

# Foundations of Software Fall 2021

Week 10

# Different Kinds of Maps

What is missing?

$$\begin{array}{lll} \textit{Term} & \rightarrow & \textit{Term} \quad (\lambda x.t) \\ \textit{Type} & \rightarrow & \textit{Term} \quad (\Lambda X.t) \end{array}$$

# Different Kinds of Maps

What is missing?

|             |               |             |                 |
|-------------|---------------|-------------|-----------------|
| <i>Term</i> | $\rightarrow$ | <i>Term</i> | $(\lambda x.t)$ |
| <i>Type</i> | $\rightarrow$ | <i>Term</i> | $(\Lambda X.t)$ |
| <i>Type</i> | $\rightarrow$ | <i>Type</i> | ???             |
| <i>Term</i> | $\rightarrow$ | <i>Type</i> | ???             |

Agenda today:

- ▶ Type operators
- ▶ Dependent types

# Type Operators and System $F_\omega$

# Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

# Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

*Type operators* are functions at type-level.

$$\lambda X :: K. T$$

# Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

*Type operators* are functions at type-level.

$$\lambda X :: K. T$$

Two Problems:

- ▶ Type checking of type operators
- ▶ Equivalence of types

# Kinding

Problem: avoid meaningless types, like *MkFun[Int, String]*.



# Kinding

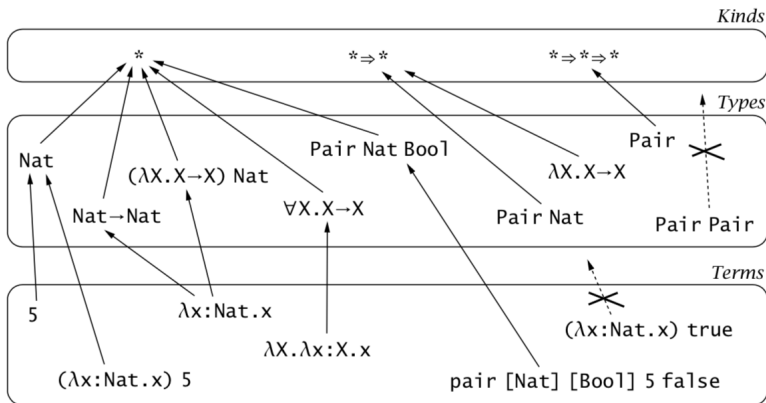
Problem: avoid meaningless types, like *MkFun[Int, String]*.

- \* proper types, e.g. *Bool*, *Int*  $\rightarrow$  *Int*
- \*  $\Rightarrow$  \* type operators: map proper type to proper type
- \*  $\Rightarrow$  \*  $\Rightarrow$  \* two-argument operators
- ( \*  $\Rightarrow$  \* )  $\Rightarrow$  \* type operators: map type operators to proper types

# Kinding

Problem: avoid meaningless types, like *MkFun[Int, String]*.

- \* proper types, e.g. *Bool*, *Int*  $\rightarrow$  *Int*
- \*  $\Rightarrow$  \* type operators: map proper type to proper type
- \*  $\Rightarrow$  \*  $\Rightarrow$  \* two-argument operators
- ( \*  $\Rightarrow$  \* )  $\Rightarrow$  \* type operators: map type operators to proper types



# Equivalence of Types

Problem: all the types below are equivalent

$$\begin{array}{lll} \text{Nat} \rightarrow \text{Bool} & \text{Nat} \rightarrow \text{Id Bool} & \text{Id Nat} \rightarrow \text{Id Bool} \\ \text{Id Nat} \rightarrow \text{Bool} & \text{Id}(\text{Nat} \rightarrow \text{Bool}) & \text{Id}(\text{Id}(\text{Id Nat} \rightarrow \text{Bool})) \end{array}$$

We need to introduce *definitional equivalence* relation on types, written  $S \equiv T$ . The most important rule is:

$$(\lambda X :: K.S) T \equiv [X \mapsto T]S \quad (\text{Q-APPABS})$$

And we need one typing rule:

$$\frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T} \quad (\text{T-EQ})$$

## First-class Type Operators

Scala supports passing type operators as argument:

```
def makeInt[F[_]](f: () => F[Int]): F[Int] = f()
```

```
makeInt[List]() => List[Int](3)
```

```
makeInt[Option]() => None
```

First-class type operators supports *polymorphism* for type operators, which enables more patterns in type-safe functional programming.

# System $F_\omega$ — Syntax

Formalizing first-class type operators leads to System  $F_\omega$ :

$t ::= \dots$  *terms*  
 $\lambda X :: K. t$  *type abstraction*

$T ::=$  *types*  
 $X$  *type variable*  
 $T \rightarrow T$  *type of functions*  
 $\forall X :: K. T$  *universal type*  
 $\lambda X :: K. T$  *operator abstraction*  
 $T \ T$  *operator application*

$K ::=$  *kinds*  
 $*$  *kind of proper types*  
 $K \Rightarrow K$  *kind of operators*

## System $F_\omega$ — Semantics

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{t_1 \ t_2 \longrightarrow t_1 \ t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:T_1.t_1) \ v_2 \longrightarrow [x \mapsto v_2]t_1 \quad (\text{E-APPABS})$$

$$\frac{t \longrightarrow t'}{t \ [T] \longrightarrow t' \ [T]} \quad (\text{E-TAPP})$$

$$(\lambda X::K.t_1) \ [T] \longrightarrow [X \mapsto T]t_1 \quad (\text{E-TAPPTABS})$$

## System $F_\omega$ — Kinding

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad (\text{K-TVAR})$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 : K_2}{\Gamma \vdash \lambda X :: K_1. T_2 :: K_1 \Rightarrow K_2} \quad (\text{K-ABS})$$

$$\frac{\Gamma \vdash T_1 : K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 : K_1}{\Gamma \vdash T_1 T_2 :: K_2} \quad (\text{K-APP})$$

$$\frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad (\text{K-ARROW})$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1. T_2 :: *} \quad (\text{K-ALL})$$

## System $F_\omega$ — Type Equivalence

$$T \equiv T \qquad \frac{T \equiv S}{S \equiv T} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \qquad (\text{Q-ARROW})$$

$$\frac{S_2 \equiv T_2}{\forall X::K_1. S_2 \equiv \forall X::K_1. T_2} \qquad (\text{K-ALL})$$

$$\frac{S_2 \equiv T_2}{\lambda X::K_1. S_2 \equiv \lambda X::K_1. T_2} \qquad (\text{Q-ABS})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \ S_2 \equiv T_1 \ T_2} \qquad (\text{Q-APP})$$

$$(\lambda X::K. T_1) \ T_2 \equiv [X \mapsto T_2] T_1 \qquad (\text{Q-APPAbs})$$



## System $F_\omega$ — Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : S \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : T} \quad (\text{T-APP})$$

$$\frac{\Gamma, X::K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X::K_1. t_2 : \forall X::K_1. T_2} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t : \forall X::K. T_2 \quad \Gamma \vdash T :: K}{\Gamma \vdash t [T] : [X \mapsto T] T_2} \quad (\text{T-TAPP})$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad (\text{T-EQ})$$

## Example

```
type PairRep[Pair :: * ⇒ * ⇒ *] = {  
  pair : ∀X.∀Y.X → Y → (Pair X Y),  
  fst  : ∀X.∀Y.(Pair X Y) → X,  
  snd  : ∀X.∀Y.(Pair X Y) → Y  
}
```

```
def swap[Pair :: * ⇒ * ⇒ *, X :: *, Y :: *]  
  (rep : PairRep Pair)  
  (pair : Pair X Y) : Pair Y X  
=  
  let x = rep.fst [X] [Y] pair in  
  let y = rep.snd [X] [Y] pair in  
  rep.pair [Y] [X] y x
```

The method *swap* works for any representation of pairs.

# Properties

*Theorem [Preservation]:* if  $\Gamma \vdash t : T$  and  $t \longrightarrow t'$ , then  $\Gamma \vdash t' : T$ .

*Theorem [Progress]:* if  $\vdash t : T$ , then either  $t$  is a value or there exists  $t'$  with  $t \longrightarrow t'$ .

# Dependent Types

# Why Does It Matter?

Example 1. Track length of vectors in types:

$$NVec \quad :: \quad Nat \rightarrow *$$
$$first \quad : \quad (n:Nat) \rightarrow NVec \ (n + 1) \rightarrow Nat$$

$(x:S) \rightarrow T$  is called **dependent function type**. It is impossible to pass a vector of length 0 to the function *first*.

# Why Does It Matter?

Example 1. Track length of vectors in types:

$$NVec \quad :: \quad Nat \rightarrow *$$
$$first \quad : \quad (n:Nat) \rightarrow NVec \ (n + 1) \rightarrow Nat$$

$(x:S) \rightarrow T$  is called **dependent function type**. It is impossible to pass a vector of length 0 to the function *first*.

Example 2. Safe formatting for *sprintf*:

$$sprintf \quad : \quad (f:Format) \rightarrow Data(f) \rightarrow String$$
$$Data([]) \quad = \quad Unit$$
$$Data("%d" :: cs) \quad = \quad Nat * Data(cs)$$
$$Data("%s" :: cs) \quad = \quad String * Data(cs)$$
$$Data(c :: cs) \quad = \quad Data(cs)$$

## Dependent Function Type (a.k.a. $\Pi$ Types)

A dependent function type is inhabited by *a dependent function*:

$$\lambda x:S.t \quad : \quad (x:S) \rightarrow T$$

## Dependent Function Type (a.k.a. $\Pi$ Types)

A dependent function type is inhabited by *a dependent function*:

$$\lambda x:S.t \quad : \quad (x:S) \rightarrow T$$

If  $T$  does not depend on  $x$ , it degenerates to function types:

$$(x:S) \rightarrow T = S \rightarrow T \quad \text{where } x \text{ does not appear free in } T$$



# The Calculus of Constructions

# The Calculus of Constructions: Syntax

$t ::=$

$s$

$x$

$\lambda x:t.t$

$t \ t$

$(x:t) \rightarrow t$

*terms*

*sort*

*variable*

*abstraction*

*application*

*dependent type*

$s ::=$

$*$

$\square$

*sorts*

*sort of proper types*

*sort of kinds*

$\Gamma ::=$

$\emptyset$

$\Gamma, x:T$

*contexts*

*empty context*

*term variable binding*

The semantics is the usual  $\beta$ -reduction.

# The Calculus of Constructions: Typing

$$\vdash * : \square \text{ (T-AXIOM)} \qquad \frac{x:T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S. t : (x:S) \rightarrow T} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : (x:S) \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 \ t_2 : [x \mapsto t_2] T} \text{ (T-APP)}$$

$$\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash T : s_2}{\Gamma \vdash (x:S) \rightarrow T : s_2} \text{ (T-PI)}$$

$$\frac{\Gamma \vdash t : T \quad T \equiv T' \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \text{ (T-CONV)}$$

The equivalence relation  $T \equiv T'$  is based on  $\beta$ -reduction.

## Four Kinds of Lambdas

| Example  | Type   |
|--|--|
| $\lambda x:\mathbb{N}.x + 1$                       | $\mathbb{N} \rightarrow \mathbb{N}$                          |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ |

## Four Kinds of Lambdas

| Example   | Type  |
|---|---|
| $\lambda x:\mathbb{N}.x + 1$                            | $\mathbb{N} \rightarrow \mathbb{N}$   |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$      | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$                  |
| $\lambda X:*. \lambda x:X. x$                           | $(X:*) \rightarrow X \rightarrow X$   |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$ | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |

# Four Kinds of Lambdas

| Example   | Type  |
|---|---|
| $\lambda x:\mathbb{N}.x + 1$                            | $\mathbb{N} \rightarrow \mathbb{N}$   |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$      | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$                  |
| $\lambda X:*. \lambda x:X. x$                           | $(X:*) \rightarrow X \rightarrow X$   |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$ | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |
| $\lambda X:*.X$   | $* \rightarrow *$   |
| $\lambda F:* \rightarrow *.F\ \mathbb{N}$               | $(* \rightarrow *) \rightarrow *$   |

## Four Kinds of Lambdas

| Example  | Type  |
|--|---|
| $\lambda x:\mathbb{N}.x + 1$                               | $\mathbb{N} \rightarrow \mathbb{N}$   |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$         | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$                  |
| $\lambda X:*. \lambda x:X. x$                              | $(X:*) \rightarrow X \rightarrow X$   |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$    | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |
| $\lambda X:*.X$  | $* \rightarrow *$   |
| $\lambda F:* \rightarrow *.F\ \mathbb{N}$                  | $(* \rightarrow *) \rightarrow *$   |
| $\lambda n:\mathbb{N}.NVec\ n$                             | $\mathbb{N} \rightarrow *$  |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.NVec\ (f\ 6)$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow *$                           |

# Strong Normalization

Given the following  $\beta$ -reduction rules

$$\frac{t_1 \longrightarrow t'_1}{\lambda x: T_1. t_1 \longrightarrow \lambda x: T_1. t'_1} \quad (\beta\text{-ABS})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2} \quad (\beta\text{-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{t_1 \ t_2 \longrightarrow t_1 \ t'_2} \quad (\beta\text{-APP2})$$

$$(\lambda x: T_1. t_1) t_2 \longrightarrow [x \mapsto t_2] t_1 \quad (\beta\text{-APPAbs})$$

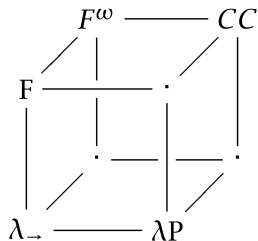
*Theorem [Strong Normalization]:* if  $\Gamma \vdash t : T$ , then there is no infinite sequence of terms  $t_i$  such that  $t = t_1$  and  $t_i \longrightarrow t_{i+1}$ .



# Pure Type Systems

$$\frac{\Gamma \vdash S : s_i \quad \Gamma, x:S \vdash T : s_j}{\Gamma \vdash (x:S) \rightarrow T : s_j} \quad (\text{T-PI})$$

| System                  | $(s_i, s_j)$ |            |  |   |
|-------------------------|--------------|------------|--|---|
| $\lambda_{\rightarrow}$ | {            | $(*, *)$   |  | } |
| $\lambda P$             | {            | $(*, *)$ , | $(*, \square)$                                     | } |
| $F$                     | {            | $(*, *)$ , | $(\square, *)$                                     | } |
| $F^\omega$              | {            | $(*, *)$ , | $(\square, *)$ $(\square, \square)$                | } |
| $CC$                    | {            | $(*, *)$ , | $(*, \square)$ $(\square, *)$ $(\square, \square)$ | } |



The Lambda Cube

$$\lambda_{\rightarrow} \longrightarrow F \longrightarrow F^\omega \longrightarrow CC$$

# Dependent Types in Coq

# Proof Assistants

Dependent type theories are at the foundation of proof assistants, like Coq, Agda, etc.

By *Curry-Howard Correspondence*

- ▶ proofs  $\longleftrightarrow$  programs
- ▶ propositions  $\longleftrightarrow$  types

# Proof Assistants

Dependent type theories are at the foundation of proof assistants, like Coq, Agda, etc.

By *Curry-Howard Correspondence*

- ▶ proofs  $\longleftrightarrow$  programs
- ▶ propositions  $\longleftrightarrow$  types

Two impactful projects based on Coq:

- ▶ CompCert: certified C compiler
- ▶ Mechanized proof of 4-color theorem

# Type Universes in Coq

The rule  $\Gamma \vdash \text{Type} : \text{Type}$  is unsound (Girard's paradox).

$$\Gamma \vdash \text{Prop} : \text{Type}_1$$

$$\Gamma \vdash \text{Set} : \text{Type}_1$$

$$\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}$$

$$\frac{\Gamma, x:A \vdash B : \text{Prop} \quad \Gamma \vdash A : s}{\Gamma \vdash (x : A) \rightarrow B : \text{Prop}}$$

$$\frac{\Gamma, x:A \vdash B : \text{Set} \quad \Gamma \vdash A : s \quad s \in \{\text{Prop}, \text{Set}\}}{\Gamma \vdash (x : A) \rightarrow B : \text{Set}}$$

$$\frac{\Gamma, x:A \vdash B : \text{Type}_i \quad \Gamma \vdash A : \text{Type}_i}{\Gamma \vdash (x : A) \rightarrow B : \text{Type}_i}$$

# Coq 101 - inductive definitions and recursion

```
1 Inductive nat : Type :=  
2   | 0  
3   | S (n : nat).
```

## Coq 101 - inductive definitions and recursion

```
1 Inductive nat : Type :=  
2   | 0  
3   | S (n : nat).
```

```
1 Fixpoint double (n : nat) : nat :=  
2   match n with  
3     | 0 => 0  
4     | S n' => S (S (double n'))  
5   end.
```

Recursion has to be structural.

## Coq 101 - inductive definitions and recursion

```
1 Inductive nat : Type :=  
2   | 0  
3   | S (n : nat).
```

```
1 Fixpoint double (n : nat) : nat :=  
2   match n with  
3     | 0 => 0  
4     | S n' => S (S (double n'))  
5   end.
```

Recursion has to be structural.

```
1 Inductive even : nat -> Prop :=  
2   | even0 : even 0  
3   | evenS : forall x:nat, even x -> even (S (S x)).
```



## Coq 101 - proofs

```
1 Definition even_prop := forall x:nat, even (double x).  
2  
3 Fixpoint even_proof(x: nat): even (double x) :=  
4   match x with  
5     | 0      => even0  
6     | S n'   => evenS (double n') (even_proof n')  
7   end.  
8  
9 Check even_proof : even_prop.
```

## Coq 101 - proofs

```
1 Definition even_prop := forall x:nat, even (double x).  
2  
3 Fixpoint even_proof(x: nat): even (double x) :=  
4   match x with  
5     | 0      => even0  
6     | S n'   => evenS (double n') (even_proof n')  
7   end.  
8  
9 Check even_proof : even_prop.
```

The 2nd branch has the type  $\text{even } S (S (\text{double } n'))$ , and Coq knows by normalizing the types:

$$\text{even } S (S (\text{double } n')) \equiv_{\beta} \text{even } (\text{double } (S n'))$$

## Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

| Proposition       | Term & Type                 |
|-------------------|-----------------------------|
| $A \wedge B$      | $t : (A, B)$                |
| $A \vee B$        | $t : A + B$                 |
| $A \rightarrow B$ | $t : A \rightarrow B$       |
| $\perp$           | $t : False$                 |
| $\neg A$          | $t : A \rightarrow False$   |
| $\forall x:A. B$  | $t : (x : A) \rightarrow B$ |
| $\exists x:A. B$  | $t : (x:A, B)$              |

# Curry-Howard correspondence in Coq

```
1 Inductive and (A B:Prop) : Prop :=  
2   conj : A -> B -> A /\ B  
3 where "A /\ B" := (and A B) : type_scope.
```

# Curry-Howard correspondence in Coq

```
1 Inductive and (A B:Prop) : Prop :=  
2   conj : A -> B -> A /\ B  
3 where "A /\ B" := (and A B) : type_scope.
```

```
1 Inductive or (A B:Prop) : Prop :=  
2   | or_introl : A -> A \/ B  
3   | or_intror : B -> A \/ B  
4 where "A \/ B" := (or A B) : type_scope.
```

# Curry-Howard correspondence in Coq

```
1 Inductive and (A B:Prop) : Prop :=  
2   conj : A -> B -> A /\ B  
3 where "A /\ B" := (and A B) : type_scope.
```

```
1 Inductive or (A B:Prop) : Prop :=  
2   | or_introl : A -> A \/ B  
3   | or_intror : B -> A \/ B  
4 where "A \/ B" := (or A B) : type_scope.
```

```
1 Inductive False : Prop :=.
```

# Curry-Howard correspondence in Coq

```
1 Inductive and (A B:Prop) : Prop :=  
2   conj : A -> B -> A /\ B  
3 where "A /\ B" := (and A B) : type_scope.
```

```
1 Inductive or (A B:Prop) : Prop :=  
2   | or_introl : A -> A \/ B  
3   | or_intror : B -> A \/ B  
4 where "A \/ B" := (or A B) : type_scope.
```

```
1 Inductive False : Prop :=.
```

```
1 Definition not (A:Prop) := A -> False.  
2 Notation "~ x" := (not x) : type_scope.
```

## Curry-Howard correspondence in Coq - continued

```
1 Notation "A -> B" := (forall (_ : A), B) : type_scope.  
2 Definition iff (A B:Prop) := (A -> B) /\ (B -> A).  
3 Notation "A <-> B" := (iff A B) : type_scope.
```



## Curry-Howard correspondence in Coq - continued

```
1 Notation "A -> B" := (forall (_ : A), B) : type_scope.  
2 Definition iff (A B:Prop) := (A -> B) /\ (B -> A).  
3 Notation "A <-> B" := (iff A B) : type_scope.
```

```
1 Inductive ex (A:Type) (P:A -> Prop) : Prop :=  
2   ex_intro : forall x:A, P x -> ex (A:=A) P.  
3  
4 Notation "'exists' x .. y , p" :=  
5   (ex (fun x => .. (ex (fun y => p)) ..)) : type_scope.
```

## Curry-Howard correspondence in Coq - continued

```
1 Notation "A -> B" := (forall (_ : A), B) : type_scope.  
2 Definition iff (A B:Prop) := (A -> B) /\ (B -> A).  
3 Notation "A <-> B" := (iff A B) : type_scope.
```

```
1 Inductive ex (A:Type) (P:A -> Prop) : Prop :=  
2   ex_intro : forall x:A, P x -> ex (A:=A) P.  
3  
4 Notation "'exists' x .. y , p" :=  
5   (ex (fun x => .. (ex (fun y => p)) ..)) : type_scope.
```

```
1 Inductive eq (A:Type) (x:A) : A -> Prop :=  
2   eq_refl : x = x :>A  
3  
4 Notation "x = y" := (eq x y) : type_scope.
```

# The equivalence between LEM and DNE

In **intuitionistic logics**, the *law of excluded middle* (LEM) and the *law of double negation* (DNE) are not provable.

▶ LEM:  $\forall P. P \vee \neg P$

▶ DNE:  $\forall P. \neg \neg P \rightarrow P$

By curry-howard correspondence, there are no terms that inhabit the types above.

# The equivalence between LEM and DNE

In **intuitionistic logics**, the *law of excluded middle* (LEM) and the *law of double negation* (DNE) are not provable.

▶ LEM:  $\forall P. P \vee \neg P$

▶ DNE:  $\forall P. \neg\neg P \rightarrow P$

By curry-howard correspondence, there are no terms that inhabit the types above.

However,  $\forall P. P \rightarrow \neg\neg P$  can be proved.

# The equivalence between LEM and DNE

In **intuitionistic logics**, the *law of excluded middle* (LEM) and the *law of double negation* (DNE) are not provable.

▶ LEM:  $\forall P. P \vee \neg P$

▶ DNE:  $\forall P. \neg\neg P \rightarrow P$

By curry-howard correspondence, there are no terms that inhabit the types above.

However,  $\forall P. P \rightarrow \neg\neg P$  can be proved. **How?**

# The equivalence between LEM and DNE

In **intuitionistic logics**, the *law of excluded middle* (LEM) and the *law of double negation* (DNE) are not provable.

► LEM:  $\forall P. P \vee \neg P$

► DNE:  $\forall P. \neg\neg P \rightarrow P$

By curry-howard correspondence, there are no terms that inhabit the types above.

However,  $\forall P. P \rightarrow \neg\neg P$  can be proved. **How?**

We will prove that LEM is equivalent to DNE:

- 1 **Definition** LEM: Prop := forall P: Prop, P  $\vee$   $\neg$ P.
- 2 **Definition** DNE: Prop := forall P: Prop,  $\neg\neg$ P  $\rightarrow$  P.
- 3 **Definition** LEM\_DNE\_EQ: Prop := LEM  $\leftrightarrow$  DNE.

# LEM $\rightarrow$ DNE

```
1 Definition LEM_To_DNE :=
2   fun (lem: forall P : Prop, P \ / ~ P) (Q:Prop) (q: ~~Q)
3     =>
4     match lem Q with
5     | or_introl l =>
6       l
7     | or_intror r =>
8       match (q r) with end
9     end.
10
11 Check LEM_To_DNE : LEM -> DNE.
```

## DNE $\rightarrow$ LEM

```
1 Definition DNE_To_LEM :=
2   fun (dne: forall P : Prop, ~~P -> P) (Q:Prop) =>
3     (dne (Q \ / ~ Q))
4     (fun H: ~(Q \ / ~Q) =>
5       let nq := (fun q: Q => H (or_introl q))
6       in H (or_intror nq)
7     ).
8
9 Check DNE_To_LEM : DNE -> LEM.
10
11 Definition proof := conj LEM_To_DNE DNE_To_LEM.
12 Check proof : LEM <-> DNE.
```



# Dependent Types in Programming Languages

Despite the huge success in proof assistants, its adoption in programming languages is limited.

- ▶ Scala supports *path-dependent types* and *literal types*.
- ▶ Dependent Haskell is proposed by researchers.

# Dependent Types in Programming Languages

Despite the huge success in proof assistants, its adoption in programming languages is limited.

- ▶ Scala supports *path-dependent types* and *literal types*.
- ▶ Dependent Haskell is proposed by researchers.

**Challenge:** the decidability of type checking.

# Problem with Type Checking

Value constructors:

$NVec : \mathbb{N} \rightarrow *$

$nil : NVec\ 0$

$cons : \mathbb{N} \rightarrow (n:\mathbb{N}) \rightarrow NVec\ n \rightarrow NVec\ n + 1$

Appending vectors:

$append : (m:\mathbb{N}) \rightarrow (n:\mathbb{N}) \rightarrow NVec\ m \rightarrow NVec\ n \rightarrow NVec\ (n + m)$

$append = \lambda m:\mathbb{N}. \lambda n:\mathbb{N}. \lambda l:NVec\ m. \lambda t:NVec\ n.$

$\quad\quad\quad match\ l\ with$

$\quad\quad\quad | nil \Rightarrow t$

$\quad\quad\quad | cons\ x\ r\ y \Rightarrow cons\ x\ (r + n)\ (append\ r\ n\ y\ t)$

**Question:** How does the type checker know  $S\ (r + n) = n + (S\ r)$ ?