Foundations of Software
Fall 2020

Week 9

Different Kinds of Maps

What is missing?

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

Different Kinds of Maps

What is missing?

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

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

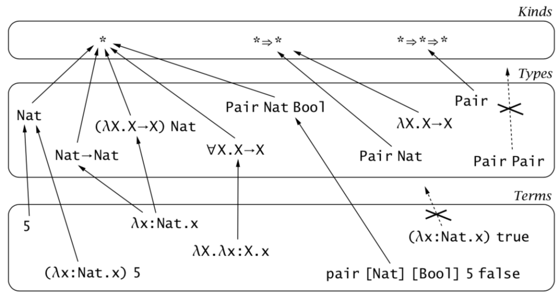
Problem: avoid meaningless types, like $MkFun[Int, String]$.

$*$	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:

$\mathtt{t}$ ::=	...	*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*

Dependent Types

Why Does It Matter?

Example 1. Track length of vectors in types:

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

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

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

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

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

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

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:

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

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:

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

By Curry-Howard correspondence, it corresponds to universal quantification:

$$(x{:}A) \to B(x) \qquad \longleftrightarrow \qquad \forall x{:}A.B(x)$$

First-Order Dependent Types

First-Order Dependent Types: λLF

System λLF generalizes STLC with dependent function types and *type families*.

`t` ::=		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
`T` ::=		*types*
	`X`	*type/family variable*
	$(x{:}T) \to T$	*dependent function type*
	`T t`	*type family application*

Type or family variables X can only be declared in the typing context Γ. E.g., we may assume $Vector :: Nat \to *$ as a type family variable.

System λLF: Kinds

Kinds can distinguish *proper types* from *type families*.

K ::=		*kinds*
	$*$	*kinds of proper types*
	$(x{:}T) \to K$	*kind of type families*
Γ ::=		*contexts*
	$\emptyset$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*
	$\Gamma, X{::}K$	*type variable binding*

Well-formed kinds $\Gamma \vdash K$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash T :: * \qquad \Gamma, x{:}T \vdash K}{\Gamma \vdash (x{:}T) \to K} \qquad \text{(WF-PI)}$$

System λLF: Kinding

Kinding ensures that types are well-formed $\Gamma \vdash T :: K$

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

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

$$\frac{\Gamma \vdash S :: (x{:}T) \to K \qquad \Gamma \vdash t : T}{\Gamma \vdash S\ t :: [x \mapsto t]K} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T :: K \qquad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'} \qquad \text{(K-CONV)}$$

System λLF: Typing

Typing ensures that terms are well-formed $\Gamma \vdash t :: T$

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

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

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

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

System λLF: Equivalence Rules

With types in kinds, and terms in types, equivalence becomes more complex than System F_ω.

$$Vector\ ((\lambda n{:}\mathbb{N}.n * n)\ 2) \quad \leftrightarrow \quad Vector\ 4$$

λLF defines on several equivalence relations:

- kind equivalence $\Gamma \vdash K \equiv K'$
- type equivalence $\Gamma \vdash T \equiv T' :: *$
- term equivalence $\Gamma \vdash t \equiv t' : T$

System λLF: Equivalence Rules

With types in kinds, and terms in types, equivalence becomes more complex than System F_ω.

$$Vector\ ((\lambda n{:}\mathbb{N}.n * n)\ 2) \quad \leftrightarrow \quad Vector\ 4$$

λLF defines on several equivalence relations:

- kind equivalence $\Gamma \vdash K \equiv K'$
- type equivalence $\Gamma \vdash T \equiv T' :: *$
- term equivalence $\Gamma \vdash t \equiv t' : T$

For *decidable* type checking, type systems usually embrace

- definitional equality, i.e. equality by definition (e.g. $x := 3$)
- computational equality, usually β-equality and η-equality.

System λLF: Kind Equivalence

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: * \qquad \Gamma, x{:}T_1 \vdash K_1 \equiv K_2}{\Gamma \vdash (x{:}T_1) \to K_1 \equiv (x{:}T_2) \to K_2} \quad (\text{QK-Pi})$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K} \quad (\text{QK-Refl})$$

$$\frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \quad (\text{QK-Sym})$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3} \quad (\text{QK-Trans})$$

System λLF: Type Equivalence

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \qquad \Gamma, x{:}T_1 \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash (x{:}S_1) \to S_2 \equiv (x{:}T_1) \to T_2 :: *} \quad (\text{QT-Pi})$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: (x{:}T) \to K \qquad \Gamma \vdash t_1 \equiv t_2 : T}{\Gamma \vdash S_1\ t_1 \equiv S_2\ t_2 :: [x \mapsto t_1]K} \quad (\text{QT-App})$$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K} \quad (\text{QT-Refl})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: K}{\Gamma \vdash T_2 \equiv T_1 :: K} \quad (\text{QT-Sym})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: K \qquad \Gamma \vdash T_2 \equiv T_3 :: K}{\Gamma \vdash T_1 \equiv T_3 :: K} \quad (\text{QT-Trans})$$

System λLF: Term Equivalence

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: * \qquad \Gamma, x{:}S_1 \vdash t_1 \equiv t_2 : T}{\Gamma \vdash \lambda x{:}S_1.t_1 \equiv \lambda x{:}S_2.t_2 : (x{:}S_1) \to T} \quad (\text{Q-Abs})$$

$$\frac{\Gamma \vdash t_1 \equiv s_1 : (x{:}S) \to T \qquad \Gamma \vdash t_2 \equiv s_2 : S}{\Gamma \vdash t_1\ t_2 \equiv s_1\ s_2 : [x \mapsto t_2]T} \quad (\text{Q-App})$$

$$\frac{\Gamma, x{:}S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x{:}S.t)\ s \equiv [x \mapsto s]t : [x \mapsto s]T} \quad (\text{Q-Beta})$$

$$\frac{\Gamma \vdash t : (x{:}S) \to T \qquad x \notin FV(t)}{\Gamma \vdash \lambda x{:}S.t\ x \equiv t : (x{:}S) \to T} \quad (\text{Q-Eta})$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t :: T} \ (\text{Q-Refl}) \qquad \frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \ (\text{Q-Sym})$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : T \qquad \Gamma \vdash t_2 \equiv t_3 : T}{\Gamma \vdash t_1 \equiv t_3 : T} \quad (\text{Q-Trans})$$

Strong Normalization

Given the following β-reduction rules

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

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

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

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

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

The Calculus of Constructions

The Calculus of Constructions: Syntax

`t` ::=		*terms*
	`s`	*sort*
	`x`	*variable*
	λ`x:t.t`	*abstraction*
	`t t`	*application*
	$(x{:}t) \to t$	*dependent type*
`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

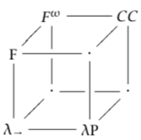
Theorem [*Strong Normalization*]: if $\Gamma \vdash t : T$, then there is no infinite sequence of terms t_i such that $t = t_1$ and $t_i \longrightarrow t_{i+1}$.

Question : Why the property is important?

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



The system λP is λLF in PTS-style.

Dependent Types in Practice

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.

Proofs in Coq: Example

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

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

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

Proofs in Coq: Example, Continued

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

Fixpoint even_rec(m: nat)(p0: (even (double O)))
(pS: forall n:nat,
     (even (double n)) -> (even (double (S n))))
: even (double m) :=
  match m with
    | O => p0
    | S n' => pS n' (even_rec n' p0 pS)
  end.

Definition even_proof: even_prop :=
  fun n => even_rec n even0
    (fun m evenN => (evenS (double m) evenN)).
```

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:

$$\begin{array}{lcl} Vec & : & \mathbb{N} \to * \\ nil & : & Vec\ 0 \\ cons & = & \lambda n{:}\mathbb{N}.D \to Vec\ n \to Vec\ n+1 \end{array}$$

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 $r+1+n = r+n+1$?