → Bool)) → Nat list)
```

algebraic data type

Algebraic Data Type

- product type: Aggregation of different types (`struct` in C).
- sum type: Union of different types.

```
data NatAndBool : Type where -- Product type of Nat and Bool
  NB : Nat → Bool → NatAndBool

NatAndBool' : Type
NatAndBool' = (Nat, Bool)

data NatOrBool : Type where -- Sum type of Nat and Bool
  N : Nat  → NatOrBool
  B : Bool → NatOrBool


data Shape : Type where
  Triangle  : Double → Double → Shape -- base and height
  Rectangle : Double → Double → Shape -- length and width
  Circle    : Double → Shape          -- radius

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

recursive type

```
data NatList : Type where
  Nil  : NatList
  Cons : Nat → NatList → NatList
```

parameterized type

generic type

Parameterized Type

```
data List : Type → Type where
  Nil  : List a
  Cons : a → List a → List a

data Sum : Type → Type → Type where
  Left  : a → Sum a b
  Right : b → Sum a b

data Product : Type → Type → Type where
  MkProduct : a → b → Product a b

Product' : Type → Type → Type
Product' a b = (a, b)

sort : List a → (a → a → Bool) → List a
```

pattern matching

```
length : List a → Nat
length l = case l of
  Nil ⇒ 0
  Cons h t ⇒ 1 + length t

length Nil = 0
length (Cons h t) = 1 + length t
```

So … what are we proving?

Classical Propositional Logic Revisit

- Conjunction ("and"): if A and B are both **true**, then $A \land B$ is **true**
- Disjunction ("or"): if either A or B is **true**, then $A \lor B$ is **true**
- Implication: if A and B are both **true** or A is **false**, then $A \to B$ is **true**

Constructive View of Propositional Logic

INTRODUCTION RULES

- Conjunction: if A and B are both **provable**, then $A \land B$ is **provable**
- Disjunction: if either A or B is **provable**, then $A \lor B$ is **provable**
- Implication: if B is **provable** under the assumption that A is **provable**, then $A \rightarrow B$ is **provable**

ELIMINATION RULES

- Conjunction: if $A \land B$ is **provable**, then A is **provable**, B is **provable**
- Disjunction: if $A \lor B$ is **provable**, $A \rightarrow C$ is **provable** and $B \rightarrow C$ is **provable**, then C is **provable**
- Implication: if $A \rightarrow B$ is **provable** and A is **provable**, then B is **provable**

Connecting logic with programming

CURRY-HOWARD CORRESPONDENCE

- propositions are types
- proofs are values

```
And : Type → Type → Type
And a b = (a, b)

Or : Type → Type → Type
Or a b = Sum a b

Imply : Type → Type → Type
Imply a b = a → b
```

```
andIntro : a → b → And a b
andIntro a b = (a, b)

andElim1 : And a b → a
andElim1 = fst -- fst : (a, b) → a

andElim2 : And a b → b
andElim2 = snd -- snd : (a, b) → b

orIntro1 : a → Or a b
orIntro1 = Left  -- Left  : a → Sum a b

orIntro2 : b → Or a b
orIntro2 = Right -- Right : b → Sum a b

orElim : Or a b → Imply a c → Imply b c → c
orElim ab ac bc = case ab of
  Left  a ⇒ ac a
  Right b ⇒ bc b

implyElim : Imply a b → a → b
implyElim ab a = ab a
```

Example: Our First Proof

Distributivity of Disjunction Over Conjunction:

A ∨ (B ∧ C) → (A ∨ B) ∧ (A ∨ C)

```
orDistr : (Or a (And b c)) → (And (Or a b) (Or a c))
orDistr assumption =
  orElim assumption orDistrL orDistrR

orDistrL : a → And (Or a b) (Or a c)
orDistrL a = andIntro (orIntro1 a) (orIntro1 a)

orDistrR : And b c → And (Or a b) (Or a c)
orDistrR bc = andIntro
  (orIntro2 (andElim1 bc))
  (orIntro2 (andElim2 bc))
```

More Powerful Logic and Type System

Predicate Logic and Quantifiers

UNIVERSAL QUANTIFIER

- Introduction: if `P` is **provable** for an arbitrary value `a` of type `T`, then `∀ a:T. P` is **provable**
  - `∀ n: Nat. n = n`
  - `∀ n: Nat. Even(n) ∨ Odd(n)`
- Elimination: if `∀ a:T. P` is **provable**, and there's some value `a'` of type `T`, then `[a'/a]P` is **provable**
  - `∀ n: Nat. n = n → x: Nat → x = x`
  - `∀ n: Nat. Even(n) ∨ Odd(n) → x: Nat → Even(x) ∨ Odd(x)`

~~EXISTENTIAL QUANTIFIER~~

Universal Quantifier and Parameterized Type

QUANTIFY Proposition AND Type

```
∀ A: Proposition. ∀ B: Proposition. A ∧ B → B ∧ A
```

```
-- And : Type → Type → Type
andCommute : {A : Type} → {B : Type} → And A B → And B A
andCommute (a, b) = (b, a)
```

QUANTIFY Nat

```
∀ n: Nat. n = n
```

```
data (=) : Nat → Nat → Type where
  -- ...

natEqualReflexive : (n : Nat) → n = n
```

Predicate and Dependent Type

BOOLEAN PREDICATE

```
nonEmpty : List a → Bool
nonEmpty Nil        = False
nonEmpty (Cons h t) = True

natEqual : Nat → Nat → Bool
```

TYPE-LEVEL PREDICATE

```
data NonEmpty : List a → Type where
  IsNonEmpty : (h : a) → (t : List a) → NonEmpty (Cons h t)

data (=) : Nat → Nat → Type
```

INTRODUCTION RULE FOR `NonEmpty`

- For all type `A`, for all `h` of type `A`, for all `t` of type `List A`, `NonEmpty (Cons h t)` is **provable**

Type-level Equality

- Introduction: for all type `T`, all value `t` of type `T`, `t = t` is **provable**
- Elimination: if `a = b` is **provable**, and `P(a)` is **provable** for some predicate `P`, then `P(b)` is **provable**
- Equality, as a relation, is **reflexive**, **symmetric** and **transitive**

```
data (=) : {A : Type} → A → A → Type where
  Refl : {A : Type} → {a : A} → a = a

natEqualReflexive : (n : Nat) → n = n
natEqualReflexive n = Refl

sym : a = b → b = a
sym ab = rewrite ab in Refl

trans : a = b → b = c → a = c
trans ab bc = rewrite ab in rewrite bc in Refl
```

Reason About Natural Numbers

Peano Arithmetic

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

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

plusLeftIdentity : (n : Nat) → Z + n = n
plusLeftIdentity n = Refl

plusRightIdentity : (n : Nat) → n + Z = n
plusRightIdentity n = Refl
-- Error: type mismatch between
--     n = n     (Type of Refl)
-- and n + Z = n (Expected Type)
```

Prove by Induction

`∀ n: Nat. n + 0 = n`

- Base case: when `n = 0`, `0 + 0 = 0` holds
- Induction step: when `n = S n’` and assuming `n’ + 0 = n’`.
  `S n’ + 0 = S (n’ + 0) = S n’`.

Q.E.D.

```
plusRightIdentity : (n : Nat) → n + Z = n
plusRightIdentity Z = Refl
plusRightIdentity (S n) =
  rewrite plusRightIdentity n in Refl
```

Example: + is commutative

```
plusCommute : (m : Nat) → (n : Nat) → m + n = n + m
plusCommute Z n = sym (plusRightIdentity n)
plusCommute (S m) n =
  -- S (m + n) = n + S m
  rewrite plusCommute m n in
  -- S (n + m) = n + S m
  rewrite plusRightS n m in
  -- S (n + m) = S n + m
  Refl

plusRightS : (m : Nat) → (n : Nat) → m + S n = S m + n
plusRightS Z n = Refl
plusRightS (S m) n = rewrite plusRightS m n in Refl
```

Structural Induction and Termination Check

WHY DON'T WE DO THIS?

```
plusCommute : (m : Nat) → (n : Nat) → m + n = n + m
plusCommute m n = plusCommute m n
```

```
anything : {a : Type} → a
anything = anything
```

INDUCTION STEP REVISIT

```
plusRightIdentity (S n) =
  rewrite plusRightIdentity n in Refl

plusCommute (S m) n =
  rewrite plusCommute m n in
  rewrite plusRightS n m in Refl

length (Cons h t) = 1 + length t
```

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

data List : Type → Type where
  Nil  : List a
  Cons : a → List a → List a
```

Recursively-Defined Proposition

TYPE-LEVEL ORDERING

```
data (≤) : Nat → Nat → Type where
  LeN : {n : Nat} → n ≤ n
  LeS : {m : Nat} → {n : Nat} → m ≤ n → m ≤ S n

zeroLeN : (n : Nat) → Z ≤ n
zeroLeN Z = LeN
zeroLeN (S n) = LeS (zeroLeN n)

nLeS : {m : Nat} → {n : Nat} → m ≤ n → S m ≤ S n
nLeS LeN = LeN
nLeS (LeS m_le_pn) = LeS (nLeS m_le_pn)

leRelax : {m : Nat} → {n : Nat} → S m ≤ n → m ≤ n
leRelax LeN = LeS LeN
leRelax (LeS sm_le_pn) = LeS (leRelax sm_le_pn)
```

Extra Example: simple property of + and ⩽

```
∀ a b c d : Nat. a + b = c + d → a ⩽ c ∨ b ⩽ d
```

```
plusLeSplit : (a : Nat) → (b : Nat) → (c : Nat) → (d : Nat) →
              a + b = c + d → Or (a ⩽ c) (b ⩽ d)
plusLeSplit Z b c d eq = orIntro1 (zeroLeN c)
plusLeSplit a b Z d eq = orIntro2 (plusLe a b d eq)
plusLeSplit (S a') b (S c') d eq =
  case plusLeSplit a' b c' d (succInjective (a' + b) (c' + d) eq) of
       Left a'_le_c' ⇒ orIntro1 (nLeS a'_le_c')
       Right b_le_d  ⇒ orIntro2 b_le_d

plusLe : (a : Nat) → (b : Nat) → (c : Nat) →
         a + b = c → b ⩽ c
plusLe Z b c b_eq_c = rewrite b_eq_c in LeN
plusLe (S a') b c eq = leRelax -- S b ⩽ c → b ⩽ c
  -- induction on a
  (plusLe a' (S b) c
    -- a' + S b = c
    (replace {P = \p ⇒ p = c} (sym (plusRightS a' b)) eq))
```

Falsehood Proof

Logical Negation

CLASSICAL LOGIC

- If A is **false**, then ¬ A is **true**.

CONSTRUCTIVE LOGIC

- Introduction: If A is “not provable”(?), then ¬ A is **provable**
- Elimination: if ¬ A is **provable**, then …?

⊥ (Bottom) Proposition (Type)

- Introduction: **NONE**
- Elimination: If ⊥ is **provable**, for all proposition P, P is **provable**

```
data Void : Type where
  -- no constructor

void : {a : Type} → Void → a
void v impossible
```

Negation in Constructive Logic

- ¬ P is **defined as** (**definitionally equivalent** to) P → ⊥
- Elimination of implication: If ¬ P is **provable** and P is **provable**, ⊥ is **provable** (therefore everything is provable).

```
Not : Type -> Type
Not P = P -> Void

contradiction : Not (And p (Not p))
-- contradiction: And p (p → Void) → Void
contradiction (p, notp) = notp p


discriminateNat : {n : Nat} -> Not (Z = S n)
discriminateNat Refl impossible

notEqualCommute : Not (a = b) -> Not (b = a)
notEqualCommute not_ab ba = not_ab (sym ba)
```

Limitation of Constructive Logic

LAW OF EXCLUDED MIDDLE ???

```
∀ P: Proposition. P ∨ ¬ P
```

DOUBLE NEGATION ELIMINATION ???

```
∀ P: Proposition. ¬ ¬ P → P
```

PROPERTY OF QUANTIFIERS ???

```
∀ P. (¬ ∀ a. P(a) → ∃ x. ¬ P(x))
```

But we can do …

```
∀ P: Proposition. P → ¬ ¬ P

truthIrrefutable : {P : Type} → P → Not (Not P)
-- curry : ((a, b) → c) → a → b → c
-- contradiction    : (P, not P) → Void
-- truthIrrefutable : P → not P → Void
truthIrrefutable = curry contradiction


∀ P: Proposition. ¬ ¬ (P v ¬ P)

loemIrrefutable : {P : Type} → Not (Not (Or P (Not P)))
loemIrrefutable not_loem = contradiction (notLoemContra not_loem)

notLoemContra : {P : Type} →
                Not (Or P (Not P)) → And (Not P) (Not (Not P))
notLoemContra not_loem = (notp, notnotp)
  where notp    p  = not_loem (orIntro1 p)
        notnotp np = not_loem (orIntro2 np)
```

Epilogue

Further Readings

FUNCTIONAL PROGRAMMING

- Functional Programming in Scala, Paul Chiusano & Runa Bjarnason
- Wikibook of Haskell

DEPENDENT TYPE & PROOF

- Type-driven Development with Idris, Edwin Brady
- Verified Functional Programming in Agda, Aaron Stump
- Software Foundations (in Coq)

THEORY

- Type Theory and Functional Programming, Simon Thompson