

Foundations of Software Fall 2023

Week 11

1

Different Kinds of Maps

What is missing?

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

2

Different Kinds of Maps

What is missing?

$$\begin{array}{lcl} \textit{Term} & \rightarrow & \textit{Term} \ (\lambda x.t) \\ \textit{Type} & \rightarrow & \textit{Term} \ (\lambda X.t) \\ \textit{Type} & \rightarrow & \textit{Type} \ ??? \\ \textit{Term} & \rightarrow & \textit{Type} \ ??? \end{array}$$

Agenda today:

- ▶ Type operators
- ▶ Dependent types

2

Type Operators and System F_ω

3

Type Operators

Example. Type operators in Scala:

```
def termIdentity(x: Int): Int = x // similar to  
val termIdentity: Int => Int = (x: Int) => x
```

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

4

Type Operators

Example. Type operators in Scala:

```
def termIdentity(x: Int): Int = x // similar to  
val termIdentity: Int => Int = (x: Int) => x
```

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

Type operators are functions at the type-level.

$\lambda X :: K. T$

4

Type Operators

Example. Type operators in Scala:

```
def termIdentity(x: Int): Int = x // similar to  
val termIdentity: Int => Int = (x: Int) => x
```

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

Type operators are functions at the type-level.

$\lambda X :: K. T$

Two Problems:

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

4

Kinding

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

5

Kinding

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

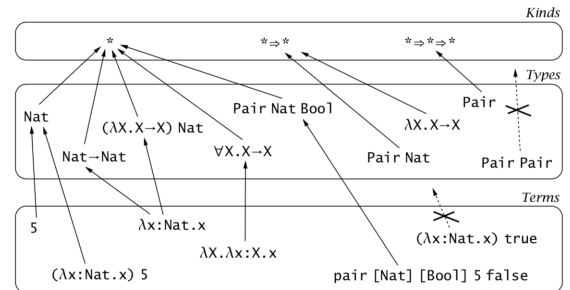
- `*` proper types, e.g. `Bool`, `Int → Int`
- `* ⇒ *` type operators: map proper types to proper types
- `* ⇒ * ⇒ *` two-argument operators
- `(* ⇒ *) ⇒ *` type operators: map type operators to proper types

5

Kinding

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

- `*` proper types, e.g. `Bool`, `Int → Int`
- `* ⇒ *` type operators: map proper types to proper types
- `* ⇒ * ⇒ *` two-argument operators
- `(* ⇒ *) ⇒ *` type operators: map type operators to proper types



5

Equivalence of Types

Problem: all the types below are equivalent

`Nat → Bool` `Nat → Id Bool` `Id Nat → Id Bool`
`Id Nat → Bool` `Id (Nat → Bool)` `Id (Id (Id Nat → Bool))`

We need to introduce a *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:

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

6

First-class Type Operators

Scala supports passing type operators as argument:

```
def makeInt[F[_]](f: () => F[Int]): F[Int] = f()
// equiv to
def makeInt[F <: [X] =>> Any](...): ...
```

```
makeInt[List]() => List[Int](3)
makeInt[Option]() => None
makeInt[[T] =>> (T, T)]() => (3, 4)
```

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

7

System F_ω — Syntax

Formalizing first-class type operators leads to System F_ω :

| | |
|---------------------|-----------------------------|
| $t ::= \dots$ | <i>terms</i> |
| $\lambda X :: K. t$ | <i>type abstraction</i> |
| $T ::=$ | <i>types</i> |
| X | <i>type variable</i> |
| $T \rightarrow T$ | <i>type of functions</i> |
| $\forall X :: K. T$ | <i>universal type</i> |
| $\lambda X :: K. T$ | <i>operator abstraction</i> |
| $T \ T$ | <i>operator application</i> |
| $K ::=$ | <i>kinds</i> |
| $*$ | <i>kind of proper types</i> |
| $K \Rightarrow K$ | <i>kind of operators</i> |

8

System F_ω — Semantics

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

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

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

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

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

9

System F_ω — Kinding

| | |
|--|-----------|
| $\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K}$ | (K-TVAR) |
| $\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1. T_2 :: K_1 \Rightarrow K_2}$ | (K-ABS) |
| $\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 \ T_2 :: K_2}$ | (K-APP) |
| $\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$ | (K-ARROW) |
| $\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1. T_2 :: *}$ | (K-ALL) |

10

System F_ω — Type Equivalence

| | | |
|--|---------------------------------|--|
| $T \equiv T$ | $\frac{T \equiv S}{S \equiv T}$ | $\frac{S \equiv U \quad U \equiv T}{S \equiv T}$ |
| $\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2}$ | (Q-ARROW) | |
| $\frac{S_2 \equiv T_2}{\forall X :: K_1. S_2 \equiv \forall X :: K_1. T_2}$ | (K-ALL) | |
| $\frac{S_2 \equiv T_2}{\lambda X :: K_1. S_2 \equiv \lambda X :: K_1. T_2}$ | (Q-ABS) | |
| $\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \ S_2 \equiv T_1 \ T_2}$ | (Q-APP) | |
| $(\lambda X :: K. T_1) \ T_2 \equiv [X \mapsto T_2] T_1$ | (Q-APPABS) | |

11

System F_ω — Typing

$$\begin{array}{c}
 \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR}) \\
 \\
 \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS}) \\
 \\
 \frac{\Gamma \vdash t_1 : S \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : T} \quad (\text{T-APP}) \\
 \\
 \frac{\Gamma, X :: K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X :: K_1. t_2 : \forall X :: K_1. T_2} \quad (\text{T-TABS}) \\
 \\
 \frac{\Gamma \vdash t : \forall X :: K. T_2 \quad \Gamma \vdash T :: K}{\Gamma \vdash t [T] : [X \mapsto T] T_2} \quad (\text{T-TAPP}) \\
 \\
 \frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad (\text{T-EQ})
 \end{array}$$

12

Kinding question

Clicker question: What are the kinds of $\lambda X :: *. X \rightarrow X$ and $\forall X :: *. X \rightarrow X$, respectively?

- A. $*$ and $*$
- B. $*$ and $* \Rightarrow *$
- C. $* \Rightarrow *$ and $*$
- D. $* \Rightarrow *$ and $* \Rightarrow *$

URL: ttopoll.eu
Session ID: cs452

13

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.

14

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 $t \longrightarrow t'$.

15

Dependent Types

16

Why Does It Matter?

Example 1. Track length of vectors in types:

```
NVec  :: Nat → *  
first : (n:Nat) → NVec (n + 1) → Nat
```

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

17

Why Does It Matter?

Example 1. Track length of vectors in types:

```
NVec  :: Nat → *  
first : (n:Nat) → NVec (n + 1) → 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      : (f:Format) → Data(f) → String
```

```
Data([])      = Unit  
Data("%" :: 'd' :: cs) = Nat * Data(cs)  
Data("%" :: 's' :: cs) = String * Data(cs)  
Data(c :: cs)    = Data(cs)
```

17

Dependent Function Type (a.k.a. Π Types)

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

```
 $\lambda x:S.t$  : (x:S) → T
```

18

Dependent Function Type (a.k.a. Π 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:

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

18

The Calculus of Constructions

19

The Calculus of Constructions: Syntax

| | | |
|-----------------------|--|------------------------------|
| $t ::=$ | | <i>terms</i> |
| s | | <i>sort</i> |
| x | | <i>variable</i> |
| $\lambda x:t.t$ | | <i>abstraction</i> |
| $t \ t$ | | <i>application</i> |
| $(x:t) \rightarrow t$ | | <i>dependent type</i> |
| $s ::=$ | | <i>sorts</i> |
| $*$ | | <i>sort of proper types</i> |
| \square | | <i>sort of kinds</i> |
| $\Gamma ::=$ | | <i>contexts</i> |
| \emptyset | | <i>empty context</i> |
| $\Gamma, x:T$ | | <i>term variable binding</i> |

The semantics is the usual β -reduction.

20

The Calculus of Constructions: Typing

| | |
|---|--|
| $\vdash * : \square$ (T-AXIOM) | $\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$ (T-VAR) |
| $\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : (x:S) \rightarrow T}$ | (T-ABS) |
| $\frac{\Gamma \vdash t_1 : (x:S) \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 \ t_2 : [x \mapsto t_2] T}$ | (T-APP) |
| $\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash T : s_2}{\Gamma \vdash (x:S) \rightarrow T : s_2}$ | (T-PI) |
| $\frac{\Gamma \vdash t : T \quad T \equiv T' \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'}$ | (T-CONV) |

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

21

Four Kinds of Lambdas

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

22

Four Kinds of Lambdas

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

22

Four Kinds of Lambdas

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

22

Four Kinds of Lambdas

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

22

Strong Normalization

Given the following β -reduction rules

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

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

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

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

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

23

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) | | | |
|-------------------------|--------------|---|---|--|
| λ_{\rightarrow} | { | $(*, *)$ | } | |
| λ_P | { | $(*, *)$, $(*, \square)$ | } | |
| F | { | $(*, *)$, $(\square, *)$ | } | |
| F^ω | { | $(*, *)$, $(\square, *)$, (\square, \square) | } | |
| CC | { | $(*, *)$, $(*, \square)$, $(\square, *)$, (\square, \square) | } | |

The Lambda Cube
 $\lambda_{\rightarrow} \rightarrow F \rightarrow F^\omega \rightarrow CC$

24

Dependent Types in Coq

25

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

26

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

26

Type Universes in Coq

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

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

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

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

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

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

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

27

Coq 101 - inductive definitions and recursion

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

28

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**.

28

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)).

```

28

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.

```

29

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:

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

29

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 | $t : \text{False}$ |
| $\neg A$ | $t : A \rightarrow \text{False}$ |
| $\forall x:A. B$ | $t : (x : A) \rightarrow B$ |
| $\exists x:A. B$ | $t : (x:A, B)$ |

30

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.
```

31

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.
```

31

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 :=.
```

31

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.
```

31

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.

```

32

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.

```

32

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.

```

32

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.

33

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.

33

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?**

33

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 ∨ ~P.
2 Definition DNE: Prop := forall P: Prop, ~~P -> P.
3 Definition LEM_DNE_EQ: Prop := LEM <-> DNE.
```

33

LEM → 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.
```

34

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.
```

35

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.

36

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.

36

Problem with Type Checking

Value constructors:

```
NVec : ℕ → *
nil   : NVec 0
cons  : ℕ → (ℕ → NVec n → NVec n + 1
```

Appending vectors:

```
append : (m:ℕ) → (n:ℕ) → NVec m → NVec n → NVec (n + m)
append = λm:ℕ. λn:ℕ. λl:NVec m. λt:NVec n.
  match l with
  | nil => t
  | cons x r y => cons x (r + n) (append r n y t)
```

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

37