# Foundations of Software Fall 2022

Week 11

1

```
Different Kinds of Maps

What is missing?

\begin{array}{cccc} \textit{Term} & \rightarrow & \textit{Term} & (\lambda x.t) \\ \textit{Type} & \rightarrow & \textit{Term} & (\Lambda X.t) \end{array}
```

Different Kinds of Maps

What is missing?

```
\begin{array}{ccccc} \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

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 the type-level.

 $\lambda X :: K.T$ 

4

#### 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.

 $\lambda X :: K.T$ 

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].

```
 \begin{array}{lll} * & & \text{proper types, e.g. } \textit{Bool, Int} \rightarrow \textit{Int} \\ * \Rightarrow * & & \text{type operators: map proper types to proper types} \\ * \Rightarrow * \Rightarrow * & & \text{two-argument operators} \\ (* \Rightarrow *) \Rightarrow * & & \text{type operators: map type operators to proper types} \\ \end{array}
```

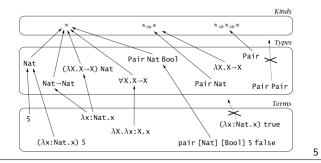
5

#### **Kinding**

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

```
 \begin{array}{ll} * & \text{proper types, e.g. } \textit{Bool, Int} \rightarrow \textit{Int} \\ * \Rightarrow * & \text{type operators: map proper types to proper types} \\ * \Rightarrow * \Rightarrow * & \text{two-argument operators} \\ \end{array}
```

 $(*\Rightarrow *)\Rightarrow *$  type operators: map type operators to proper types



#### Equivalence of Types

Problem: all the types below are equivalent

```
egin{aligned} \textit{Nat} & \rightarrow \textit{Bool} & \textit{Nat} & \rightarrow \textit{Id} \; \textit{Bool} & \textit{Id} \; \textit{Nat} & \rightarrow \textit{Id} \; \textit{Bool} \\ \textit{Id} \; \textit{Nat} & \rightarrow \; \textit{Bool} & \textit{Id} \; (\textit{Nat} & \rightarrow \; \textit{Bool}) & \textit{Id} \; (\textit{Id} \; (\textit{Id} \; \textit{Nat} & \rightarrow \; \textit{Bool}) \\ \end{aligned}
```

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

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

And we need one typing rule:

$$\frac{\Gamma \vdash t : S \qquad S \equiv T}{\Gamma \vdash t : T}$$
 (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.

7

#### System $F_{\omega}$ — Syntax

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

$$t ::= \dots$$

$$\lambda X :: K.t$$

$$T ::=$$

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

 $K ::= {* \atop K \Rightarrow K}$ 

types

type variable type of functions universal type operator abstraction operator application

type abstraction

kinds kind of proper types kind of operators

8

10

## System $F_{\omega}$ — Semantics

$$\frac{t_1 \longrightarrow t_1'}{t_1 \ t_2 \longrightarrow t_1' \ t_2} \tag{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 \ t_2 \longrightarrow t_1 \ t_2'} \tag{E-App2}$$

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

$$\frac{t\longrightarrow t'}{t\left[T\right]\longrightarrow t'\left[T\right]}\tag{E-TAPP}$$

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

9

## System $F_{\omega}$ — Kinding

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

$$\frac{\Gamma, X :: \mathcal{K}_1 \vdash \mathcal{T}_2 : \mathcal{K}_2}{\Gamma \vdash \lambda X :: \mathcal{K}_1 . \mathcal{T}_2 :: \mathcal{K}_1 \Rightarrow \mathcal{K}_2}$$
 (K-Abs)

$$\frac{\Gamma \vdash T_1 : \mathcal{K}_1 \Rightarrow \mathcal{K}_2 \qquad \Gamma \vdash T_2 : \mathcal{K}_1}{\Gamma \vdash T_1 \ T_2 :: \mathcal{K}_2} \tag{K-App)}$$

$$\frac{\Gamma \vdash \mathcal{T}_1 : * \qquad \Gamma \vdash \mathcal{T}_2 : *}{\Gamma \vdash \mathcal{T}_1 \to \mathcal{T}_2 : *} \tag{K-Arrow)}$$

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

System  $F_{\omega}$  — Type Equivalence

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

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}$$
 (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)

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \ S_2 \equiv T_1 \ T_2} \tag{Q-App)}$$

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

#### System $F_{\omega}$ — Typing

$$\frac{x:T\in\Gamma}{\Gamma\vdash x:T}\tag{T-VAR}$$

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

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

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

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

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

#### Example

```
type \ PairRep[Pair :: * \Rightarrow * \Rightarrow *] = \{ \\ pair : \forall X. \forall Y. X \rightarrow Y \rightarrow (Pair \ X \ Y), \\ fst : \forall X. \forall Y. (Pair \ X \ Y) \rightarrow X, \\ snd : \forall X. \forall Y. (Pair \ X \ Y) \rightarrow Y \\ \} \\ \\ def \ swap[Pair :: * \Rightarrow * \Rightarrow *, 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.

13

## 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

15

#### Why Does It Matter?

Example 1. Track length of vectors in types:

```
\textit{NVec} :: \textit{Nat} \rightarrow *
first : (n:Nat) \rightarrow NVec(n+1) \rightarrow Nat
```

 $(x:S) \to T$  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:

```
\textit{NVec} :: \textit{Nat} \rightarrow *
first : (n:Nat) \rightarrow NVec (n+1) \rightarrow Nat
```

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

```
: (f:Format) \rightarrow Data(f) \rightarrow String
sprintf
\begin{array}{lll} \textit{Data}([]) & = & \textit{Unit} \\ \textit{Data}('\%' :: 'd' :: cs) & = & \textit{Nat} * \textit{Data}(cs) \\ \textit{Data}('\%' :: 's' :: cs) & = & \textit{String} * \textit{Data}(cs) \\ \end{array}
Data(c :: cs)
                                                          = Data(cs)
```

16

#### Dependent Function Type (a.k.a. ☐ Types)

A dependent function type is inhabited by a dependent function:

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

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

A dependent function type is inhabited by a dependent function:

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

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

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

17

16

## The Calculus of Constructions

The Calculus of Constructions: Syntax t ::= terms sort variable  $\lambda x:t.t$ abstraction application t t  $(x:t) \rightarrow t$ dependent type sort of proper types sort of kinds contextsempty context Γ, *x*: *T* term variable binding The semantics is the usual  $\beta$ -reduction.

18

20

#### The Calculus of Constructions: Typing

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

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

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

$$\frac{\Gamma \vdash S : s_1 \qquad \Gamma, x : S \vdash T : s_2}{\Gamma \vdash (x : S) \to T : s_2} \tag{T-PI}$$

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

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

Four Kinds of Lambdas

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

21

#### Four Kinds of Lambdas

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

Four Kinds of Lambdas

Example	Туре
$\lambda x: \mathbb{N}.x + 1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f: \mathbb{N} \to \mathbb{N}.f \times$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X:*.\lambda x:X.x$	$(X:*) \rightarrow X \rightarrow X$
$\lambda F$ :* $\rightarrow$ *. $\lambda x$ : $F \mathbb{N}$ . $x$	$(F:* o *) o (F\;\mathbb{N}) o (F\;\mathbb{N})$
$\lambda X$ :*. $X$	$* \rightarrow *$
$\lambda F$ :* $\rightarrow$ *. $F$ $\mathbb{N}$	$(* \rightarrow *) \rightarrow *$

21

21

#### Four Kinds of Lambdas

Example	Туре
$\lambda x: \mathbb{N}.x + 1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f: \mathbb{N} \to \mathbb{N}.f \times$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X$ :*. $\lambda x$ : $X$ . $x$	$(X:*) \rightarrow X \rightarrow X$
$\lambda F:* \to *.\lambda x: F \mathbb{N}.x$	$(F:* o *) o (F\;\mathbb{N}) o (F\;\mathbb{N})$
$\lambda X$ :*. $X$	$* \rightarrow *$
$\lambda F$ :* $\rightarrow$ *. $F$ $\mathbb{N}$	(*  o *)  o *
$\lambda n:\mathbb{N}.NVec\ n$	$\mathbb{N} \to *$
$\lambda f: \mathbb{N} \to \mathbb{N}.NVec (f 6)$	$(\mathbb{N} \to \mathbb{N}) \to *$

## Strong Normalization

Given the following  $\beta$ -reduction rules

$$\frac{t_1 \longrightarrow t_1'}{\lambda x : \mathcal{T}_1.t_1 \longrightarrow \lambda x : \mathcal{T}_1.t_1'} \qquad \qquad (\beta\text{-Abs})$$

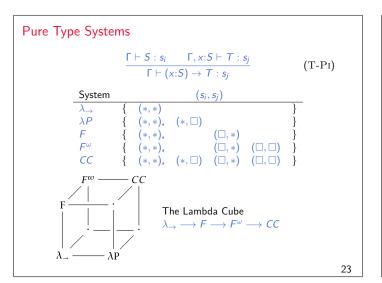
$$rac{t_1 \longrightarrow t_1'}{t_1 \ t_2 \longrightarrow t_1' \ t_2}$$
 (\beta-App1)

$$\frac{t_2 \longrightarrow t_2'}{t_1 \ t_2 \longrightarrow t_1 \ t_2'} \tag{\beta-APP2}$$

$$(\lambda x: T_1.t_1)t_2 \longrightarrow [x \mapsto t_2]t_1$$
  $(\beta$ -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  $t_i \longrightarrow t_{i+1}$ .

22



# Dependent Types in Coq

24

#### **Proof Assistants**

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

By Curry-Howard Correspondence

- ightharpoonup proofs  $\longleftrightarrow$  programs
- ▶ propositions ←→ types

#### **Proof Assistants**

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

By Curry-Howard Correspondence

- ightharpoonup proofs  $\longleftrightarrow$  programs
- ▶ propositions ←→ types

Two impactful projects based on Coq:

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

25

```
Type Universes in Coq

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

\Gamma \vdash Prop : Type_1
\Gamma \vdash Set : Type_1
\Gamma \vdash Type_i : Type_{i+1}
\frac{\Gamma, x:A \vdash B : Prop \qquad \Gamma \vdash A : s}{\Gamma \vdash (x : A) \rightarrow B : Prop}
\frac{\Gamma, x:A \vdash B : Set \qquad \Gamma \vdash A : s \qquad s \in \{Prop, Set\}}{\Gamma \vdash (x : A) \rightarrow B : Set}
\frac{\Gamma, x:A \vdash B : Type_i \qquad \Gamma \vdash A : Type_i}{\Gamma \vdash (x : A) \rightarrow B : Type_i}
```

```
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
```

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

Fixpoint double (n : nat) : nat :=
    match n with
    | 0 => 0
    | S n' => S (S (double n'))
    end.

Recursion has to be structural.

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

30

#### Recap: Curry-Howard Correspondence

Propositions as types in the context of intuitionistic logic.

Proposition	Term & Type
$A \wedge B$	t:(A,B)
$A \lor B$	t: A + B
A  o B	t:A o B
	t : False
$\neg A$	$t: A  ightarrow  extit{False}$
∀ <i>x</i> : <i>A</i> . <i>B</i>	$t:(x:A)\to B$
∃ <i>x</i> : <i>A</i> . <i>B</i>	t: (x:A, B)

Curry-Howard correspondence in Coq

28

```
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

```
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 :=
lor_introl : A -> A \/ B
lor_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 :=
lor_introl : A -> A \/ B
lor_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.
```

 $_{\rm 1}$   $\,$  Inductive False : Prop :=.

30

30

#### Curry-Howard correspondence in Coq

#### Curry-Howard correspondence in Coq - continued

30

```
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 Cog - 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.
```

31

#### 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: ∀P.P ∨ ¬P
    DNE: ∀P.¬¬P → 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: ∀P.P ∨ ¬P
    DNE: ∀P.¬¬P → 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.

32

31

#### 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: ∀P.P ∨ ¬P
    DNE: ∀P.¬¬P → 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?

32

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: ∀P.P ∨ ¬P
    DNE: ∀P.¬¬P → 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.
```

32

```
\mathsf{LEM} \to \mathsf{DNE}
```

```
Definition LEM_To_DNE :=

fun (lem: forall P : Prop, P \/ ~ P) (Q:Prop) (q: ~~Q)

match lem Q with

or_introl 1 =>

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

Check LEM_To_DNE : LEM -> DNE.
```

 $\mathsf{DNE} \to \mathsf{LEM}$ 

```
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.

35

#### Problem with Type Checking

Value constructors:

```
 \begin{array}{lll} \textit{NVec} & : & \mathbb{N} \to * \\ \textit{nil} & : & \textit{NVec} \ 0 \\ \textit{cons} & : & \mathbb{N} \to (n .: \mathbb{N}) \to \textit{NVec} \ n \to \textit{NVec} \ n + 1 \\ \end{array}
```

Appending vectors:

```
\begin{array}{ll} \textit{append} & : & (m:\mathbb{N}) \to (n:\mathbb{N}) \to \textit{NVec } m \to \textit{NVec } n \to \textit{NVec } (n+m) \\ \textit{append} & = & \lambda m:\mathbb{N}.\,\lambda n:\mathbb{N}.\,\lambda l:\textit{NVec } m.\,\lambda t:\textit{NVec } n. \\ & & \textit{match } l \textit{ with } \\ & | \textit{nil} \Rightarrow t \\ & | \textit{cons } x \textit{ r } y \Rightarrow \textit{cons } x \textit{ (r+n) } (\textit{append } r \textit{ n } y \textit{ t}) \end{array}
```

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