Foundations of Software
Fall 2020

Week 9

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?

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

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 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

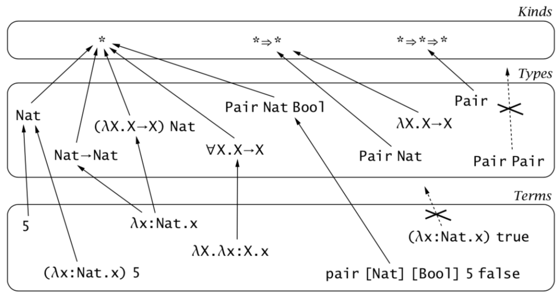
Problem: avoid meaningless types, like $MkFun[Int, String]$.

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

Kinding

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



Equivalence of Types

Problem: all the types below are equivalent

$Nat \to Bool$	$Nat \to Id\ Bool$	$Id\ Nat \to Id\ Bool$
$Id\ Nat \to Bool$	$Id\ (Nat \to Bool)$	$Id(Id(Id\ Nat \to Bool)$

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

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

t ::=	...	*terms*
	$\lambda X :: K.t$	*type abstraction*
T ::=		*types*
	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*
K ::=		*kinds*
	$*$	*kind of proper types*
	$K \Rightarrow K$	*kind of operators*

Dependent Types

Why Does It Matter?

Example 1. Track length of vectors in types:

$$\begin{array}{lll} Vector & :: & Nat \to * \\ first & : & (n{:}Nat) \to Vector\ (n+1) \to D \end{array}$$

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

$$\begin{array}{lll} Vector & :: & Nat \to * \\ first & : & (n{:}Nat) \to Vector\ (n+1) \to D \end{array}$$

$(x{:}S) \to T$ 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*:

$$sprintf \quad : \quad (f{:}Format) \to Data(f) \to String$$

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

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*:

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

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

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

The Calculus of Constructions

The Calculus of Constructions: Syntax

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

The semantics is the usual β-reduction.

The Calculus of Constructions: Typing

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

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

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

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

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

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

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad (\beta\text{-APP1})$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad (\beta\text{-APP2})$$

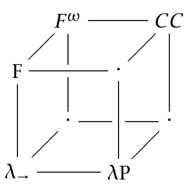
$$(\lambda x{:}T_1.t_1)t_2 \longrightarrow [x \mapsto t_2]t_1 \qquad (\beta\text{-APPABS})$$

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

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

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



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

Coq is based on *Calculus of Inductive Construction*, which is an extension of CC with inductive definition.

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

Coq is based on *Calculus of Inductive Construction*, which is an extension of CC with inductive definition.

Two impactful projects based on Coq:

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

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
$A \wedge B$	$t : (A, B)$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	$t : (A, B)$
$A \vee B$	$t : A + B$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	$t : (A, B)$
$A \vee B$	$t : A + B$
$A \rightarrow B$	$t : A \rightarrow B$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	$t : (A, B)$
$A \vee B$	$t : A + B$
$A \rightarrow B$	$t : A \rightarrow B$
$\neg A$	$t : A \rightarrow False$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	$t : (A, B)$
$A \vee B$	$t : A + B$
$A \rightarrow B$	$t : A \rightarrow B$
$\neg A$	$t : A \rightarrow False$
$\bot$	$t : False$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	$t : (A, B)$
$A \vee B$	$t : A + B$
$A \to B$	$t : A \to B$
$\neg A$	$t : A \to False$
$\bot$	$t : False$
$\forall x{:}A.\, B$	$t : (x : A) \to B$

Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

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

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) do not hold.

- 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) do not hold.

- 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) do not hold.

- 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?

The equivalence between LEM and DNE

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

- 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 $\to$ 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 $\to$ 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: Vector Again

Value constructors:

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

Appending vectors:

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

Question: How does the type checker know $r+1+n = r+n+1$?