Design Principles of Programming Languages
编程语言的设计原理

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2024

Type-Level Computation

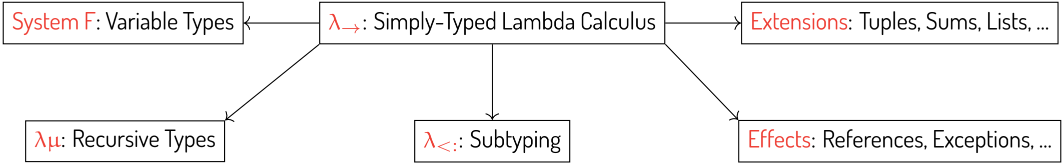
类型层计算

We Have Studied ...

Principle

The uses of type systems go beyond detecting errors.

- Type systems offer support for **abstraction**, **safety**, **efficiency**, ...
- Language design goes **hand-in-hand** with type-system design.



Observation

Different **combinations** lead to different languages.

- System F + $\lambda\mu$ supports polymorphic recursive types.
- System F + $\lambda_{<:}$ supports bounded quantification (see Chap. 26).

The Essence of λ

Principle (Computation)

λ-abstraction is **THE** mechanism of defining computation.

- In $\lambda_{\rightarrow}$, $\lambda x{:}T.\ t$ abstracts **terms** out of **terms**.
- In System F, $\lambda X.\ t$ abstracts **terms** out of **types**.

Principle (Characterization of Computation)

Typing is **THE** mechanism of characterizing computation.

- Syntactically: **types** characterize **terms**.
- Semantically: a **type** denotes a set of **terms** that evaluates to particular values.

Question

Can we introduce computation to the type level?
How to characterize such type-level computation?

Type Operators

Remark

We have seen **parametric** type definitions:

$$\mathtt{Pair}_{\mathtt{T1,T2}} = \forall \mathtt{X}.\ (\mathtt{T1} \to \mathtt{T2} \to \mathtt{X}) \to \mathtt{X};$$
$$\mathtt{Sum}_{\mathtt{T1,T2}} = \forall \mathtt{X}.\ (\mathtt{T1} \to \mathtt{X}) \to (\mathtt{T2} \to \mathtt{X}) \to \mathtt{X};$$
$$\mathtt{List}_{\mathtt{T}} = \forall \mathtt{x}.\ (\mathtt{T} \to \mathtt{X} \to \mathtt{X}) \to \mathtt{X} \to \mathtt{X};$$

Observation

Pair, Sum, and List behave like **type-level functions**!

$$\mathtt{Pair} = \lambda\mathtt{T1}.\ \lambda\mathtt{T2}.\ (\forall \mathtt{X}.\ (\mathtt{T1} \to \mathtt{T2} \to \mathtt{X}) \to \mathtt{X});$$
$$\mathtt{Sum} = \lambda\mathtt{T1}.\ \lambda\mathtt{T2}.\ (\forall \mathtt{X}.\ (\mathtt{T1} \to \mathtt{X}) \to (\mathtt{T2} \to \mathtt{X}) \to \mathtt{X});$$
$$\mathtt{List} = \lambda\mathtt{T}.\ (\forall \mathtt{x}.\ (\mathtt{T} \to \mathtt{X} \to \mathtt{X}) \to \mathtt{X} \to \mathtt{X});$$

Type-Level Computation

Principle (Type-Level Computation)

λ-abstraction is **THE** mechanism of defining computation.

```
Pair = λT1. λT2. (∀X. (T1→T2→X) → X);
Sum = λT1. λT2. (∀X. (T1→X) → (T2→X) → X);
List = λT. (∀x. (T→X→X) → X → X);
```

We introduce $\lambda X.\ T$ to abstract **types** out of **types**.

Observation

Type-level computation allows writing the **same** type in **different** ways.

Example

Consider `Id = λX. X`. The following types are equivalent:

`Nat→Bool` `Nat→Id Bool` `Id Nat→Id Bool` `Id Nat→Bool` `Id (Nat→Bool)`

Type-Level Abstraction & Application

Syntax

$$\mathsf{T} ::= \color{red}{\mathsf{X}} \mid \color{red}{\lambda \mathsf{X}.\,\mathsf{T}} \mid \color{red}{\mathsf{T}\ \mathsf{T}} \mid \mathsf{T} \to \mathsf{T} \mid \texttt{Bool} \mid \texttt{Nat} \mid \ldots$$

$$\mathsf{TV} ::= \color{red}{\lambda \mathsf{X}.\,\mathsf{T}} \mid \mathsf{TV} \to \mathsf{TV} \mid \texttt{Bool} \mid \texttt{Nat} \mid \ldots$$

Evaluation: $\mathsf{T} \longrightarrow \mathsf{T}'$

$$\frac{\mathsf{T}_1 \longrightarrow \mathsf{T}_1'}{\mathsf{T}_1\ \mathsf{T}_2 \longrightarrow \mathsf{T}_1'\ \mathsf{T}_2} \qquad \frac{\mathsf{T}_2 \longrightarrow \mathsf{T}_2'}{\mathsf{TV}_1\ \mathsf{T}_2 \longrightarrow \mathsf{TV}_1\ \mathsf{T}_2'} \qquad \frac{}{(\lambda \mathsf{X}.\,\mathsf{T}_{12})\ \mathsf{TV}_2 \longrightarrow [\mathsf{X} \mapsto \mathsf{TV}_2]\mathsf{T}_{12}}$$

$$\frac{\mathsf{T}_1 \longrightarrow \mathsf{T}_1'}{(\mathsf{T}_1 \to \mathsf{T}_2) \longrightarrow (\mathsf{T}_1' \to \mathsf{T}_2)} \qquad \frac{\mathsf{T}_2 \longrightarrow \mathsf{T}_2'}{(\mathsf{TV}_1 \to \mathsf{T}_2) \longrightarrow (\mathsf{TV}_1 \to \mathsf{T}_2')}$$

Question

It seems that we formulate a type-level **untyped** lambda calculus. **Any issues?**

Issue 1: Unequal Equivalent Types

Example

Consider `Id = λX. X`. Two type-level values `λX. Id X` and `λX. X` are **unequal** but **equivalent**.

Observation

We do not care about how types evaluate.
We care about if they are equivalent.

Equivalence: $S \equiv T$

$$\frac{}{T \equiv T} \qquad \frac{T \equiv S}{S \equiv T} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}$$

$$\frac{S_2 \equiv T_2}{\lambda X.\, S_2 \equiv \lambda X.\, T_2} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \qquad \frac{}{(\lambda X.\, T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12}}$$

Issue 2: Errors in Type-Level Computation

Example

Consider `(λ X. X X) Nat`. The type evaluates to `Nat Nat`, which is an **illy-formed** type.
Consider `(λ X. X X) (λ X. X X)`. The type's evaluation **diverges**.

Principle (Characterization of Type-Level Computation)

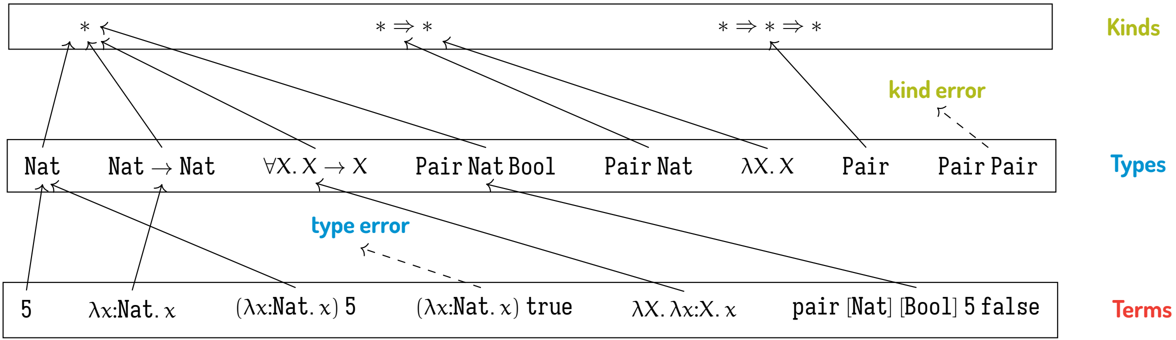
Recall that **types** characterize **terms**.
What can characterize **types**?

Kinds: "Types of Types"

Kinds characterize **types**.

$*$	proper types (e.g., `Bool` and `Nat` $\rightarrow$ `Bool`)
$* \Rightarrow *$	type operators, i.e., functions from proper types to proper types
$* \Rightarrow * \Rightarrow *$	functions from proper types to type operators, i.e., two-argument operators
$(* \Rightarrow *) \Rightarrow *$	functions from type operators to proper types

Terms, Types, and Kinds



Question

- What is the difference between $\forall X.\, X \to X$ and $\lambda X.\, X \to X$?
- Why doesn't an arrow type Nat $\to$ Nat have an arrow kind like $* \Rightarrow *$?

Kinding

Syntax

$$T ::= X \mid \lambda X{::}K.\, T \mid T\ T \mid T \to T \mid \texttt{Bool} \mid \texttt{Nat} \mid \ldots$$

$$K ::= * \mid K \Rightarrow K \qquad\qquad \Gamma ::= \varnothing \mid \Gamma, x : T \mid \Gamma, X :: K$$

$\Gamma \vdash T :: K$: "type T has kind K in context Γ"

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1.\, T_2 :: K_1 \Rightarrow K_2} \qquad \frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \qquad \frac{}{\Gamma \vdash \texttt{Bool} :: *} \qquad \frac{}{\Gamma \vdash \texttt{Nat} :: *}$$

Observation

The **kinding** relation $\Gamma \vdash T :: K$ is very similar to the **typing** relation $\Gamma \vdash t : T$.

$\lambda_\omega = \lambda_\to$ + Type Operators

t	$::=$		*terms:*
		x	*variable*
		$\lambda x{:}T.\, t$	*abstraction*
		$t\ t$	*application*
v	$::=$		*values:*
		$\lambda x{:}T.\, t$	*abstraction value*
T	$::=$		*types:*
		X	*type variable*
		$\lambda X{::}K.\, T$	*operator abstraction*
		$T\ T$	*operator application*
		$T \to T$	*type of functions*
Γ	$::=$		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x : T$	*term variable binding*
		$\Gamma, X :: K$	*type variable binding*
K	$::=$		*kinds:*
		$*$	*kind of proper types*
		$K \Rightarrow K$	*kind of operators*

Typing

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 .\, t_2 : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}}$$

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

Observation

If $\varnothing \vdash t : T$, then $\varnothing \vdash T :: *$.

Question

How to decide type equivalence $S \equiv T$ **algorithmically**?

Approach 1: Parallel Reduction

$S \Rrightarrow T$: "type S parallelly reduces to type T"

$$\frac{}{T \Rrightarrow T} \qquad \frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1 \to S_2 \Rrightarrow T_1 \to T_2} \qquad \frac{S_2 \Rrightarrow T_2}{\lambda X{::}K_1 .\, S_2 \Rrightarrow \lambda X{::}K_1 .\, T_2} \qquad \frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1\ S_2 \Rrightarrow T_1\ T_2}$$

$$\frac{S_{12} \Rrightarrow T_{12} \qquad S_2 \Rrightarrow T_2}{(\lambda X{::}K_{11} .\, S_{12})\ S_2 \Rrightarrow [X \mapsto T_2]T_{12}}$$

Example

Let $S \overset{\text{def}}{=} \mathtt{Id}\,\mathtt{Nat} \to \mathtt{Bool}$ and $T \overset{\text{def}}{=} \mathtt{Id}\,(\mathtt{Nat} \to \mathtt{Bool})$. Then

$$S = ((\lambda X{::}*.\, X)\ \mathtt{Nat}) \to \mathtt{Bool} \Rrightarrow \mathtt{Nat} \to \mathtt{Bool}, \qquad T = (\lambda X{::}*.\, X)\ (\mathtt{Nat} \to \mathtt{Bool}) \Rrightarrow \mathtt{Nat} \to \mathtt{Bool}.$$

Theorem

$S \equiv T$ **if and only if** there exists some U such that $S \Rrightarrow^* U$ and $T \Rrightarrow^* U$.

Approach 2: Weak-Head Reduction

$S \rightsquigarrow T$: "type S weak-head reduces to type T"

Weak-head reduction only reduces **outermost** type-level applications.

$$\frac{T_1 \rightsquigarrow T_1'}{T_1\ T_2 \rightsquigarrow T_1'\ T_2} \qquad \frac{}{(\lambda X{::}K.\, T_{12})\ T_2 \rightsquigarrow [X \mapsto T_2]T_{12}}$$

We denote by $S \Downarrow T$ to mean "type S weak-head normalizes to type T."

$$\frac{T \not\rightsquigarrow}{T \Downarrow T} \qquad \frac{S \rightsquigarrow T \qquad T \Downarrow T'}{S \Downarrow T'}$$

$\Gamma \vdash S \Leftrightarrow T :: K$ and $\Gamma \vdash S \leftrightarrow T :: K$: Algorithmic and Structural Equivalence

$$\frac{S \Downarrow S' \qquad T \Downarrow T' \qquad \Gamma \vdash S \leftrightarrow T :: *}{\Gamma \vdash S \Leftrightarrow T :: *} \qquad \frac{X \notin \Gamma \qquad \Gamma, X :: K_1 \vdash S\ X \Leftrightarrow T\ X :: K_2}{\Gamma \vdash S \Leftrightarrow T :: K_1 \Rightarrow K_2}$$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X \leftrightarrow X :: K} \qquad \frac{\Gamma \vdash S_1 \leftrightarrow T_1 :: * \qquad \Gamma \vdash S_2 \leftrightarrow T_2 :: *}{\Gamma \vdash S_1 \to S_2 \leftrightarrow T_1 \to T_2 :: *} \qquad \frac{\Gamma \vdash S_1 \leftrightarrow T_1 :: K_1 \Rightarrow K_2 \qquad \Gamma \vdash S_2 \Leftrightarrow T_2 :: K_1}{\Gamma \vdash S_1\ S_2 \leftrightarrow T_1\ T_2 :: K_2}$$

Parallel Reduction vs. Weak-Head Reduction

Example

```
Pair = λY::*. {Y,Y};
List = λY::*. (μX. <nil:Unit,cons:{Y,X}>);
```

Determine that `List(List(Pair(Nat)))` and `List(List({Nat,Nat}))` are equivalent.

Parallel Reduction

$$\texttt{List}(\texttt{List}(\texttt{Pair}(\texttt{Nat}))) \Rrightarrow^* \mu\texttt{X}.\,\texttt{<nil:Unit}, \texttt{cons:}\{\mu\texttt{Y}.\,\texttt{<nil:Unit}, \texttt{cons:}\{\{\texttt{Nat}, \texttt{Nat}\}, \texttt{Y}\}\texttt{>}, \texttt{X}\}\texttt{>}$$

$$\texttt{List}(\texttt{List}(\{\texttt{Nat}, \texttt{Nat}\})) \Rrightarrow^* \mu\texttt{X}.\,\texttt{<nil:Unit}, \texttt{cons:}\{\mu\texttt{Y}.\,\texttt{<nil:Unit}, \texttt{cons:}\{\{\texttt{Nat}, \texttt{Nat}\}, \texttt{Y}\}\texttt{>}, \texttt{X}\}\texttt{>}$$

Parallel Reduction vs. Weak-Head Reduction

Example

```
Pair = λY::*. {Y,Y};
List = λY::*. (μX. <nil:Unit,cons:{Y,X}>);
```

Determine that `List(List(Pair(Nat)))` and `List(List({Nat,Nat}))` are equivalent.

Weak-Head Reduction

We start with $\varnothing \vdash \texttt{List}(\texttt{List}(\texttt{Pair}(\texttt{Nat}))) \Leftrightarrow \texttt{List}(\texttt{List}(\{\texttt{Nat},\texttt{Nat}\})) :: *$.

$$\texttt{List}(\texttt{List}(\texttt{Pair}(\texttt{Nat}))) \Downarrow \mu\texttt{X}.\,\texttt{<nil:Unit},\texttt{cons:}\{\texttt{List}(\texttt{Pair}(\texttt{Nat})),\texttt{X}\}\texttt{>}$$
$$\texttt{List}(\texttt{List}(\{\texttt{Nat},\texttt{Nat}\})) \Downarrow \mu\texttt{X}.\,\texttt{<nil:Unit},\texttt{cons:}\{\texttt{List}(\{\texttt{Nat},\texttt{Nat}\}),\texttt{X}\}\texttt{>}$$

By structural equivalence, we resort to check $\varnothing \vdash \texttt{Pair}(\texttt{Nat}) \Leftrightarrow \{\texttt{Nat},\texttt{Nat}\} :: *$.

$$\texttt{Pair}(\texttt{Nat}) \Downarrow \{\texttt{Nat},\texttt{Nat}\}$$
$$\{\texttt{Nat},\texttt{Nat}\} \Downarrow \{\texttt{Nat},\texttt{Nat}\}$$

System F_ω: The Combination of System F and λ_ω

Syntax

$$
\begin{aligned}
t &::= x \mid \lambda x{:}T.\, t \mid t\ t \mid \lambda X{::}K.\, t \mid t\ [T] \mid \{{}^*T, t\} \text{ as } T \mid \texttt{let}\ \{X, x\} = t\ \texttt{in}\ t \\
v &::= \lambda x{:}T.\, t \mid \lambda X{::}K.\, t \mid \{{}^*T, v\} \text{ as } T \\
T &::= X \mid \lambda X{::}K.\, T \mid T\ T \mid T \to T \mid \forall X{::}K.\, T \mid \{\exists X{::}K, T\} \\
\Gamma &::= \varnothing \mid \Gamma, x : T \mid \Gamma, X :: K \\
K &::= * \mid K \Rightarrow K
\end{aligned}
$$

Observation

- The universal type $\forall X.\, T$ becomes $\forall X{::}K.\, T$, i.e., we can abstract terms out of **type operators**.
- The existential type $\{\exists X, T\}$ becomes $\{\exists X{::}K, T\}$, i.e., we can pack a term to hide some **type operator**.

Typing, Kinding, and Type Equivalence

Typing

$$\frac{\Gamma, X :: K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{::}K_1 . t_2 : \forall X{::}K_1 . T_2}$$

$$\frac{\Gamma \vdash t_1 : \forall X{::}K_{11} . T_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}$$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \qquad \Gamma \vdash U :: K_1}{\Gamma \vdash \{{*}U, t_2\} \text{ as } \{\exists X{::}K_1, T_2\} : \{\exists X{::}K_1, T_2\}}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X{::}K_{11}, T_{12}\} \qquad \Gamma, X{::}K_{11}, x : T_{12} \vdash t_2 : T_2 \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2}$$

Kinding and Type Equivalence

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{::}K_1 . T_2 :: *}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \{\exists X{::}K_1, T_2\} :: *}$$

$$\frac{S_2 \equiv T_2}{\forall X{::}K_1 . S_2 \equiv \forall X{::}K_1 . T_2}$$

$$\frac{S_2 \equiv T_2}{\{\exists X{::}K_1, S_2\} \equiv \{\exists X{::}K_1, T_2\}}$$

Review: Abstract Data Types (ADTs)

Definition

An abstract data type (ADT) consists of

- a type name `A`,
- a concrete representation type `T`,
- implementations of operations for manipulating values of type `T`, and
- an **abstraction boundary** enclosing the representation and operations.

```
counterADT =
   {*Nat, {new = 1,
           get = λi:Nat. i,
           inc = λi:Nat. succ(i)}}
 as {∃Counter,
      {new: Counter, get: Counter→Nat, inc: Counter→Counter}};
▶ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

Abstract Type Operators

Question

We want to implement an ADT of pairs.

- The ADT provides operations for building pairs and taking them apart.
- Those operations need to be **polymorphic**.

The abstract type `Pair` would not be a proper type, but an **abstract type operator**!

```
PairSig = {∃Pair :: *⇒*⇒*,
           {pair: ∀X. ∀Y. X→Y→(Pair X Y),
            fst : ∀X. ∀Y. (Pair X Y)→X,
            snd : ∀X. ∀Y. (Pair X Y)→Y}};
```

Abstract Type Operators

Example

```
pairADT = {*(λX. λY. ∀R. (X→Y→R) → R),
            {pair = λX. λY. λx:X. λy:Y. λR. λp:(X→Y→R). p x y,
             fst  = λX. λY. λp:(∀R. (X→Y→R) → R). p [X] (λx:X. λy:Y. x),
             snd  = λX. λY. λp:(∀R. (X→Y→R) → R). p [Y] (λx:X. λy:Y. y)}}
           as PairSig;
▶ pairADT : PairSig

let {Pair,pair} = pairADT
in pair.fst [Nat] [Bool] (pair.pair [Nat] [Bool] 5 true);
▶ 5 : Nat
```

More Examples

Option: Combination with Variants

```
Option = λX. <none:Unit,some:X>;
none = λX. <none=unit> as (Option X);
▶ none : ∀X. (Option X)
some = λX. λx:X. <some=x> as (Option X);
▶ some : ∀X. X → (Option X)
```

List: Combination with Variants, Tuples, and Recursive Types

```
List = μL::(*⇒*). λX. <nil:Unit,cons:{X,(L X)}>;
nil = λX. <nil=unit> as (List X);
▶ nil : ∀X. (List X)
cons = λX. λh:X. λt:(List X). <cons={h,t}> as (List X);
▶ cons : ∀X. X → (List X) → (List X)
```

More Examples

Queue: Implementing a Queue using Two Lists

```
QueueSig = {∃Q :: *⇒*,
            {empty : ∀X. (Q X),
             insert: ∀X. X → (Q X) → (Q X),
             remove: ∀X. (Q X) → Option {X,(Q X)}}};
queueADT = {*(λX. {List X,List X}),
            {empty  = λX. {nil [X],nil [X]},
             insert = λX. λa:X. λq:{List X,List X}. {(cons [X] a q.1),q.2},
             remove =
              λX. λq:{List X,List X}.
               let q' = case q.2 of <nil=u> ⇒ {nil [X], reverse [X] q.1}
                                  | <cons={h,t}> ⇒ q
               in case q'.2 of
                 <nil=u> ⇒ none [{X,{List X,List X}}]
               | <cons={h,t}> ⇒ some [{X,{List X,List X}}] {h,{q'.1,t}}}} as QueueSig;
▶ queueADT : QueueSig
```

Preservation

Observation

The structural rule (T-Eq) makes induction proof difficult:

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

Preservation of Shapes (for Arrows)

If $S_1 \rightarrow S_2 \Rrightarrow^* T$, then $T = T_1 \rightarrow T_2$ with $S_1 \Rrightarrow^* T_1$ and $S_2 \Rrightarrow^* T_2$.

Inversion (for Arrows)

If $\Gamma \vdash \lambda x{:}S_1 . s_2 : T_1 \rightarrow T_2$, then $T_1 \equiv S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$. Also $\Gamma \vdash S_1 :: *$.

Theorem (30.3.14)

If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

Progress

Canonical Forms (for Arrows)

If t is a closed value with $\varnothing \vdash t : T_1 \to T_2$, then t is an abstraction.

Theorem (30.3.16)

Suppose t is a closed, well-typed term (that is, $\varnothing \vdash t : T$ for some T).
Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Kinding

Remark

Recall that we observed that if $\varnothing \vdash t : T$, then $\varnothing \vdash T :: *$.

Context Formation

$$\frac{}{\varnothing \text{ ctx}} \qquad \frac{\Gamma \text{ ctx} \qquad \Gamma \vdash T :: *}{\Gamma, x : T \text{ ctx}} \qquad \frac{\Gamma \text{ ctx}}{\Gamma, X :: K \text{ ctx}}$$

Theorem

If Γ ctx and $\Gamma \vdash t : T$, then $\Gamma \vdash T :: *$.

Fragments of System F_ω

Definition

In System F_1, the only kind is $*$ and no quantification ($\forall$) or abstraction (λ) over types is permitted.
The remaining systems are defined with reference to a hierarchy of kinds at **level** i:

$$\begin{aligned}
\mathcal{K}_1 &= \varnothing \\
\mathcal{K}_{i+1} &= \{*\} \cup \{J \Rightarrow K \mid J \in \mathcal{K}_i \land K \in \mathcal{K}_{i+1}\} \\
\mathcal{K}_\omega &= \bigcup_{1 \leqslant i} \mathcal{K}_i
\end{aligned}$$

Example

- System F_1 is the simply-typed lambda-calculus $\lambda_\rightarrow$.
- In System F_2, we have $\mathcal{K}_2 = \{*\}$, so there is no lambda-abstraction at the type level but we allow quantification over proper types.
  - F_2 is just the System F; this is why System F is also called the **second-order lambda-calculus**.
- For System F_3, we have $\mathcal{K}_3 = \{*, * \Rightarrow *, * \Rightarrow * \Rightarrow *, \ldots\}$, i.e., type-level abstractions are over proper types.

Type-Level Natural Numbers

Remark

The kinding system of λ_ω and F_ω consists of only $*$ and $K_1 \Rightarrow K_2$.
Can we extend kinding to support more versatile type-level computation?

Observation

We can extend type-level computation as long as **type equivalence remains decidable**.

Natural-Number Kind

$$K ::= * \mid K \Rightarrow K \mid \mathbb{N}$$

$$T ::= X \mid \lambda X{::}K.\, T \mid T\ T \mid T \to T \mid \forall X{::}K.\, T \mid \{\exists X{::}K, T\} \mid \mathtt{ZERO} \mid \mathtt{SUCC}\ T \mid \ldots$$

With recursive types, we can define length-indexed lists:

```
List = λX. μL::(ℕ⇒*). λM::ℕ. IF ISZERO(M) THEN Unit ELSE {X,(L (PRED M))};
▶ List :: * ⇒ ℕ ⇒ *
```

Type-Level Natural Numbers

Example

```
List = λX. μL::(ℕ⇒*). λM::ℕ. IF ISZERO(M) THEN Unit ELSE {X,(L (PRED M))};
▶ List :: * ⇒ ℕ ⇒ *

nil = λX. unit as (List X ZERO);
▶ nil : ∀X. (List X ZERO)
cons = λX. λM::ℕ. λh:X. λt:(List X M). {h,t} as (List X (SUCC M));
▶ cons : ∀X. ∀M::ℕ. X → (List X M) → (List X (SUCC M))
```

Example

```
PLUS = μP::(ℕ⇒ℕ⇒ℕ). λM::ℕ. λN::ℕ. IF ISZERO(M) THEN N ELSE SUCC (P (PRED M) N);
▶ PLUS :: ℕ ⇒ ℕ ⇒ ℕ
```

Type-Level Natural Numbers

Natural-Number Kind

Type-level recursion would render type equivalence **undecidable**.
Let us consider $\mathbb{N}$ as an **inductively-defined** kind.

$$\mathsf{T} ::= \mathsf{X} \mid \lambda\mathsf{X}{::}\mathsf{K}.\,\mathsf{T} \mid \mathsf{T}\;\mathsf{T} \mid \mathsf{T} \to \mathsf{T} \mid \forall\mathsf{X}{::}\mathsf{K}.\,\mathsf{T} \mid \{\exists\mathsf{X}{::}\mathsf{K},\mathsf{T}\} \mid \mathtt{ZERO} \mid \mathtt{SUCC}\;\mathsf{T} \mid \mathtt{ITER}\;\mathsf{T}\;\mathtt{WITH}\;\mathtt{ZERO} \Rightarrow \mathsf{T} \mid \mathtt{SUCC} \Rightarrow \mathsf{Y}.\,\mathsf{T}$$

Below are the kinding rules for $\mathbb{N}$:

$$\frac{}{\Gamma \vdash \mathtt{ZERO} :: \mathbb{N}} \qquad \frac{\Gamma \vdash \mathsf{T}_1 :: \mathbb{N}}{\Gamma \vdash \mathtt{SUCC}\;\mathsf{T}_1 :: \mathbb{N}} \qquad \frac{\Gamma \vdash \mathsf{T}_0 :: \mathbb{N} \qquad \Gamma \vdash \mathsf{T}_1 :: \mathsf{K} \qquad \Gamma, \mathsf{Y} :: \mathsf{K} \vdash \mathsf{T}_2 :: \mathsf{K}}{\Gamma \vdash \mathtt{ITER}\;\mathsf{T}_0\;\mathtt{WITH}\;\mathtt{ZERO} \Rightarrow \mathsf{T}_1 \mid \mathtt{SUCC} \Rightarrow \mathsf{Y}.\,\mathsf{T}_2 :: \mathsf{K}}$$

Example

```
List = λX. λM::ℕ. ITER M OF ZERO ⇒ Unit | SUCC ⇒ Y. {X,Y};
▶ List :: * ⇒ ℕ ⇒ *
PLUS = λM::ℕ. λN::ℕ. ITER M OF ZERO ⇒ N | SUCC ⇒ Y. SUCC Y;
▶ PLUS :: ℕ ⇒ ℕ ⇒ ℕ
```

Type-Level Natural Numbers

Term-Level Case on Type-Level Natural Numbers

$$\frac{\Gamma \vdash T_0 :: \mathbb{N} \qquad \Gamma, T_0 \equiv \mathtt{ZERO} :: \mathbb{N} \vdash t_1 : T \qquad \Gamma, Y :: \mathbb{N}, T_0 \equiv \mathtt{SUCC}\ Y :: \mathbb{N} \vdash t_2 : T \qquad \Gamma \vdash T :: *}{\Gamma \vdash \mathtt{tcase}\ T_0\ \mathtt{of}\ \mathtt{ZERO} \Rightarrow t_1 \mid \mathtt{SUCC}\ Y \Rightarrow t_2 : T}$$

Example

```
List = λX. λM::ℕ. ITER M OF ZERO ⇒ Unit | SUCC ⇒ Y. {X,Y};
▶ List :: * ⇒ ℕ ⇒ *
PLUS = λM::ℕ. λN::ℕ. ITER M OF ZERO ⇒ N | SUCC ⇒ Y. SUCC Y;
▶ PLUS :: ℕ ⇒ ℕ ⇒ ℕ

▶ append : ∀X. ∀M::ℕ. ∀N::ℕ. (List X M) → (List X N) → (List X (PLUS M N))
append = λX. fix λf. λM::ℕ. λN::ℕ. λl1:(List X M). λl2:(List X N).
           tcase M of ZERO ⇒ let unit  = l1 in l2 as (List X (PLUS M N))
                      SUCC M' ⇒ let {h,t} = l1 in {h,(f M' N t l2)} as (List X (PLUS M N));
```

Type-Level Natural Numbers

Remark

Because type-equivalence constraints can appear in the context, we need **hypothetical** type equivalence.
Ref: J. Cheney and R. Hinze. 2003. First-Class Phantom Types. Technical report. Cornell University.

Hypothetical Type Equivalence: $\Gamma \vdash S \equiv T :: K$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K} \qquad \frac{\Gamma \vdash T \equiv S :: K}{\Gamma \vdash S \equiv T :: K} \qquad \frac{\Gamma \vdash S \equiv U :: K \qquad \Gamma \vdash U \equiv T :: K}{\Gamma \vdash S \equiv T :: K}$$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \qquad \Gamma \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash S_1 \to S_2 \equiv T_1 \to T_2 :: *} \qquad \frac{\Gamma, X :: K_1 \vdash S_2 \equiv T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1 . S_2 \equiv \lambda X{::}K_1 . T_2 :: K_1 \Rightarrow K_2}$$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash S_2 \equiv T_2 :: K_{11}}{\Gamma \vdash S_1\ S_2 \equiv T_1\ T_2 :: K_{12}} \qquad \frac{\Gamma, X :: K_{11} \vdash T_{12} :: K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash (\lambda X{::}K_{11} . T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} :: K_{12}}$$

Type-Level Natural Numbers

Hypothetical Type Equivalence: $\Gamma \vdash S \equiv T :: K$

$$\frac{}{\Gamma \vdash \mathtt{ZERO} \equiv \mathtt{ZERO} :: \mathbb{N}} \qquad \frac{\Gamma \vdash S_1 \equiv T_1 :: \mathbb{N}}{\Gamma \vdash \mathtt{SUCC}\ S_1 \equiv \mathtt{SUCC}\ T_1 :: \mathbb{N}}$$

$$\frac{\Gamma \vdash S_0 \equiv T_0 :: \mathbb{N} \qquad \Gamma \vdash S_1 \equiv T_1 :: K \qquad \Gamma, Y :: K \vdash S_2 \equiv T_2 :: K}{\Gamma \vdash \mathtt{ITER}\ S_0\ \mathtt{WITH}\ \mathtt{ZERO} \Rightarrow S_1 \mid \mathtt{SUCC} \Rightarrow Y.\, S_2 \equiv \mathtt{ITER}\ T_0\ \mathtt{WITH}\ \mathtt{ZERO} \Rightarrow T_1 \mid \mathtt{SUCC} \Rightarrow Y.\, T_2 :: K}$$

$$\frac{\Gamma \vdash T_1 :: K \qquad \Gamma, Y :: K \vdash T_2 :: K}{\Gamma \vdash \mathtt{ITER}\ \mathtt{ZERO}\ \mathtt{WITH}\ \mathtt{ZERO} \Rightarrow T_1 \mid \mathtt{SUCC} \Rightarrow Y.\, T_2 \equiv T_1 :: K}$$

$$\frac{\Gamma \vdash T_0 :: \mathbb{N} \qquad \Gamma \vdash T_1 :: K \qquad \Gamma, Y :: K \vdash T_2 :: K}{\begin{array}{c}\Gamma \vdash \mathtt{ITER}\ (\mathtt{SUCC}\ T_0)\ \mathtt{WITH}\ \mathtt{ZERO} \Rightarrow T_1 \mid \mathtt{SUCC} \Rightarrow Y.\, T_2 \\ \equiv \\ [Y \mapsto \mathtt{ITER}\ T_0\ \mathtt{WITH}\ \mathtt{ZERO} \Rightarrow T_1 \mid \mathtt{SUCC} \Rightarrow Y.\, T_2]T_2 :: K\end{array}}$$

Type-Level Natural Numbers

Hypothetical Type Equivalence: $\Gamma \vdash S \equiv T :: K$

$$\frac{S \equiv T :: \mathbb{N} \in \Gamma}{\Gamma \vdash S \equiv T :: \mathbb{N}} \qquad \frac{\Gamma \vdash \mathtt{SUCC}\ S_1 \equiv \mathtt{SUCC}\ T_1 :: \mathbb{N}}{\Gamma \vdash S_1 \equiv T_1 :: \mathbb{N}}$$

Example

$$\mathtt{append} \equiv \lambda X.\ \mathbf{fix}\ \lambda f{:}\_.\ \lambda M{::}\mathbb{N}.\ \lambda N{::}\mathbb{N}.\ \lambda l_1{:}(\mathtt{List}\ X\ M).\ \lambda l_2{:}(\mathtt{List}\ X\ N).$$
$$\mathtt{tcase}\ M\ \mathtt{of}\ \mathtt{ZERO} \Rightarrow \mathtt{t1} \mid \mathtt{SUCC}\ M' \Rightarrow \mathtt{t2}$$
$$\mathtt{t1} \equiv \mathtt{let}\ \mathtt{unit} = l_1\ \mathtt{in}\ l_2\ \mathtt{as}\ (\mathtt{List}\ X\ (\mathtt{PLUS}\ M\ N))$$
$$\mathtt{t2} \equiv \mathtt{let}\ \{h, t\} = l_1\ \mathtt{in}\ \{h, (f\ M'\ N\ t\ l_2)\}\ \mathtt{as}\ (\mathtt{List}\ X\ (\mathtt{PLUS}\ M\ N))$$

Let $T_{app} \equiv \forall X{::}*.\ \forall M{::}\mathtt{Nat}.\ \forall N{::}\mathtt{Nat}.\ (\mathtt{List}\ X\ M) \to (\mathtt{List}\ X\ N) \to (\mathtt{List}\ X\ (\mathtt{PLUS}\ M\ N))$. We need to check

$$X :: *, f : T_{app}, M :: \mathbb{N}, N :: \mathbb{N}, l_1 : \mathtt{List}\ X\ M, l_2 : \mathtt{List}\ X\ N, M \equiv \mathtt{ZERO} :: \mathbb{N} \vdash \mathtt{t1} : \mathtt{List}\ X\ (\mathtt{PLUS}\ M\ N)$$
$$X{::}*, f{:}T_{app}, M{::}\mathbb{N}, N{::}\mathbb{N}, l_1{:}\mathtt{List}\ X\ M, l_2{:}\mathtt{List}\ X\ N, M'{::}\mathbb{N}, M \equiv \mathtt{SUCC}\ M' :: \mathbb{N} \vdash \mathtt{t2} : \mathtt{List}\ X\ (\mathtt{PLUS}\ M\ N)$$

Indexed Types

Observation

Previously, to support type-level natural numbers, we enriched the type level with natural-number operations.

- This approach complicates type-equivalence checking.
- This approach cannot make use of automatic solvers for natural-number reasoning.

Principle

We can separate natural numbers from the type level to reside in **its own index level**.

$$S ::= \{a :: \mathbb{N} \mid \theta\} \mid \{\theta\}$$

$$I ::= a \mid n \mid I + I \mid I \times I \mid \ldots$$

$$\theta ::= \top \mid \bot \mid \neg\theta \mid \theta \wedge \theta \mid \theta \vee \theta \mid I = I \mid I \leqslant I \mid \ldots$$

$$K ::= * \mid K \Rightarrow K \mid \mathbb{N} \Rightarrow K$$

$$T ::= X \mid \lambda X{::}K.\,T \mid T\;T \mid T \to T \mid \forall X{::}K.\,T \mid \{\exists X{::}K, T\} \mid \lambda a{::}\mathbb{N}.\,T \mid T\;I \mid \forall S.\,T \mid \{\exists S, T\}$$

Length-indexed lists: λ `X.` μ`L` :: `(`$\mathbb{N}\Rightarrow$`*).` λ `M`::$\mathbb{N}$`.` `{`$\exists$`{M=0},Unit} + {`$\exists$`{M'`::$\mathbb{N}$`|M=M'+1},{X,(L M')}}`.

Indexed Types

Remark

The kind $\{a : \mathbb{N} \mid \theta\}$ is usually called a **refinement** kind.
Ref: H. Xi and F. Pfenning. 1999. Dependent Types in Practical Programming. In *Princ. of Prog. Lang.* (POPL'99). doi: 10.1145/292540.292560.

Index Checking

$$\frac{\Gamma \vdash t : \forall\{a :: \mathbb{N} \mid \theta\}.\, T \qquad \Gamma \vdash i :: \{a :: \mathbb{N} \mid \theta\}}{\Gamma \vdash t\,[i] : [a \mapsto i]T} \qquad \frac{\Gamma \vdash t : \forall\{\theta\}.\, T \qquad \Gamma \vdash \texttt{@} :: \{\theta\}}{\Gamma \vdash t\,[\texttt{@}] : T}$$

$$\frac{\Gamma \models [a \mapsto i]\theta}{\Gamma \vdash i :: \{a :: \mathbb{N} \mid \theta\}} \qquad \frac{\Gamma \models \theta}{\Gamma \vdash \texttt{@} :: \{\theta\}}$$

Constraint Checking

For example, consider $\{a :: \mathbb{N} \mid a \geqslant 5\}, x : (\texttt{List Nat}\ a) \models \neg(a = 0)$.
We can resort to check validity of the formula in first-order logic: $\forall a : \mathbb{N}.(a \geqslant 5) \implies \neg(a = 0)$.

Extensible Records

Remark

In Chap. 11, we studied records, i.e., named tuples, which are not **extensible**.

Extensible Records

- **Extension**: We can extend a record r with label ℓ and term t by $\{\ell = t \mid r\}$.

```
origin = {x = 0 | {y = 0 | {}}};
origin3 = {z = 0 | origin};
named = λs. λr. {name = s | r};
```

- **Selection**: The selection operation $r.\ell$ selects the value of a label ℓ from a record r.

```
distance = λp. sqrt ((p.x * p.x) + (p.y * p.y));
distance (named "2d" origin) + distance origin3;
```

- **Restriction**: The restriction operation $r - \ell$ removes a label ℓ from a record r.

```
update_name = λr. λs. {name = s | r - name };
rename_name_nn = λr. {nn = r.name | r - name };
```

Scoped Labels

Observation

Typing extensible records needs to ensure the **safety** of the operations.

- Selection $r.\ell$ and restriction $r - \ell$ requires the label ℓ to be **present** in r.
- Usually, extension $\{\ell = t \mid r\}$ requires the label ℓ to be **absent** in r.

Scoped Labels

Let us consider **ordered** and **scoped** labels in records, which allow **duplicated** labels.
Ref: D. Leijen. 2005. Extensible records with scoped labels. *In Symp. on Trends in Functional Programming* (TFP'05), 297–312.

```
p = {x=2, x=true};
▶ p : {x:Nat, x:Bool}
p.x;
▶ 2 : Nat
(p - x).x;
▶ true : Bool
```

Type-Level Rows

Principle

A **row** is a list of labeled types, which can be manipulated at the type level.

$$K ::= * \mid K \Rightarrow K \mid \mathsf{row}$$

$$T ::= X \mid \lambda X{::}K.\, T \mid T\ T \mid T \to T \mid \forall X{::}K.\, T \mid \{\exists X{::}K, T\} \mid (\!|\,|\!) \mid (\!|\ell : T \mid T|\!) \mid \{T\}$$

For example, the record type $\{\mathtt{x} : \mathtt{Nat}, \mathtt{y} : \mathtt{Nat}\}$ is encoded as $\{(\!|\mathtt{x} : \mathtt{Nat} \mid (\!|\mathtt{y} : \mathtt{Nat} \mid (\!|\,|\!)|\!)|\!)\}$.
Below are the kinding rules for row:

$$\frac{}{\Gamma \vdash (\!|\,|\!) :: \mathsf{row}} \qquad \frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: \mathsf{row}}{\Gamma \vdash (\!|\ell : T_1 \mid T_2|\!) :: \mathsf{row}} \qquad \frac{\Gamma \vdash T :: \mathsf{row}}{\Gamma \vdash \{T\} :: *}$$

Well-Typed Record Operations

$$\{\ell = \_ \mid \_\} : \forall R{::}\mathsf{row}.\, \forall X{::}*.\, X \to \{R\} \to \{(\!|\ell : X \mid R|\!)\}$$

$$(\_.\ell) : \forall R{::}\mathsf{row}.\, \forall X{::}*.\, \{(\!|\ell : X \mid R|\!)\} \to X$$

$$(\_ - \ell) : \forall R{::}\mathsf{row}.\, \forall X{::}*.\, \{(\!|\ell : X \mid R|\!)\} \to \{R\}$$

Row Equivalence

Question

The type $\forall R{::}\mathsf{row}.\ \forall X{::}*.\ \{(\!|\ell : X \mid R|\!)\} \to X$ of the selection operation requires ℓ to be the **first** label.
How to relax this requirement?

Type-Level Row Equivalence

$$\frac{}{(\!|\,|\!) \equiv (\!|\,|\!)} \qquad \frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{(\!|\ell : S_1 \mid S_2|\!) \equiv (\!|\ell : T_1 \mid T_2|\!)} \qquad \frac{\ell \neq \ell'}{(\!|\ell : T_1 \mid (\!|\ell' : T_2 \mid T_3|\!)|\!) \equiv (\!|\ell' : T_2 \mid (\!|\ell : T_1 \mid T_3|\!)|\!)}$$

Example

$$\dfrac{\dfrac{\dfrac{\vdots}{\varnothing \vdash \{x = 0 \mid \{y = \mathsf{true} \mid \{\}\}\} : \{(\!|x : \mathsf{Nat} \mid (\!|y : \mathsf{Bool} \mid (\!|\,|\!)|\!)|\!)\}} \qquad \dfrac{x \neq y}{\{(\!|x : \mathsf{Nat} \mid (\!|y : \mathsf{Bool} \mid (\!|\,|\!)|\!)|\!)\} \equiv \{(\!|y : \mathsf{Bool} \mid (\!|x : \mathsf{Nat} \mid (\!|\,|\!)|\!)|\!)\}}}{\varnothing \vdash \{x = 0 \mid \{y = \mathsf{true} \mid \{\}\}\} : \{(\!|y : \mathsf{Bool} \mid (\!|x : \mathsf{Nat} \mid (\!|\,|\!)|\!)|\!)\}}}{\varnothing \vdash \{x = 0 \mid \{y = \mathsf{true} \mid \{\}\}\}.y : \mathsf{Bool}}$$

Use Rows for Extensible Variants

Principle

Records model labeled tuples. Variants model a labeled choice among values.

$$\mathsf{T} ::= \mathsf{X} \mid \lambda\mathsf{X}{::}\mathsf{K}.\,\mathsf{T} \mid \mathsf{T}\ \mathsf{T} \mid \mathsf{T} \to \mathsf{T} \mid \forall\mathsf{X}{::}\mathsf{K}.\,\mathsf{T} \mid \{\exists\mathsf{X}{::}\mathsf{K}, \mathsf{T}\} \mid (\!|\,|\!) \mid (\!|\ell : \mathsf{T} \mid \mathsf{T}|\!) \mid \{\mathsf{T}\} \mid {\color{red}<\mathsf{T}>}$$

For example, the variant type $<\mathtt{none} : \mathtt{Unit}, \mathtt{some} : \mathtt{Nat}>$ is encoded as $<(\!|\mathtt{none} : \mathtt{Unit} \mid (\!|\mathtt{some} : \mathtt{Nat} \mid (\!|\,|\!)|\!)|\!)>$.

Well-Typed Variant Operations

- **Injection**: We write $<\ell = t>$ to build a variant with label ℓ and term t.
$$<\ell = \_> : \forall \mathsf{R}{::}\mathsf{row}.\ \forall \mathsf{X}{::}*.\ \mathsf{X} \to <(\!|\ell : \mathsf{X} \mid \mathsf{R}|\!)>$$
- **Embedding**: We write $<\ell \mid v>$ to embed a variant v in a type that also allows label ℓ.
$$<\ell \mid \_> : \forall \mathsf{R}{::}\mathsf{row}.\ \forall \mathsf{X}{::}*.\ <\mathsf{R}> \to <(\!|\ell : \mathsf{X} \mid \mathsf{R}|\!)>$$
- **Decomposition**: We write $\ell \in v\ ?\ t_1\ :\ t_2$ to decompose a variant v and check if it is labeled with ℓ.
$$(\ell \in \_\ ?\ \_ : \_) : \forall \mathsf{R}{::}\mathsf{row}.\ \forall \mathsf{X}{::}*.\ \forall \mathsf{Y}{::}*.\ <(\!|\ell : \mathsf{X} \mid \mathsf{R}|\!)> \to (\mathsf{X} \to \mathsf{Y}) \to (<\mathsf{R}> \to \mathsf{Y}) \to \mathsf{Y}$$

Type-Level Labels

Question

Can we also introduce a kind for **labels**?

Principle

$$K ::= * \mid K \Rightarrow K \mid \mathsf{row} \mid \mathsf{label}$$

$$T ::= X \mid \lambda X{::}K.\,T \mid T\ T \mid T \to T \mid \forall X{::}K.\,T \mid \{\exists X{::}K, T\} \mid (\!|\,|\!) \mid (\!|T : T \mid T|\!) \mid \{T\} \mid \text{<}T\text{>} \mid \#\ell$$

$$\frac{}{\Gamma \vdash \#\ell :: \mathsf{label}} \qquad \frac{\Gamma \vdash T_1 :: \mathsf{label} \qquad \Gamma \vdash T_2 :: * \qquad \Gamma \vdash T_3 :: \mathsf{row}}{\Gamma \vdash (\!|T_1 : T_2 \mid T_3|\!) :: \mathsf{row}}$$

Type-Level Record Computation

Question

Can we support non-trivial type-level record computation?

Principle

Ref: A. Chlipala. 2010. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In *Prog. Lang. Design and Impl.* (PLDI'10), 122–133. doi: 10.1145/1806596.1806612.

$$\mathsf{T} ::= \mathsf{X} \mid \lambda \mathsf{X}{::}\mathsf{K}.\, \mathsf{T} \mid \mathsf{T}\ \mathsf{T} \mid \mathsf{T} \to \mathsf{T} \mid \forall \mathsf{X}{::}\mathsf{K}.\, \mathsf{T} \mid \{\exists \mathsf{X}{::}\mathsf{K}, \mathsf{T}\} \mid (\!|\,|\!) \mid (\!|\mathsf{T} : \mathsf{T} \mid \mathsf{T}|\!) \mid \{\mathsf{T}\} \mid \texttt{<}\mathsf{T}\texttt{>} \mid \#\ell \mid \texttt{map}$$

$$\frac{}{\Gamma \vdash \texttt{map} :: (* \Rightarrow *) \Rightarrow \textsf{row} \Rightarrow \textsf{row}}$$

Example

Consider `Meta` $=$ λ `T`. $\{(\!|$ `#name:String, #show:(T→String)` $|\!)\}$.
Then `map Meta` $(\!|$ `#x:Nat, #y:Bool` $|\!)$ is equivalent to $(\!|$ `#x:(Meta Nat), #y:(Meta Bool)` $|\!)$.

Example: A Generic Table Formatter

```
Meta = λT. {⦇ #name:String, #show:(T→String) ⦈};
▶ Meta :: * ⇒ *

Folder = λR::row. ∀TF::(row⇒*).
              (∀L::label. ∀T. ∀R::row. TF R → TF ⦇ L : T | R ⦈) → TF ⦇⦈ → TF R;
▶ Folder :: row ⇒ *

▶ mk_table : ∀R::row. Folder R → { map Meta R } → { R } → String
mk_table = λR::row. λfl:(Folder R). λmr:{map Meta R}. λx:{R}.
      fl (λR::row. {map Meta R} → {R} → String)
         (λL::label. λT. λR::row.
            λacc:({map Meta R}→{R}→String).
            λmr:{map Meta ⦇ L : T | R ⦈}.
            λx:{⦇ L : T | R ⦈}.
"<tr><th>" ^ mr.L.name ^ "</th><td>" ^ mr.L.show x.L ^ "</td></tr>" ^ acc (mr-L) (x-L))
         (λ_:{map Meta ⦇⦈}. λ_:{⦇⦈}. "") mr x
```

The Essence of λ: Characterization

Principle

Types characterize **terms**. **Kinds** characterize **types**.

Question

Can we have more than **three** levels of expressions?

Aside (Pure Type Systems, Part I)

Let S be a set of **sorts**, e.g., $S = \{*, \square\}$ where

- $*$ represents the sort of **all (proper) types** and
- $\square$ represents the sort of **all kinds**.

Let M be a set of **axioms**, e.g., $M = \{(\varnothing \vdash * : \square)\}$, meaning "$*$ is a kind for (proper) types."

One can definitely add more sorts to S and more axioms to M accordingly!

The Essence of λ: Abstraction

Principle

- In $\lambda_{\rightarrow}$, we use $\lambda x{:}T.\ t$ to abstract **terms** out of **terms**.
- In λ_{ω}, we use $\lambda X{::}K.\ T$ to abstract **types** out of **types**.

Aside (Pure Type Systems, Part II)

Let S be a set of **sorts**, e.g., $S = \{*, \Box\}$. Let M be a set of **axioms**, e.g., $M = \{(\varnothing \vdash * : \Box)\}$.

Let $R \subseteq S \times S$ be a set of **rules**: for each $(s_1, s_2) \in R$, we have

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma \vdash B : s_2}{\Gamma \vdash A \rightsquigarrow^{s_1}_{s_2} B : s_2}\ \text{Arrow} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \rightsquigarrow^{s_1}_{s_2} B : s_2}{\Gamma \vdash \lambda x{:}A.\ b : A \rightsquigarrow^{s_1}_{s_2} B}\ \text{Abs}$$

$$\frac{\Gamma \vdash F : A \rightsquigarrow^{s_1}_{s_2} B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B}\ \text{App}$$

Let $R \subseteq S \times S$ be a set of **rules**: for each $(s_1, s_2) \in R$, we have

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma \vdash B : s_2}{\Gamma \vdash A \rightsquigarrow_{s_2}^{s_1} B : s_2}\ \text{Arrow} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \rightsquigarrow_{s_2}^{s_1} B : s_2}{\Gamma \vdash \lambda x{:}A.\, b : A \rightsquigarrow_{s_2}^{s_1} B}\ \text{Abs}$$

$$\frac{\Gamma \vdash F : A \rightsquigarrow_{s_2}^{s_1} B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B}\ \text{App}$$

$\lambda_{\rightarrow}$: Abstracting Terms out of Terms

Let $R \stackrel{\text{def}}{=} \{(*, *)\}$. Then $\rightsquigarrow_*^*$ represents arrow types $\rightarrow$.

$$\frac{\Gamma \vdash T_1 : * \qquad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightsquigarrow_*^* T_2 : *}$$ means "if T_1, T_2 are types, then $T_1 \rightarrow T_2$ is a type"

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \qquad \Gamma \vdash T_1 \rightsquigarrow_*^* T_2 : *}{\Gamma \vdash \lambda x{:}T_1.\, t_2 : T_1 \rightsquigarrow_*^* T_2}$$ means the typing rule (T-Abs)

$$\frac{\Gamma \vdash t_1 : T_{11} \rightsquigarrow_*^* T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}}$$ means the typing rule (T-App)

Let $R \subseteq S \times S$ be a set of **rules**: for each $(s_1, s_2) \in R$, we have

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma \vdash B : s_2}{\Gamma \vdash A \leadsto_{s_2}^{s_1} B : s_2}\ \text{Arrow} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \leadsto_{s_2}^{s_1} B : s_2}{\Gamma \vdash \lambda x{:}A.\, b : A \leadsto_{s_2}^{s_1} B}\ \text{Abs}$$

$$\frac{\Gamma \vdash F : A \leadsto_{s_2}^{s_1} B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\,a : B}\ \text{App}$$

λ_ω: Abstracting Types out of Types

Let $R \stackrel{\text{def}}{=} \{(*, *), (\square, \square)\}$. Then $\leadsto_*^*$ represents arrow types $\to$ and $\leadsto_\square^\square$ represents arrow kinds $\Rightarrow$.

$$\frac{\Gamma \vdash K_1 : \square \qquad \Gamma \vdash K_2 : \square}{\Gamma \vdash K_1 \leadsto_\square^\square K_2 : \square}$$ means "if K_1, K_2 are kinds, then $K_1 \Rightarrow K_2$ is a kind"

$$\frac{\Gamma, X : K_1 \vdash T_2 : K_2 \qquad \Gamma \vdash K_1 \leadsto_\square^\square K_2 : \square}{\Gamma \vdash \lambda X{:}K_1.\, T_2 : K_1 \leadsto_\square^\square K_2}$$ means the typing rule (K-Abs)

$$\frac{\Gamma \vdash T_1 : K_{11} \leadsto_\square^\square K_{12} \qquad \Gamma \vdash T_2 : K_{11}}{\Gamma \vdash T_1\, T_2 : K_{12}}$$ means the typing rule (K-App)

The Essence of λ: Abstraction

Principle

In System F, we use $\lambda X.\ t$ to abstract **terms** out of **types**.

Observation

We can think of $\lambda X.\ t$ as $\lambda X{::}*.\ t$, i.e., a type abstraction should be applied to a proper type.
The type of $\lambda X{::}*.\ t$ then has the form $\forall X{::}*.\ T$—**not an arrow!**
$\forall X{::}*.\ T$ can be thought of as a **dependent arrow** $(X{::}*) \rightrightarrows T$: the domain is a **kind** and the range is a **type**.
In next chapter, we will see a generalized form $\forall X{::}K.\ T$, or as a dependent arrow $(X{::}K) \rightrightarrows T$.

Aside (Pure Type Systems, Part III)

Let $R \subseteq S \times S$ be a set of **rules**: for each $(s_1, s_2) \in R$, we have

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma \vdash B : s_2}{\Gamma \vdash A \leadsto^{s_1}_{s_2} B : s_2}\,\text{Arrow} \qquad \text{becomes} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x{:}A) \leadsto^{s_1}_{s_2} B : s_2}\,\text{Arrow}^{D}$$

Then $(X : *) \leadsto^{\square}_{*} T$ represents $\forall X{::}*.\ T$!

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \leadsto^{s_1}_{s_2} B : s_2}{\Gamma \vdash \lambda x{:}A.\, b : A \leadsto^{s_1}_{s_2} B}\text{Abs} \quad \text{becomes} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (x{:}A) \leadsto^{s_1}_{s_2} B : s_2}{\Gamma \vdash \lambda x{:}A.\, b : (x{:}A) \leadsto^{s_1}_{s_2} B}\text{Abs}^{\text{D}}$$

$$\frac{\Gamma \vdash F : A \leadsto^{s_1}_{s_2} B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B}\text{App} \quad \text{becomes} \quad \frac{\Gamma \vdash F : (x{:}A) \leadsto^{s_1}_{s_2} B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : [x \mapsto a]B}\text{App}^{\text{D}}$$

System F: Abstracting Terms out of Types

Let $R \overset{\text{def}}{=} \{(*, *), (\Box, *)\}$. Then $\leadsto^{*}_{*}$ represents arrow types $\to$ and $\leadsto^{\Box}_{*}$ represents universal types $\forall$.

$$\frac{\Gamma \vdash K_1 : \Box \qquad \Gamma, X : K_1 \vdash T_2 : *}{\Gamma \vdash (X : K_1) \leadsto^{\Box}_{*} T_2 : *}$$ means "if K_1 is a kind and T_2 is a type, then $\forall X{::}K_1 .\, T_2$ is a type"

$$\frac{\Gamma, X : K_1 \vdash t_2 : T_2 \qquad \Gamma \vdash (X{:}K_1) \leadsto^{\Box}_{*} T_2 : *}{\Gamma \vdash \lambda X{:}K_1 .\, t_2 : (X{:}K_1) \leadsto^{\Box}_{*} T_2}$$ means the typing rule (T-TAbs)

$$\frac{\Gamma \vdash t_1 : (X{:}K_{11}) \leadsto^{\Box}_{*} T_{12} \qquad \Gamma \vdash T_2 : K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}$$ means the typing rule (T-TApp)

The Essence of λ: Abstraction

Aside (Pure Type Systems, Part IV)

$\lambda_{\rightarrow}$	abstract **terms** out of **terms**	$\{(*, *)\}$
F	abstract **terms** out of **types**	$\{(*, *), (\square, *)\}$
λ_{ω}	abstract **types** out of **types**	$\{(*, *), (\square, \square)\}$
F_{ω}	$F + \lambda_{\omega}$ **(next chapter)**	$\{(*, *), (\square, *), (\square, \square)\}$

There are eight variants, each of which is $(*, *)$ plus a subset of $\{(\square, *), (\square, \square), (*, \square)\}$!

Question

What does the rule $(*, \square)$ mean? "Abstracting **types** out of **terms** by $\lambda x{:}T.\, T$?"

$$\frac{\Gamma \vdash T_1 : * \qquad \Gamma, x : T_1 \vdash K_2 : \square}{\Gamma \vdash (x{:}T_1) \rightsquigarrow^{*}_{\square} K_2 : \square}\ \text{Arrow}^{D} \qquad \frac{\Gamma, x : T_1 \vdash T_2 : K_2 \qquad \Gamma \vdash (x{:}T_1) \rightsquigarrow^{*}_{\square} K_2 : \square}{\Gamma \vdash \lambda x{:}T_1 .\, T_2 : (x{:}T_1) \rightsquigarrow^{*}_{\square} K_2}\ \text{Abs}^{D}$$

$$\frac{\Gamma \vdash T_1 : (x{:}T_{11}) \rightsquigarrow^{*}_{\square} K_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash T_1\ [t_2] : [x \mapsto t_2]K_{12}}\ \text{App}^{D}$$

$$K ::= * \mid (x{:}T) \rightsquigarrow^{*}_{\Box} K$$
$$T ::= \mathtt{Nat} \mid \lambda x{:}T.\, T \mid T\,[t] \mid (x{:}T) \rightsquigarrow^{*}_{*} T$$
$$t ::= \mathtt{zero} \mid \mathtt{succ}(t) \mid x \mid \lambda x{:}T.\, t \mid t\ t$$

$$\frac{\Gamma, x : T_1 \vdash T_2 :: K_2 \qquad \Gamma \vdash T_1 :: *}{\Gamma \vdash \lambda x{:}T_1.\, T_2 :: (x{:}T_1) \rightsquigarrow^{*}_{\Box} K_2} \text{ K-VAbs}$$

$$\frac{\Gamma \vdash T_1 :: (x{:}T_{11}) \rightsquigarrow^{*}_{\Box} K_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash T_1\ [t_2] :: [x \mapsto t_2]K_{12}} \text{ K-VApp}$$

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

$$\frac{\Gamma \vdash t_1 : (x{:}T_{11}) \rightsquigarrow^{*}_{*} T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : [x \mapsto t_2]T_{12}} \text{ T-App}$$

Example (Dependent Types)

Consider the type `NatList` and its two introduction terms `nil` and `cons`.

$$\mathtt{NatList} :: \mathtt{Nat} \rightsquigarrow^{*}_{\Box} *$$
$$\mathtt{nil} : \mathtt{NatList}\ [\mathtt{zero}]$$
$$\mathtt{cons} : (n{:}\mathtt{Nat}) \rightsquigarrow^{*}_{*} \mathtt{Nat} \rightsquigarrow^{*}_{*} \mathtt{NatList}\ [n] \rightsquigarrow^{*}_{*} \mathtt{NatList}\ [\mathtt{succ}(n)]$$

The Essence of λ: The Lambda Cube

Aside (Pure Type Systems, Part V)

$$
\begin{tikzcd}
 & F_\omega \arrow[rr, dash] \arrow[dd, dash] & & CC \arrow[dd, dash] \\
F \arrow[ur, dash] \arrow[rr, dash] \arrow[dd, dash] & & \cdot \arrow[ur, dash] \arrow[dd, dash] & \\
 & \lambda_\omega \arrow[rr, dash] & & \cdot \\
\lambda_{\to} \arrow[ur, dash] \arrow[rr, dash] & & \lambda P \arrow[ur, dash] &
\end{tikzcd}
$$

$\lambda_{\to}$	simply-typed lambda-calculus	$\{(*, *)\}$
F	parametric polymorphism	$\{(*, *), (\square, *)\}$
λ_ω	type operators	$\{(*, *), (\square, \square)\}$
λP	dependent types	$\{(*, *), (*, \square)\}$
F_ω	higher-order polymorphism	$\{(*, *), (\square, *), (\square, \square)\}$
CC	calculus of constructions	$\{(*, *), (\square, *), (\square, \square), (*, \square)\}$

Homework

Question

Extend System F_ω with local type definition as follows.

$$t ::= \ldots \mid \texttt{let}\ X = T\ \texttt{in}\ t$$
$$\Gamma ::= \ldots \mid \Gamma, X :: K = T$$

For example, the term **`let`** `X=Nat` **`in`** `(λx:X. x + 1) 4` evalutes to 5.
Extend the rules for context formation Γ ctx, type equivalence $\Gamma \vdash S \equiv T :: K$, kinding $\Gamma \vdash T :: K$, typing $\Gamma \vdash t : T$, and evaluation $t \longrightarrow t'$.