Theory of Types and Programming Languages Spring 2022

Week 11

Plan

TODAY:

1. type operators
2. dependent types

Note: This week's material is *not* from the TAPL textbook;

– it is mostly from Chapter 2 of "*Advanced Topics in Types and Programming Languages*" (Benjamin C. Pierce et. al, MIT Press)

Different Kinds of Maps

What is missing?

$$\begin{array}{lcll} \mathit{Term} & \to & \mathit{Term} & (\lambda x.t) \\ \mathit{Type} & \to & \mathit{Term} & (\Lambda X.t) \end{array}$$

Different Kinds of Maps

What is missing?

$$\begin{array}{lcll} \mathit{Term} & \rightarrow & \mathit{Term} & (\lambda x.t) \\ \mathit{Type} & \rightarrow & \mathit{Term} & (\Lambda X.t) \\ \mathit{Type} & \rightarrow & \mathit{Type} & ??? \\ \mathit{Term} & \rightarrow & \mathit{Type} & ??? \end{array}$$

Agenda today:

- Type operators
- Dependent types

Type Operators and System F_ω

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 the type level.

$$\lambda X.\ T$$

Type Operators

Example. Type operators in Scala:

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

Type operators are functions at the type level.

$$\lambda X.\ T$$

Three Problems:

- Type checking of type operators
- Equivalence of types
- *Abstracting* over type operators

Kinding

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

Kinding

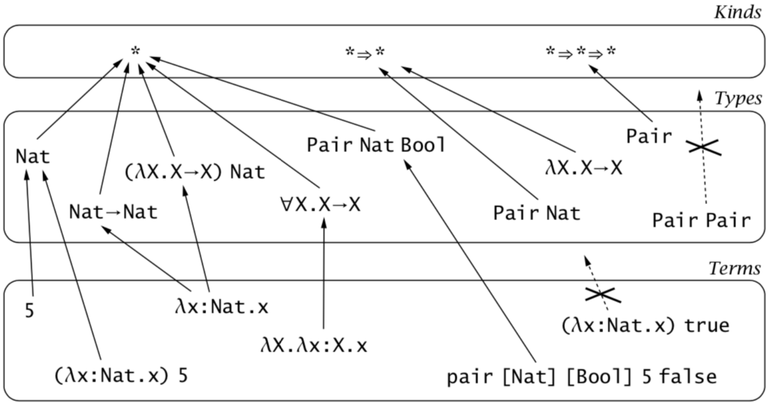
Problem: avoid meaningless types, like $\mathit{MkFun}[\mathit{Int}, \mathit{String}]$.

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



Kinding Notation

By analogy with lambda parameter type annotation, we write:

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

where K is the *kind* of X in this abstraction

Equivalence of Types

Problem: all the types below are equivalent

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

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

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

And we need one typing rule:

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

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_ω — Syntax

Formalizing first-class type operators leads to *System* F_ω:

$\mathtt{t}$	$::=$	...	*terms*
		$\lambda X :: K.t$	*type abstraction*
$\mathtt{T}$	$::=$		*types*
		$\mathtt{X}$	*type variable*
		$T \to T$	*type of functions*
		$\forall X :: K.T$	*universal type*
		$\lambda X :: K.T$	*operator abstraction*
		$T\ T$	*operator application*
$\mathtt{K}$	$::=$		*kinds*
		$*$	*kind of proper types*
		$K \Rightarrow K$	*kind of operators*

System F_ω — Semantics

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \quad \text{(E-App2)}$$

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

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

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

System F_ω — Kinding

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \qquad \text{(K-TVar)}$$

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

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

$$\frac{\Gamma \vdash T_1 : * \qquad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \to T_2 :: *} \qquad \text{(K-Arrow)}$$

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

System F_ω — Type Equivalence

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

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

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

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

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

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

System F_ω — Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : S \to T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T} \qquad \text{(T-App)}$$

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

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

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

Example

$$
\begin{array}{l}
\mathit{type}\ \mathit{PairRep}[\mathit{Pair} :: * \Rightarrow * \Rightarrow *] = \{ \\
\quad \mathit{pair} : \forall X.\forall Y.X \to Y \to (\mathit{Pair}\ X\ Y), \\
\quad \mathit{fst} : \forall X.\forall Y.(\mathit{Pair}\ X\ Y) \to X, \\
\quad \mathit{snd} : \forall X.\forall Y.(\mathit{Pair}\ X\ Y) \to Y \\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{def}\ \mathit{swap}[\mathit{Pair} :: * \Rightarrow * \Rightarrow *, X :: *, Y :: *] \\
\quad (\mathit{rep} : \mathit{PairRep}\ \mathit{Pair}) \\
\quad (\mathit{pair} : \mathit{Pair}\ X\ Y) : \mathit{Pair}\ Y\ X \\
= \\
\quad \mathit{let}\ x = \mathit{rep.fst}\ [X]\ [Y]\ \mathit{pair}\ \mathit{in} \\
\quad \mathit{let}\ y = \mathit{rep.snd}\ [X]\ [Y]\ \mathit{pair}\ \mathit{in} \\
\quad \mathit{rep.pair}\ [Y]\ [X]\ y\ x
\end{array}
$$

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 integer vectors in types:

$$\begin{array}{lcl} Vec & :: & Nat \to * \\ first & : & (n{:}Nat) \to Vec\ (n+1) \to Int \end{array}$$

$(x{:}S) \to T$ 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 integer vectors in types:

$$\begin{array}{lcl} Vec & :: & Nat \to * \\ first & : & (n{:}Nat) \to Vec\ (n+1) \to Int \end{array}$$

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

Example 2. Safe formatting for $sprintf$:

$$\begin{array}{lcl} sprintf & : & (f{:}Format) \to Data(f) \to String \\ \\ Data([]) & = & Unit \\ Data('\%' :: 'd' :: cs) & = & Nat * Data(cs) \\ Data('\%' :: 's' :: cs) & = & String * Data(cs) \\ Data(c :: cs) & = & Data(cs) \end{array}$$

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

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

$$\lambda x{:}S.\ t \quad : \quad (x{:}S) \to T$$

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

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

$$\lambda x{:}S.\ t \quad : \quad (x{:}S) \to T$$

'$(x{:}S) \to T$' is also written '$\Pi_{x:S}\ T$' in the literature.

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

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

$$\lambda x{:}S.\ t \quad : \quad (x{:}S) \to T$$

'$(x{:}S) \to T$' is also written '$\Pi_{x:S}\ T$' in the literature.

When T does not depend on x, degenerates to function type

Notation:

$$S \to T \ \triangleq\ (x{:}S) \to T \qquad \textit{where } x \textit{ does not appear free in } T$$

The Calculus of Constructions

The Calculus of Constructions: Syntax

$\mathtt{t},\mathtt{T} ::=$	*terms*
$\quad \mathtt{s}$	*sort*
$\quad \mathtt{x}$	*variable*
$\quad \lambda\mathtt{x}:\mathtt{t}.\mathtt{t}$	*abstraction*
$\quad \mathtt{t}\ \mathtt{t}$	*application*
$\quad (x{:}t) \to t$	*dependent type*
$\mathtt{s} ::=$	*sorts*
$\quad *$	*sort of proper types*
$\quad \square$	*sort of kinds*
$\Gamma ::=$	*contexts*
$\quad \emptyset$	*empty context*
$\quad \Gamma, x{:}T$	*term variable binding*

The semantics is the usual β-reduction.

The Calculus of Constructions: Typing

$$\vdash * : \Box \quad (\text{T-Axiom}) \qquad\qquad \frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-Var})$$

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

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

$$\frac{\Gamma \vdash S : s_1 \qquad \Gamma, x{:}S \vdash T : s_2}{\Gamma \vdash (x{:}S) \to T : s_2} \quad (\text{T-Pi})$$

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

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

Four Kinds of Lambdas

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

Four Kinds of Lambdas

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

Four Kinds of Lambdas

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

Four Kinds of Lambdas

Example	Type
$\lambda x{:}\mathbb{N}.\ x + 1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.\ f\ x$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X{:}*.\ \lambda x{:}X.\, x$	$(X{:}*) \to X \to X$
$\lambda F{:}* \to *.\ \lambda x{:}F\ \mathbb{N}.\, x$	$(F{:}* \to *) \to (F\ \mathbb{N}) \to (F\ \mathbb{N})$
$\lambda X{:}*.\ X$	$* \to *$
$\lambda F{:}* \to *.\ F\ \mathbb{N}$	$(* \to *) \to *$
$\lambda n{:}\mathbb{N}.\ \mathit{Vec}\ n$	$\mathbb{N} \to *$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.\ \mathit{Vec}\ (f\ 6)$	$(\mathbb{N} \to \mathbb{N}) \to *$

Strong Normalization

Given the following β-reduction rules

$$\frac{t_1 \longrightarrow t_1'}{\lambda x{:}T_1.t_1 \longrightarrow \lambda x{:}T_1.\ t_1'} \qquad (\beta\text{-}\textsc{Abs})$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad (\beta\text{-}\textsc{App1})$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad (\beta\text{-}\textsc{App2})$$

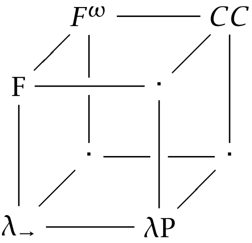
$$(\lambda x{:}T_1.t_1)t_2 \longrightarrow [x \mapsto t_2]t_1 \qquad (\beta\text{-}\textsc{AppAbs})$$

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

$$\frac{\Gamma \vdash S : s_i \qquad \Gamma, x{:}S \vdash T : s_j}{\Gamma \vdash (x{:}S) \to T : s_j} \quad \text{(T-Pi)}$$

System		(s_i, s_j)				
$\lambda_{\to}$	$\{$	$(*, *)$				$\}$
λP	$\{$	$(*, *)$,	$(*, \Box)$			$\}$
F	$\{$	$(*, *)$,		$(\Box, *)$		$\}$
F^{ω}	$\{$	$(*, *)$,		$(\Box, *)$,	$(\Box, \Box)$	$\}$
CC	$\{$	$(*, *)$,	$(*, \Box)$,	$(\Box, *)$,	$(\Box, \Box)$	$\}$



The Lambda Cube

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

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

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

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

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

$$\frac{\Gamma, x{:}A \vdash B : \mathit{Prop} \qquad \Gamma \vdash A : s}{\Gamma \vdash (x : A) \to B : \mathit{Prop}}$$

$$\frac{\Gamma, x{:}A \vdash B : \mathit{Set} \qquad \Gamma \vdash A : s \qquad s \in \{\mathit{Prop}, \mathit{Set}\}}{\Gamma \vdash (x : A) \to B : \mathit{Set}}$$

$$\frac{\Gamma, x{:}A \vdash B : \mathit{Type}_i \qquad \Gamma \vdash A : \mathit{Type}_i}{\Gamma \vdash (x : A) \to B : \mathit{Type}_i}$$

Coq 101 - inductive definitions and recursion

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Coq 101 - inductive definitions and recursion

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Fixpoint double (n : nat) : nat :=
  match n with
    | O => O
    | S n' => S (S (double n'))
  end.
```

Recursion has to be structural.

Coq 101 - inductive definitions and recursion

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

```
Fixpoint double (n : nat) : nat :=
  match n with
    | O => O
    | S n' => S (S (double n'))
  end.
```

Recursion has to be structural.

```
Inductive even : nat -> Prop :=
  | even0 : even O
  | evenS : forall x:nat, even x -> even (S (S x)).
```

Coq 101 - proofs

```
Definition even_prop := forall x:nat, even (double x).

Fixpoint even_proof(x: nat): even (double x) :=
  match x with
  | O      => even0
  | S n'   => evenS (double n') (even_proof n')
  end.

Check even_proof : even_prop.
```

Coq 101 - proofs

```
Definition even_prop := forall x:nat, even (double x).

Fixpoint even_proof(x: nat): even (double x) :=
  match x with
  | O      => even0
  | S n'   => evenS (double n') (even_proof n')
  end.

Check even_proof : even_prop.
```

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

$$even\, S\, (S\, (double\; n')) \equiv_\beta even\, (double\, (S\; n'))$$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

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

Curry-Howard correspondence in Coq

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

Curry-Howard correspondence in Coq

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

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

Curry-Howard correspondence in Coq

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

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

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

Curry-Howard correspondence in Coq

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

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

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

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

Curry-Howard correspondence in Coq - continued

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

Curry-Howard correspondence in Coq - continued

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

```
Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.

Notation "’exists’ x .. y , p" :=
  (ex (fun x => .. (ex (fun y => p)) ..)) : type_scope.
```

Curry-Howard correspondence in Coq - continued

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

```
Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.

Notation "'exists' x .. y , p" :=
  (ex (fun x => .. (ex (fun y => p)) ..)) : type_scope.
```

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
  eq_refl : x = x :>A

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: $\forall P. \neg\neg P \to P$

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: $\forall P. \neg\neg P \to P$

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

However, $\forall P.\, P \to \neg\neg P$ 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: $\forall P. P \lor \neg P$
- DNE: $\forall P. \neg\neg P \to P$

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

However, $\forall P.\, P \to \neg\neg P$ 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: $\forall P. \neg\neg P \to P$

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

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

We will prove that LEM is equivalent to DNE:

```
Definition LEM: Prop := forall P: Prop, P \/~P.
Definition DNE: Prop := forall P: Prop, ~~P -> P.
Definition LEM_DNE_EQ: Prop := LEM <-> DNE.
```

LEM → DNE

```
Definition LEM_To_DNE :=
  fun (lem: forall P : Prop, P \/ ~ P) (Q:Prop) (q: ~~Q) =>
    match lem Q with
    | or_introl l =>
      l

    | or_intror r =>
      match (q r) with end
    end.

Check LEM_To_DNE : LEM -> DNE.
```

DNE $\rightarrow$ LEM

```
Definition DNE_To_LEM :=
  fun (dne: forall P : Prop, ~~P -> P) (Q:Prop) =>
    (dne (Q \/ ~ Q))
      (fun H: ~(Q \/ ~Q) =>
        let nq := (fun q: Q => H (or_introl q))
        in H (or_intror nq)
      ).

Check DNE_To_LEM :  DNE -> LEM.

Definition proof := conj LEM_To_DNE DNE_To_LEM.
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:

$$\begin{array}{lcl} Vec & : & \mathbb{N} \rightarrow * \\ nil & : & Vec\ 0 \\ cons & : & \mathbb{N} \rightarrow (n{:}\mathbb{N}) \rightarrow Vec\ n \rightarrow Vec\ n+1 \end{array}$$

Appending vectors:

$$\begin{array}{lcl} append & : & (m{:}\mathbb{N}) \rightarrow (n{:}\mathbb{N}) \rightarrow Vec\ m \rightarrow Vec\ n \rightarrow Vec\ (n+m) \\ append & = & \lambda m{:}\mathbb{N}.\, \lambda n{:}\mathbb{N}.\, \lambda l{:}Vec\ m.\, \lambda t{:}Vec\ n. \\ & & \quad match\ l\ with \\ & & \quad \mid nil \Rightarrow t \\ & & \quad \mid cons\ x\ r\ y \Rightarrow cons\ x\ (r+n)\ (append\ r\ n\ y\ t) \end{array}$$

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