Foundations of Software
Spring 2025

Week 12

Different Kinds of functions

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 functions

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

Defining Pairs: System F vs. Type-Level Functions

Pair in System F

Pair as a Type-Level Function

Defining Pairs: System F vs. Type-Level Functions

Pair in System F

- **Representation**: For given types $A, B : \mathit{Type}$: $\mathrm{Pair}(A, B) = \forall X.(A \to B \to X) \to X$

Pair as a Type-Level Function

- **Representation**: Defined as an explicit function operating on types. $\mathrm{Pair}_\omega = \lambda A : \mathit{Type}.\lambda B : \mathit{Type}.\mathrm{Pair}(A, B)$

Defining Pairs: System F vs. Type-Level Functions

Pair in System F

- **Representation**: For given types $A, B : Type$: $\text{Pair}(A, B) = \forall X.(A \to B \to X) \to X$
- **Nature of "Pair"**: For any concrete types A, B, we can write down the type $\text{Pair}(A, B)$. "Pair" itself is not a first-class, manipulable function within System F's type language.

Pair as a Type-Level Function

- **Representation**: Defined as an explicit function operating on types. $\text{Pair}_\omega = \lambda A : Type.\lambda B : Type.\text{Pair}(A, B)$
- **Nature of "Pair$_\omega$"**: It has a type: $\text{Pair}_\omega : Type \to Type \to Type$. Pair_ω could be bound to a variable, passed as an argument, etc...

Defining Pairs: System F vs. Type-Level Functions

Pair in System F

- **Representation**: For given types $A, B : Type$: $\text{Pair}(A, B) = \forall X.(A \to B \to X) \to X$
- **Nature of "Pair"**: For any concrete types A, B, we can write down the type $\text{Pair}(A, B)$. "Pair" itself is not a first-class, manipulable function within System F's type language.

Pair as a Type-Level Function

- **Representation**: Defined as an explicit function operating on types. $\text{Pair}_\omega = \lambda A : Type.\lambda B : Type.\text{Pair}(A, B)$
- **Nature of "Pair$_\omega$"**: It has a type: $\text{Pair}_\omega : Type \to Type \to Type$. Pair_ω could be bound to a variable, passed as an argument, etc...

Key Difference: We would like to allow type constructors (like Pair_ω) to be treated as first-class citizens (functions) at the type level, enabling abstraction over them.

Defining Pairs: System F vs. Type-Level Functions

We can represent pairs in different ways, with different implications for how we can manipulate "Pair-ness" itself at the type level.
In system F: It's a *schema* or a *definition pattern*. Its "constructor" behavior is observed externally (meta-theoretically).

Defining Pairs: System F vs. Type-Level Functions

We can represent pairs in different ways, with different implications for how we can manipulate "Pair-ness" itself at the type level.
In system F: It's a *schema* or a *definition pattern*. Its "constructor" behavior is observed externally (meta-theoretically).

Limitation: Cannot easily abstract over "Pair" itself. We can't pass the "Pair-making-ability" (Pair) as an argument to another (polymorphic) type definition within System F directly.

Defining Pairs: System F vs. Type-Level Functions

We can represent pairs in different ways, with different implications for how we can manipulate "Pair-ness" itself at the type level.
In system F: It's a *schema* or a *definition pattern*. Its "constructor" behavior is observed externally (meta-theoretically).

Limitation: Cannot easily abstract over "Pair" itself. We can't pass the "Pair-making-ability" (Pair) as an argument to another (polymorphic) type definition within System F directly.

Advantages of first class type-level functions: Enables higher-order type-level programming, allows defining generic type combinators that operate on other type constructors (like 'Pair', 'List', 'Option').

Example 1 - Scala

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

The method *swap* works for any representation of pairs.

Example 2 - OCaml

```
module type ORDERED_ELEMENT = sig
  type t
  val compare : t -> t -> int
end
module type WRAPPER_CONSTRUCTOR = sig
  type 'a t
  val lift_compare :
   ('x -> 'x -> int) -> ('x t -> 'x t -> int)
end
module MakeSetOfWrappedTypes
  (Wrapper : WRAPPER_CONSTRUCTOR)
  (Element : ORDERED_ELEMENT)
  : Set.S  ...
```

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Spoiler: We can. Two subtleties to look for:

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Spoiler: We can. Two subtleties to look for:

- Reducing type-level functions and equivalence of types
- "Type checking" of type operators

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Spoiler: We can. Two subtleties to look for:

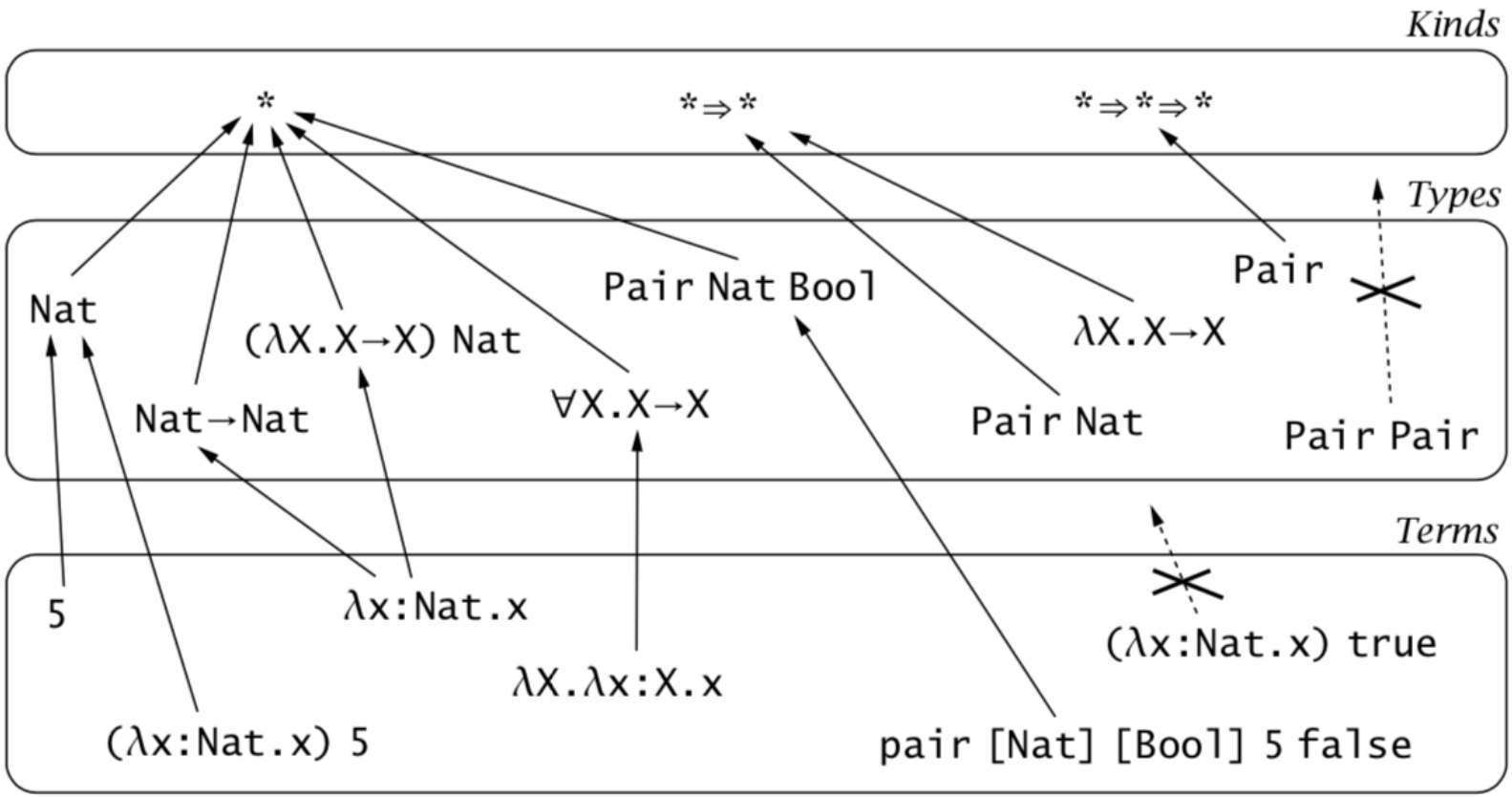
- Reducing type-level functions and equivalence of types
- "Type checking" of type operators

The Types (or Kinds) of type-level terms and functions

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

The Types (or Kinds) of type-level terms and functions

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



Equivalence of Types

Problem: all the types below are equivalent

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

We need to introduce a *definitional equivalence* relation on types, written $S \equiv T$.
And we need one *new* typing rule:

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

$S \equiv T$ if and only if S and T have the same normal form.

The temptation is too great

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Lambda-calculus at the type-level

Idea: We could define a new languages to manipulate type-level functions!

Little Detour: We first *briefly* succomb to the temptation and show how we could define a new language to manipulate type-level functions. And we then avoid going there, and instead try to *recycle* our perfectly good existing lambda-terms.

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*

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 $t \longrightarrow t'$.

Limitation 2: Dependent Types

Limitation of types for programming

Example 1. Track length of vectors in types:

$$\begin{array}{lcl} \mathit{NVec} & :: & \mathit{Nat} \to * \\ \mathit{first} & : & \forall(n{:}\mathit{Nat}).\mathit{NVec}\ (n+1) \to \mathit{Nat} \end{array}$$

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

Limitation of types for programming

Example 1. Track length of vectors in types:

$$\begin{array}{lcl} \mathit{NVec} & :: & \mathit{Nat} \to * \\ \mathit{first} & : & \forall(n{:}\mathit{Nat}).\mathit{NVec}\ (n+1) \to \mathit{Nat} \end{array}$$

$\forall(x{:}S).T$ 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*:

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

Limitation of types for logic

No notion of equality in System F!

Limitation of types for logic

No notion of equality in System F!

We can't even prove that $0 + 1 = 1$!

Limitation of types for logic

No notion of equality in System F!

We can't even prove that $0 + 1 = 1$!

How are we going to be able to prove properties about programs?

Dependent Function Type (a.k.a. $\forall$ or Π Types)

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

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

Dependent Function Type (a.k.a. $\forall$ or Π Types)

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

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

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

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

Example: Value whose type depends on a boolean condition

- Let `Bool` be the type with values `true` and `false`.
- Let `Int` be the type of integers, and `String` be the type of text strings.
- Consider a function `conditionalValue` that takes a boolean `b : Bool`:
  - If `b` is `true`, it returns an `Int`.
  - If `b` is `false`, it returns a `String`.

Dependent Function Type - an example

Consider a function `conditionalValue` that takes a boolean `b : Bool`:

- If `b` is `true`, it returns an `Int`.
- If `b` is `false`, it returns a `String`.

Dependent Function Type - an example

Consider a function `conditionalValue` that takes a boolean `b : Bool`:

- If `b` is `true`, it returns an `Int`.
- If `b` is `false`, it returns a `String`.

The type of such a function is a dependent function type:

$$\texttt{conditionalValue} : \forall(b : \texttt{Bool}).(\text{if } b \text{ then } \texttt{Int} \text{ else } \texttt{String})$$

Dependent Function Type - an example

Consider a function `conditionalValue` that takes a boolean `b : Bool`:

- If `b` is `true`, it returns an `Int`.
- If `b` is `false`, it returns a `String`.

The type of such a function is a dependent function type:

$$\texttt{conditionalValue} : \forall(b : \texttt{Bool}).(\text{if } b \text{ then } \texttt{Int} \text{ else } \texttt{String})$$

- The input type S is `Bool`.
- The input variable x is `b`.
- The return type T is `(if b then Int else String)`, which critically depends on the value of `b`.

Dependent Function Type - an example

Consider a function `conditionalValue` that takes a boolean `b : Bool`:

- If `b` is `true`, it returns an `Int`.
- If `b` is `false`, it returns a `String`.

The type of such a function is a dependent function type:

$$\texttt{conditionalValue} : \forall(b : \texttt{Bool}).(\text{if } b \text{ then } \texttt{Int} \text{ else } \texttt{String})$$

- The input type S is `Bool`.
- The input variable x is `b`.
- The return type T is `(if b then Int else String)`, which critically depends on the value of b.

An implementation sketch (in an ideal world) for such a function:

$$\texttt{conditionalValue} = \lambda b : \texttt{Bool}.\ \text{if } b \text{ then } 42 \text{ else } \text{"hello"}$$

Killing two birds with one stone: The Calculus of Constructions

The Calculus of Constructions: Syntax

$\mathtt{t}$ ::=		*terms*
	$\mathtt{x}$	*variable*
	$\lambda \mathtt{x}:\mathtt{t}.\mathtt{t}$	*abstraction*
	$\mathtt{t}\ \mathtt{t}$	*application*
	$\mathtt{s}$	*sort*
	$\forall(x{:}t).t$	*dependent type*
$\mathtt{s}$ ::=		*sorts*
	$\mathtt{U}_0$	*classic types*
	$\mathtt{U}_1$	*types of types (kinds)*
	$\mathtt{U}_2$	*types of types of types*
Γ ::=		*contexts*
	$\emptyset$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*

The semantics is the usual β-reduction.

Reduction Rules

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

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

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

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

The Calculus of Constructions: Typing

$$\vdash U_i : U_{i+1} \text{ (T-AxiomT)} \qquad \frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)}$$

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

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

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

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

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

Hard Theorem

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

Hard Theorem

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

Why is it hard?

Hard Theorem

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

Why is it hard?

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
$\lambda x{:}\mathbb{N}.x+1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.f\ x$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X{:}U_0.\lambda x{:}X.\,x$	$\forall(X{:}U_0).X \to X$
$\lambda F{:}U_0 \to U_0.\lambda x{:}F\ \mathbb{N}.\,x$	$\forall(F{:}U_0 \to U_0).(F\ \mathbb{N}) \to (F\ \mathbb{N})$

Four Kinds of Lambdas

Example	Type
$\lambda x{:}\mathbb{N}.x+1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.f\ x$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X{:}U_0.\lambda x{:}X.\,x$	$\forall(X{:}U_0).X \to X$
$\lambda F{:}U_0 \to U_0.\lambda x{:}F\ \mathbb{N}.\,x$	$\forall(F{:}U_0 \to U_0).(F\ \mathbb{N}) \to (F\ \mathbb{N})$
$\lambda X{:}U_0.X$	$U_0 \to U_0$
$\lambda F{:}U_0 \to U_0.F\ \mathbb{N}$	$(U_0 \to U_0) \to U_0$

Four Kinds of Lambdas

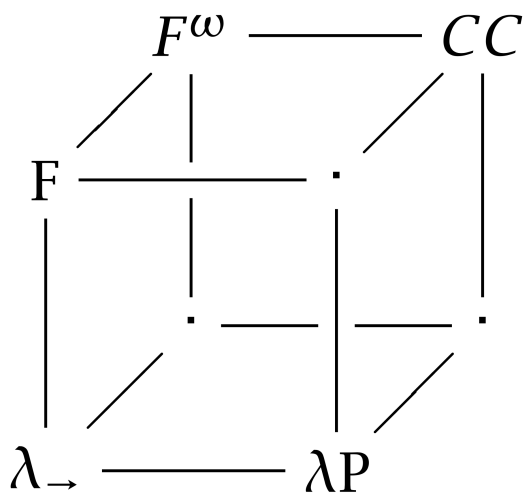
Example	Type
$\lambda x{:}\mathbb{N}.x+1$	$\mathbb{N} \to \mathbb{N}$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.f\ x$	$(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$\lambda X{:}U_0.\lambda x{:}X.\,x$	$\forall(X{:}U_0).X \to X$
$\lambda F{:}U_0 \to U_0.\lambda x{:}F\ \mathbb{N}.\,x$	$\forall(F{:}U_0 \to U_0).(F\ \mathbb{N}) \to (F\ \mathbb{N})$
$\lambda X{:}U_0.X$	$U_0 \to U_0$
$\lambda F{:}U_0 \to U_0.F\ \mathbb{N}$	$(U_0 \to U_0) \to U_0$
$\lambda n{:}\mathbb{N}.NVec\ n$	$\mathbb{N} \to U_0$
$\lambda f{:}\mathbb{N} \to \mathbb{N}.NVec\ (f\ 6)$	$(\mathbb{N} \to \mathbb{N}) \to U_0$

We have almost reached the point where the *object* language is the same as Lean's ambient language!

Pure Type Systems

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

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



The Lambda Cube

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

What to do with dependent types

Leibniz's equality

Leibniz Equality: The Identity of Indiscernibles

- **Fundamental Question:** What does it mean for two things, say x and y, to be equal?
- **Leibniz's Principle:** Two objects are identical if and only if they share all the same properties.
  - If x has any property P, then y must also have property P.
  - (And often vice-versa: if y has P, then x has P).

Leibniz Equality: The Identity of Indiscernibles

- **Fundamental Question:** What does it mean for two things, say x and y, to be equal?
- **Leibniz's Principle:** Two objects are identical if and only if they share all the same properties.
  - If x has any property P, then y must also have property P.
  - (And often vice-versa: if y has P, then x has P).
- **In essence:** x and y are the same if one can be substituted for the other in any statement *salva veritate* (without changing the truth value).
- 💡 This provides a way to define equality based on what we can *observe* or *predicate* about objects.

Formal Definition (in Lean 4)

Leibniz Equality Definition

```
def LeibnizEq {alpha : Type u} (x y : alpha) : Prop :=
  forall (P : alpha -> Prop), P x -> P y
```

Formal Definition (in Lean 4)

Leibniz Equality Definition

```
def LeibnizEq {alpha : Type u} (x y : alpha) : Prop :=
  forall (P : alpha -> Prop), P x -> P y
```

This definition essentially states: *y* possesses every property that *x* possesses.

Core Properties

1. **Reflexivity, transitivity, symmetry** Let's do it together!

Core Properties

1. **Reflexivity, transitivity, symmetry** Let's do it together!
2. **Substitutivity (Indiscernibility of Identicals):**
   - The very definition `LeibnizEq` $x\,y$ *is* the principle of substitutivity.
   - If `LeibnizEq` $x\,y$ holds, then y can be substituted for x wherever x appears in a property P such that Px is true, and Py will also be true.
   - This is the foundation for "rewriting" based on equality.

Type Universes in Lean/Coq

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

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

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

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

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

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

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

Math Vs Logic

Addressing what we diligently ignored: LEM

In intuitionistic logics (System F, Calculus of Constructions, etc...), 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.

Addressing what we diligently ignored: LEM

In intuitionistic logics (System F, Calculus of Constructions, etc...), 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.

Addressing what we diligently ignored: LEM

In intuitionistic logics (System F, Calculus of Constructions, etc...), 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?

Addressing what we diligently ignored: LEM

In intuitionistic logics (System F, Calculus of Constructions, etc...), 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 also prove that LEM is equivalent to DNE.

Limitations of Calculus of Constructions for Logic: Lack of support for induction!

We did all that work for nothing!

Limitations of Calculus of Constructions for Logic: Lack of support for induction!

We did all that work for nothing!

We still cannot prove $1 + n = n + 1$ on our Church-encoded numbers :(.

Limitations of Calculus of Constructions for Logic: Lack of support for induction!

We did all that work for nothing!

We still cannot prove $1 + n = n + 1$ on our Church-encoded numbers :(.

What we *can* do (Coquand et Huet, 1985):

2. Logical constructions

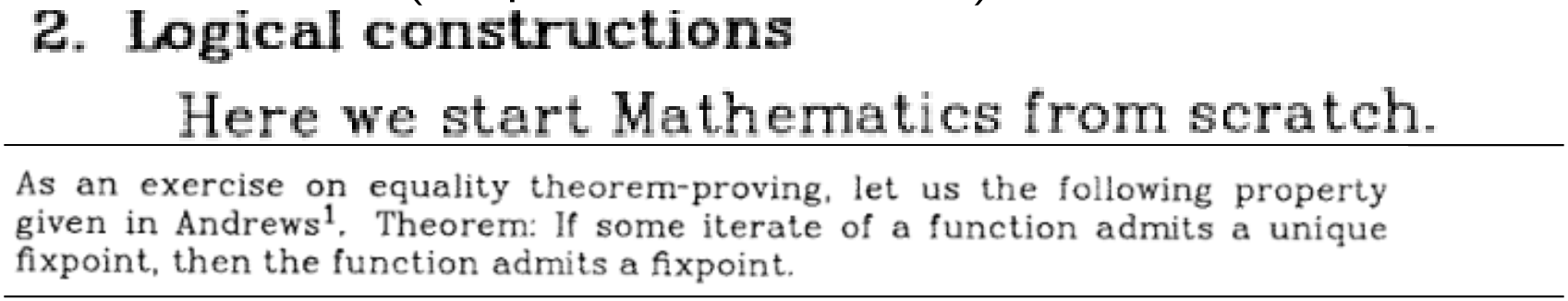
Here we start Mathematics from scratch.

Limitations of Calculus of Constructions for Logic: Lack of support for induction!

We did all that work for nothing!

We still cannot prove $1 + n = n + 1$ on our Church-encoded numbers :(.

What we *can* do (Coquand et Huet, 1985):



2. Logical constructions

Here we start Mathematics from scratch.

As an exercise on equality theorem-proving, let us the following property given in Andrews[1]. Theorem: If some iterate of a function admits a unique fixpoint, then the function admits a fixpoint.

Limitations of Calculus of Constructions for Logic: Lack of support for induction!

We did all that work for nothing!

We still cannot prove $1 + n = n + 1$ on our Church-encoded numbers :(.

What we *can* do (Coquand et Huet, 1985):

2. Logical constructions

Here we start Mathematics from scratch.

As an exercise on equality theorem-proving, let us the following property given in Andrews[1]. Theorem: If some iterate of a function admits a unique fixpoint, then the function admits a fixpoint.

5.3. Tarski's theorem

As an exercise in relational constructions, let us now show Tarski's theorem : If a function is monotonous over a complete upper-semi-lattice, it admits a fixpoint[57].

Don't forget to admit to limitations!

This completes the description of our prototype implementation. Actually, we must admit a little cheating on the parser's part: the parser knows beforehand about the mixfix syntax of a few constants. This is because we do not know how to modify dynamically the tables of the parser generated by Yacc. This is an unimportant technical detail the reader need not be concerned about, since our parser is consistent with all the *LET_SYNTAX* commands.

Don't forget to admit to limitations!

This completes the description of our prototype implementation. Actually, we must admit a little cheating on the parser's part: the parser knows beforehand about the mixfix syntax of a few constants. This is because we do not know how to modify dynamically the tables of the parser generated by Yacc. This is an unimportant technical detail the reader need not be concerned about, since our parser is consistent with all the *LET_SYNTAX* commands.

```
let COMPL_SEMI_LATTICE =
    "{M|A->*}!B.([u:A](bound u M)->([x:A](bound x M)->(R u x))->B)->B";;
PROP 'csl' COMPL_SEMI_LATTICE;;

let TARSKI = "(Trans R) -> csl -> [f:A->A](mon f) -> <A>Sig((fix f))";;

let TARSKI_PROOF = "[trans:(Trans R)][ub:csl][f:A->A][mo:(mon f)]
    !B.[h:[u:A]((R u (f u))&(R (f u) u))->B]
    (ub ([u:A](R u (f u))) B
       ([x:A][h1:[u:A](R u (f u))->(R u x)]
          [h2:[u:A]([v:A](R v (f v))->(R v u))->(R x u)]
  let p = ([y:A][h':(R y (f y))](trans y (f y) (f x) h' (mo y x (h1 y h'))))
    in let xRfx = (h2 (f x) p) in
    (h x <(R x (f x)),(R (f x) x)>(xRfx,(h1 (f x) (mo x (f x) xRfx))))))";;
```

Source of all evil : problem with type Checking

Value constructors:

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

Appending vectors:

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

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

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. While it is decidable in Coq/Lean, implementing full-blown calculus of construction would not be decidable in Scala/Haskell, why?