



# 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 2026



# Type-Level Computation

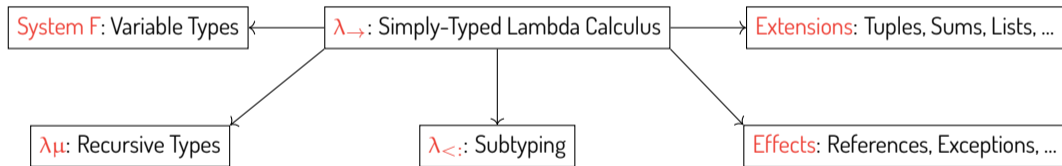
## 类型层计算

# 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 $\lambda$

## Principle (Computation)

$\lambda$ -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:

**Pair**<sub>T1,T2</sub> =  $\forall X. (T1 \rightarrow T2 \rightarrow X) \rightarrow X$ ;

**Sum**<sub>T1,T2</sub> =  $\forall X. (T1 \rightarrow X) \rightarrow (T2 \rightarrow X) \rightarrow X$ ;

**List**<sub>T</sub> =  $\forall X. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X$ ;

## Observation

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

**Pair** =  $\lambda T1. \lambda T2. (\forall X. (T1 \rightarrow T2 \rightarrow X) \rightarrow X)$ ;

**Sum** =  $\lambda T1. \lambda T2. (\forall X. (T1 \rightarrow X) \rightarrow (T2 \rightarrow X) \rightarrow X)$ ;

**List** =  $\lambda T. (\forall X. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X)$ ;



# Type-Level Computation

## Principle (Type-Level Computation)

$\lambda$ -abstraction is **THE** mechanism of defining computation.

$\text{Pair} = \lambda T1. \lambda T2. (\forall X. (T1 \rightarrow T2 \rightarrow X) \rightarrow X);$

$\text{Sum} = \lambda T1. \lambda T2. (\forall X. (T1 \rightarrow X) \rightarrow (T2 \rightarrow X) \rightarrow X);$

$\text{List} = \lambda T. (\forall x. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X);$

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  $\text{Id} = \lambda X. X$ . The following types are equivalent:

$\text{Nat} \rightarrow \text{Bool} \quad \text{Nat} \rightarrow \text{Id} \text{ Bool} \quad \text{Id Nat} \rightarrow \text{Id} \text{ Bool} \quad \text{Id Nat} \rightarrow \text{Bool} \quad \text{Id} (\text{Nat} \rightarrow \text{Bool})$



# Type-Level Abstraction & Application

## Syntax

$$T ::= X \mid \lambda X. T \mid T T \mid T \rightarrow T \mid \text{Bool} \mid \text{Nat} \mid \dots$$

$$TV ::= \lambda X. T \mid TV \rightarrow TV \mid \text{Bool} \mid \text{Nat} \mid \dots$$

## Evaluation: $T \longrightarrow T'$

$$\frac{T_1 \longrightarrow T'_1}{T_1 T_2 \longrightarrow T'_1 T_2}$$

$$\frac{T_2 \longrightarrow T'_2}{TV_1 T_2 \longrightarrow TV_1 T'_2}$$

$$\frac{}{(\lambda X. T_{12}) TV_2 \longrightarrow [X \mapsto TV_2]T_{12}}$$

$$\frac{T_1 \longrightarrow T'_1}{(T_1 \rightarrow T_2) \longrightarrow (T'_1 \rightarrow T_2)}$$

$$\frac{T_2 \longrightarrow T'_2}{(TV_1 \rightarrow T_2) \longrightarrow (TV_1 \rightarrow T'_2)}$$

## Question

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



# Issue 1: Unequal Equivalent Types

## Example

Consider  $\text{Id} = \lambda X. X$ . Two type-level values  $\lambda X. \text{Id } X$  and  $\lambda X. X$  are **unequal** but **equivalent**.

## Observation

We do not care about how types evaluate.

We care about if they are equivalent.

## Equivalence: $S \equiv T$

$$\frac{}{\overline{T \equiv T}}$$

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

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

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2}$$

$$\frac{S_2 \equiv T_2}{\lambda X. S_2 \equiv \lambda X. T_2}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 S_2 \equiv T_1 T_2}$$

$$\frac{}{(\lambda X. T_{12}) T_2 \equiv [X \mapsto T_2] T_{12}}$$



# Issue 2: Errors in Type-Level Computation

## Example

Consider  $(\lambda X. X X) \text{Nat}$ . The type evaluates to  $\text{Nat Nat}$ , which is an **illy-formed** type.

Consider  $(\lambda X. X X) (\lambda X. X X)$ . The type's evaluation **diverges**.

## Principle (Characterization of Type-Level Computation)

Recall that **types** characterize **terms**.

**What** can characterize **types**?

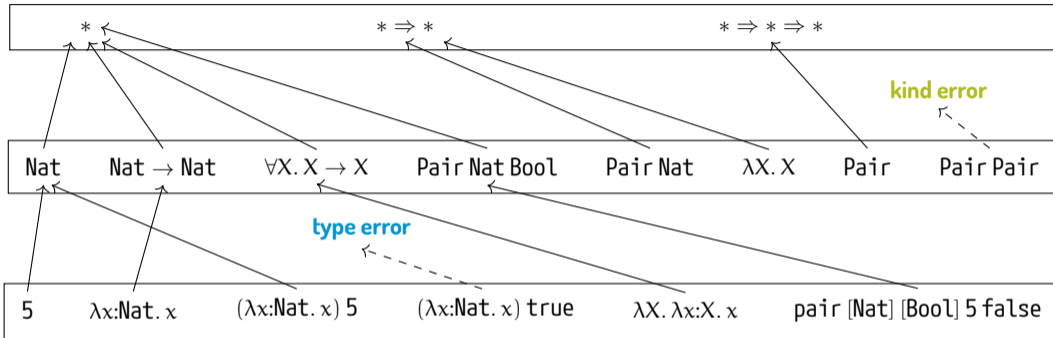
## Kinds: “Types of Types”

**Kinds** characterize **types**.

- $*$  proper types (e.g.,  $\text{Bool}$  and  $\text{Nat} \rightarrow \text{Bool}$ )
- $* \Rightarrow *$  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  $\forall X. X \rightarrow X$  and  $\lambda X. X \rightarrow X$ ?
- Why doesn't an arrow type  $\text{Nat} \rightarrow \text{Nat}$  have an arrow kind like  $* \Rightarrow *$ ?

## Syntax

$$T ::= X \mid \lambda X :: K. T \mid T T \mid T \rightarrow T \mid \text{Bool} \mid \text{Nat} \mid \dots$$
$$K ::= * \mid K \Rightarrow K$$
$$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, X :: K$$

$\Gamma \vdash T :: K$ : “type  $T$  has kind  $K$  in context  $\Gamma$ ”

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1. T_2 :: K_1 \Rightarrow K_2}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1 T_2 :: K_{12}}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$$

$$\overline{\Gamma \vdash \text{Bool} :: *}$$

$$\overline{\Gamma \vdash \text{Nat} :: *}$$

## Observation

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



# $\lambda_{\omega} = \lambda_{\rightarrow} + \text{Type Operators}$

$t ::=$

$x$

$\lambda x:T. t$

$t t$

$v ::=$

$\lambda x:T. t$

$T ::=$

$X$

$\lambda X::K. T$

$T T$

$T \rightarrow T$

$\Gamma ::=$

$\emptyset$

$\Gamma, x : T$

$\Gamma, X :: K$

$K ::=$

$*$

$K \Rightarrow K$

*terms:*

*variable*

*abstraction*

*application*

*values:*

*abstraction value*

*types:*

*type variable*

*operator abstraction*

*operator application*

*type of functions*

*contexts:*

*empty context*

*term variable binding*

*type variable binding*

*kinds:*

*kind of proper types*

*kind of operators*

## Typing

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

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

## Observation

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

## Question

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



# Approach 1: Parallel Reduction

$S \Rightarrow T$ : “type  $S$  parallelly reduces to type  $T$ ”

$$\frac{}{T \Rightarrow T} \quad \frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 \rightarrow S_2 \Rightarrow T_1 \rightarrow T_2} \quad \frac{S_2 \Rightarrow T_2}{\lambda X::K_1. S_2 \Rightarrow \lambda X::K_1. T_2} \quad \frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 S_2 \Rightarrow T_1 T_2}$$

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

## Example

Let  $S \stackrel{\text{def}}{=} \text{Id Nat} \rightarrow \text{Bool}$  and  $T \stackrel{\text{def}}{=} \text{Id} (\text{Nat} \rightarrow \text{Bool})$ . Then

$$S = ((\lambda X::*. X) \text{Nat}) \rightarrow \text{Bool} \Rightarrow \text{Nat} \rightarrow \text{Bool}, \quad T = (\lambda X::*. X) (\text{Nat} \rightarrow \text{Bool}) \Rightarrow \text{Nat} \rightarrow \text{Bool}.$$

## Theorem

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

# 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.

$$\frac{T_1 \rightsquigarrow T'_1}{T_1 T_2 \rightsquigarrow T'_1 T_2}$$

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

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

$$\frac{T \not\rightsquigarrow}{T \Downarrow T}$$

$$\frac{S \rightsquigarrow T \quad T \Downarrow T'}{S \Downarrow T'}$$

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

$$\frac{S \Downarrow S' \quad T \Downarrow T' \quad \Gamma \vdash S' \leftrightarrow T' :: *}{\Gamma \vdash S \Leftrightarrow T :: *}$$

$$\frac{X \notin \Gamma \quad \Gamma, X :: K_1 \vdash S X \Leftrightarrow T X :: K_2}{\Gamma \vdash S \Leftrightarrow T :: K_1 \Rightarrow K_2}$$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X \leftrightarrow X :: K} \quad \frac{\Gamma \vdash S_1 \leftrightarrow T_1 :: * \quad \Gamma \vdash S_2 \leftrightarrow T_2 :: *}{\Gamma \vdash S_1 \rightarrow S_2 \leftrightarrow T_1 \rightarrow T_2 :: *}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash S_2 \leftrightarrow T_2 :: K_1}{\Gamma \vdash S_1 S_2 \leftrightarrow T_1 T_2 :: K_2}$$

# Parallel Reduction vs. Weak-Head Reduction



## Example

```
Pair =  $\lambda Y::*. \{Y, Y\};$ 
```

```
List =  $\lambda Y::*. (\mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{Y, X\} \rangle);$ 
```

Determine that  $\text{List}(\text{List}(\text{Pair}(\text{Nat})))$  and  $\text{List}(\text{List}(\{\text{Nat}, \text{Nat}\}))$  are equivalent.

## Parallel Reduction

$$\text{List}(\text{List}(\text{Pair}(\text{Nat}))) \Rightarrow^* \mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\mu Y. \langle \text{nil}:\text{Unit}, \text{cons}:\{\{\text{Nat}, \text{Nat}\}, Y\} \rangle, X\} \rangle$$
$$\text{List}(\text{List}(\{\text{Nat}, \text{Nat}\})) \Rightarrow^* \mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\mu Y. \langle \text{nil}:\text{Unit}, \text{cons}:\{\{\text{Nat}, \text{Nat}\}, Y\} \rangle, X\} \rangle$$



# Parallel Reduction vs. Weak-Head Reduction

## Example

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

Determine that  $\text{List}(\text{List}(\text{Pair}(\text{Nat})))$  and  $\text{List}(\text{List}(\{\text{Nat}, \text{Nat}\}))$  are equivalent.

## Weak-Head Reduction

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

$$\text{List}(\text{List}(\text{Pair}(\text{Nat}))) \Downarrow \mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{List}(\text{Pair}(\text{Nat})), X\} \rangle$$
$$\text{List}(\text{List}(\{\text{Nat}, \text{Nat}\})) \Downarrow \mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{List}(\{\text{Nat}, \text{Nat}\}), X\} \rangle$$

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

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



# System $F_\omega$ : The Combination of System F and $\lambda_\omega$

## Syntax

$t ::= x \mid \lambda x:T. t \mid t t \mid \lambda X::K. t \mid t [T] \mid \{ *T, t \} \text{ as } T \mid \text{let } \{ X, x \} = t \text{ in } t$

$v ::= \lambda x:T. t \mid \lambda X::K. t \mid \{ *T, v \} \text{ as } T$

$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{ \exists X::K, T \}$

$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, X :: K$

$K ::= * \mid K \Rightarrow K$

## 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

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

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

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

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

## 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 =  $\lambda i:\text{Nat}. i$ ,  
           inc =  $\lambda i:\text{Nat}. \text{succ}(i)$  } }  
  as {  $\exists$  Counter,  
       { new: Counter, get: Counter  $\rightarrow$  Nat, inc: Counter  $\rightarrow$  Counter } };  
  ▶ counterADT : {  $\exists$  Counter,  
                  { new: Counter, get: Counter  $\rightarrow$  Nat, inc: Counter  $\rightarrow$  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**!

$$\text{PairSig} = \{\exists \text{Pair} :: * \Rightarrow * \Rightarrow *,$$
$$\quad \{\text{pair} : \forall X. \forall Y. X \rightarrow Y \rightarrow (\text{Pair } X \ Y),$$
$$\quad \text{fst} : \forall X. \forall Y. (\text{Pair } X \ Y) \rightarrow X,$$
$$\quad \text{snd} : \forall X. \forall Y. (\text{Pair } X \ Y) \rightarrow Y\}\};$$

# Abstract Type Operators



## Example

```
pairADT = {*( $\lambda X. \lambda Y. \forall R. (X \rightarrow Y \rightarrow R) \rightarrow R$ ),  
           {pair =  $\lambda X. \lambda Y. \lambda x:X. \lambda y:Y. \lambda R. \lambda p:(X \rightarrow Y \rightarrow R). p\ x\ y$ ,  
            fst  =  $\lambda X. \lambda Y. \lambda p:(\forall R. (X \rightarrow Y \rightarrow R) \rightarrow R). p\ [X]\ (\lambda x:X. \lambda y:Y. x)$ ,  
            snd  =  $\lambda X. \lambda Y. \lambda p:(\forall R. (X \rightarrow Y \rightarrow R) \rightarrow R). p\ [Y]\ (\lambda x:X. \lambda y:Y. y)$ }}
```

**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 =  $\lambda X. \langle \text{none}:\text{Unit}, \text{some}:X \rangle;$   
none =  $\lambda X. \langle \text{none}=\text{unit} \rangle \text{ as } (\text{Option } X);$   
▶ none :  $\forall X. (\text{Option } X)$   
some =  $\lambda X. \lambda x:X. \langle \text{some}=x \rangle \text{ as } (\text{Option } X);$   
▶ some :  $\forall X. X \rightarrow (\text{Option } X)$ 
```

## List: Combination with Variants, Tuples, and Recursive Types

```
List =  $\mu L :: (* \Rightarrow *). \lambda X. \langle \text{nil}:\text{Unit}, \text{cons}:\{X, (L X)\} \rangle;$   
nil =  $\lambda X. \langle \text{nil}=\text{unit} \rangle \text{ as } (\text{List } X);$   
▶ nil :  $\forall X. (\text{List } X)$   
cons =  $\lambda X. \lambda h:X. \lambda t:(\text{List } X). \langle \text{cons}=\{h,t\} \rangle \text{ as } (\text{List } X);$   
▶ cons :  $\forall X. X \rightarrow (\text{List } X) \rightarrow (\text{List } X)$ 
```

# More Examples



## Queue: Implementing a Queue using Two Lists

```
QueueSig = { $\exists$ Q :: * $\Rightarrow$ *,
  {empty :  $\forall$ X. (Q X),
   insert:  $\forall$ X. X  $\rightarrow$  (Q X)  $\rightarrow$  (Q X),
   remove:  $\forall$ X. (Q X)  $\rightarrow$  Option {X,(Q X)}}};
queueADT = {*( $\lambda$ X. {List X,List X}),
  {empty =  $\lambda$ X. {nil [X],nil [X]},
   insert =  $\lambda$ X.  $\lambda$ a:X.  $\lambda$ q:{List X,List X}. {(cons [X] a q.1),q.2},
   remove =
     $\lambda$ X.  $\lambda$ q:{List X,List X}.
    let q' = case q.2 of <nil=u>  $\Rightarrow$  {nil [X], reverse [X] q.1}
              | <cons={h,t}>  $\Rightarrow$  q
    in case q'.2 of
      <nil=u>  $\Rightarrow$  none [{X},{List X,List X}]
      | <cons={h,t}>  $\Rightarrow$  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:

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

## Preservation of Shapes (for Arrows)

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

## 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  $\Gamma \vdash t' : T$ .

## Canonical Forms (for Arrows)

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

## Theorem (30.3.16)

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

## Remark

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

## Context Formation

$$\frac{}{\emptyset \text{ ctx}}$$

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash T :: *}{\Gamma, x : T \text{ ctx}}$$

$$\frac{\Gamma \text{ ctx}}{\Gamma, X :: K \text{ ctx}}$$

## Theorem

If  $\Gamma \text{ ctx}$  and  $\Gamma \vdash t : T$ , then  $\Gamma \vdash T :: *$ .

# Fragments of System $F_\omega$

## Definition

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

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

## 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  $\mathcal{K}_3 = \{*, * \Rightarrow *, * \Rightarrow * \Rightarrow *, \dots\}$ , i.e., type-level abstractions are over proper types.



# Type-Level Natural Numbers

## Remark

The kinding system of  $\lambda_\omega$  and  $F_\omega$  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

$$K ::= * \mid K \Rightarrow K \mid \mathbb{N}$$
$$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{\exists X::K, T\} \mid \text{ZERO} \mid \text{SUCC } T \mid \dots$$

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

```
List =  $\lambda X. \mu L :: (\mathbb{N} \Rightarrow *) . \lambda M :: \mathbb{N} . \text{IF ISZERO}(M) \text{ THEN Unit ELSE } \{X, (L (\text{PRED } M))\};$ 
```

►  $\text{List} :: * \Rightarrow \mathbb{N} \Rightarrow *$



# Type-Level Natural Numbers

## Example

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

► List ::  $*$   $\Rightarrow$   $\mathbb{N} \Rightarrow *$

```
nil = λX. unit as (List X ZERO);
```

► nil :  $\forall X$ . (List X ZERO)

```
cons = λX. λM:: $\mathbb{N}$ . λh:X. λt:(List X M). {h,t} as (List X (SUCC M));
```

► cons :  $\forall X$ .  $\forall M::\mathbb{N}$ .  $X \rightarrow$  (List X M)  $\rightarrow$  (List X (SUCC M))

## Example

```
PLUS = μP::( $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$ ). λM:: $\mathbb{N}$ . λN:: $\mathbb{N}$ . IF ISZERO(M) THEN N ELSE SUCC (P (PRED M) N);
```

► PLUS ::  $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$



# Type-Level Natural Numbers

## Number-Number Kind

Type-level recursion would render type equivalence **undecidable**.

Let us consider  $\mathbb{N}$  as an **inductively-defined** kind.

$T ::= X \mid \lambda X :: K. T \mid T T \mid T \rightarrow T \mid \forall X :: K. T \mid \{\exists X :: K, T\} \mid \text{ZERO} \mid \text{SUCC } T \mid \text{ITER } T \text{ WITH ZERO} \Rightarrow T \mid \text{SUCC} \Rightarrow Y. T$

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

$$\frac{}{\Gamma \vdash \text{ZERO} :: \mathbb{N}} \qquad \frac{\Gamma \vdash T_1 :: \mathbb{N}}{\Gamma \vdash \text{SUCC } T_1 :: \mathbb{N}} \qquad \frac{\Gamma \vdash T_0 :: \mathbb{N} \quad \Gamma \vdash T_1 :: K \quad \Gamma, Y :: K \vdash T_2 :: K}{\Gamma \vdash \text{ITER } T_0 \text{ WITH ZERO} \Rightarrow T_1 \mid \text{SUCC} \Rightarrow Y. T_2 :: K}$$

## Example

List =  $\lambda X. \lambda M :: \mathbb{N}. \text{ITER } M \text{ OF ZERO} \Rightarrow \text{Unit} \mid \text{SUCC} \Rightarrow Y. \{X, Y\};$

► List ::  $* \Rightarrow \mathbb{N} \Rightarrow *$

PLUS =  $\lambda M :: \mathbb{N}. \lambda N :: \mathbb{N}. \text{ITER } M \text{ OF ZERO} \Rightarrow N \mid \text{SUCC} \Rightarrow Y. \text{SUCC } Y;$

► PLUS ::  $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$



# Type-Level Natural Numbers

## Term-Level Case on Type-Level Natural Numbers

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

### Example

List =  $\lambda X. \lambda M :: \mathbb{N}. \text{ITER } M \text{ OF ZERO} \Rightarrow \text{Unit} \mid \text{SUCC} \Rightarrow Y. \{X, Y\};$

► List :: \*  $\Rightarrow \mathbb{N} \Rightarrow *$

PLUS =  $\lambda M :: \mathbb{N}. \lambda N :: \mathbb{N}. \text{ITER } M \text{ OF ZERO} \Rightarrow N \mid \text{SUCC} \Rightarrow Y. \text{SUCC } Y;$

► PLUS ::  $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$

► append :  $\forall X. \forall M :: \mathbb{N}. \forall N :: \mathbb{N}. (\text{List } X \ M) \rightarrow (\text{List } X \ N) \rightarrow (\text{List } X \ (\text{PLUS } M \ N))$

append =  $\lambda X. \mathbf{fix} \ \lambda f. \lambda M :: \mathbb{N}. \lambda N :: \mathbb{N}. \lambda l1 : (\text{List } X \ M). \lambda l2 : (\text{List } X \ N).$

**tcase** M **of** ZERO  $\Rightarrow$  **let** unit = l1 **in** l2 **as** (List X (PLUS M N))

    SUCC M'  $\Rightarrow$  **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: $\Gamma \vdash S \equiv T :: K$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K}$$

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

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

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \quad \Gamma \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2 :: *}$$

$$\frac{\Gamma, X :: K_1 \vdash S_2 \equiv T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1. S_2 \equiv \lambda X :: K_1. T_2 :: K_1 \Rightarrow K_2}$$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: K_{11} \Rightarrow K_{12} \quad \Gamma \vdash S_2 \equiv T_2 :: K_{11}}{\Gamma \vdash S_1 S_2 \equiv T_1 T_2 :: K_{12}}$$

$$\frac{\Gamma, X :: K_{11} \vdash T_{12} :: K_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash (\lambda X :: K_{11}. T_{12}) T_2 \equiv [X \mapsto T_2] T_{12} :: K_{12}}$$

# Type-Level Natural Numbers



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

$$\frac{}{\Gamma \vdash \text{ZERO} \equiv \text{ZERO} :: \mathbb{N}}$$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: \mathbb{N}}{\Gamma \vdash \text{SUCC } S_1 \equiv \text{SUCC } T_1 :: \mathbb{N}}$$

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

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

$$\frac{\Gamma \vdash T_0 :: \mathbb{N} \quad \Gamma \vdash T_1 :: K \quad \Gamma, Y :: K \vdash T_2 :: K}{\Gamma \vdash \text{ITER (SUCC } T_0) \text{ WITH ZERO} \Rightarrow T_1 \mid \text{SUCC} \Rightarrow Y. T_2}$$
$$\equiv$$
$$[\Gamma \vdash \text{ITER } T_0 \text{ WITH ZERO} \Rightarrow T_1 \mid \text{SUCC} \Rightarrow Y. T_2] T_2 :: K$$



# Type-Level Natural Numbers

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

$$\frac{S \equiv T :: \mathbb{N} \in \Gamma}{\Gamma \vdash S \equiv T :: \mathbb{N}}$$

$$\frac{\Gamma \vdash \text{SUCC } S_1 \equiv \text{SUCC } T_1 :: \mathbb{N}}{\Gamma \vdash S_1 \equiv T_1 :: \mathbb{N}}$$

## Example

`append`  $\equiv$   $\lambda X.$  **fix**  $\lambda f:_. \lambda M::\mathbb{N}. \lambda N::\mathbb{N}. \lambda l_1:(\text{List } X \ M). \lambda l_2:(\text{List } X \ N).$

`tcase`  $M$  of `ZERO`  $\Rightarrow$  `t1` | `SUCC`  $M'$   $\Rightarrow$  `t2`

`t1`  $\equiv$  `let` `unit`  $=$  `l1` `in` `l2` `as`  $(\text{List } X \ (\text{PLUS } M \ N))$

`t2`  $\equiv$  `let`  $\{h, t\} = l_1$  `in`  $\{h, (f \ M' \ N \ t \ l_2)\}$  `as`  $(\text{List } X \ (\text{PLUS } M \ N))$

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

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

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



# 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**.

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

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

$$\theta ::= \top \mid \perp \mid \neg\theta \mid \theta \wedge \theta \mid \theta \vee \theta \mid I = I \mid I \leq I \mid \dots$$

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

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

Length-indexed lists:  $\lambda X. \mu L :: (\mathbb{N} \Rightarrow *) . \lambda M :: \mathbb{N}. \{\exists \{M=0\}, \text{Unit}\} + \{\exists \{M' :: \mathbb{N} \mid M=M'+1\}, \{X, (L M')\}\}$ .



# Indexed Types

## Remark

The kind  $\{\alpha : \mathbb{N} \mid \theta\}$  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

$$\frac{\Gamma \vdash t : \forall\{\alpha :: \mathbb{N} \mid \theta\}. T \quad \Gamma \vdash i :: \{\alpha :: \mathbb{N} \mid \theta\}}{\Gamma \vdash t [i] : [\alpha \mapsto i]T}$$

$$\frac{\Gamma \models [\alpha \mapsto i]\theta}{\Gamma \vdash i :: \{\alpha :: \mathbb{N} \mid \theta\}}$$

$$\frac{\Gamma \vdash t : \forall\{\theta\}. T \quad \Gamma \vdash @ :: \{\theta\}}{\Gamma \vdash t [@] : T}$$

$$\frac{\Gamma \models \theta}{\Gamma \vdash @ :: \{\theta\}}$$

## Constraint Checking

For example, consider  $\{\alpha :: \mathbb{N} \mid \alpha \geq 5\}$ ,  $x : (\text{List Nat } \alpha) \models \neg(\alpha = 0)$ .

We can resort to check validity of the formula in first-order logic:  $\forall \alpha : \mathbb{N}. (\alpha \geq 5) \implies \neg(\alpha = 0)$ .



# 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  $\ell$  and term  $t$  by  $\{\ell = t \mid r\}$ .

```
origin = {x = 0 | {y = 0 | {}}};  
origin3 = {z = 0 | origin};  
named =  $\lambda s. \lambda r. \{name = s \mid r\}$ ;
```

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

```
distance =  $\lambda p. \text{sqrt} ((p.x * p.x) + (p.y * p.y))$ ;  
distance (named "2d" origin) + distance origin3;
```

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

```
update_name =  $\lambda r. \lambda s. \{name = s \mid r - name\}$ ;  
rename_name_nn =  $\lambda r. \{nn = r.name \mid r - name\}$ ;
```



# Scoped Labels

## Observation

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

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

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

$$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{\exists X::K, T\} \mid \langle \rangle \mid \langle \ell : T \mid T \rangle \mid \{T\}$$

For example, the record type  $\{x : \text{Nat}, y : \text{Nat}\}$  is encoded as  $\langle x : \text{Nat} \mid \langle y : \text{Nat} \mid \langle \rangle \rangle \rangle$ .

Below are the kinding rules for row:

$$\frac{}{\Gamma \vdash \langle \rangle :: \text{row}} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: \text{row}}{\Gamma \vdash \langle \ell : T_1 \mid T_2 \rangle :: \text{row}} \qquad \frac{\Gamma \vdash T :: \text{row}}{\Gamma \vdash \{T\} :: *}$$

## Well-Typed Record Operations

$$\{\ell = \_ \mid \_ \} : \forall R::\text{row}. \forall X::*. X \rightarrow \{R\} \rightarrow \langle \ell : X \mid R \rangle$$

$$(\_.\ell) : \forall R::\text{row}. \forall X::*. \langle \ell : X \mid R \rangle \rightarrow X$$

$$(\_ - \ell) : \forall R::\text{row}. \forall X::*. \langle \ell : X \mid R \rangle \rightarrow \{R\}$$

# Row Equivalence

## Question

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

## Type-Level Row Equivalence

$$\frac{}{() \equiv ()} \quad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{(\ell : S_1 \mid S_2) \equiv (\ell : T_1 \mid T_2)} \quad \frac{\ell \neq \ell'}{(\ell : T_1 \mid (\ell' : T_2 \mid T_3)) \equiv (\ell' : T_2 \mid (\ell : T_1 \mid T_3))}$$

## Example

$$\frac{\frac{\vdots}{\emptyset \vdash \{x = 0 \mid \{y = \text{true} \mid \{\}\}\} : \{(x : \text{Nat} \mid (y : \text{Bool} \mid ()))\}} \quad \frac{x \neq y}{\{(x : \text{Nat} \mid (y : \text{Bool} \mid ()))\} \equiv \{(y : \text{Bool} \mid (x : \text{Nat} \mid ()))\}}}{\emptyset \vdash \{x = 0 \mid \{y = \text{true} \mid \{\}\}\} : \{(y : \text{Bool} \mid (x : \text{Nat} \mid ()))\}}}{\emptyset \vdash \{x = 0 \mid \{y = \text{true} \mid \{\}\}\}.y : \text{Bool}}$$



# Use Rows for Extensible Variants

## Principle

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

$$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{\exists X::K, T\} \mid () \mid (\ell : T \mid T) \mid \{T\} \mid \langle T \rangle$$

For example, the variant type  $\langle \text{none} : \text{Unit}, \text{some} : \text{Nat} \rangle$  is encoded as  $\langle (\text{none} : \text{Unit} \mid (\text{some} : \text{Nat} \mid ())) \rangle$ .

## Well-Typed Variant Operations

- **Injection:** We write  $\langle \ell = t \rangle$  to build a variant with label  $\ell$  and term  $t$ .

$$\langle \ell = \_ \rangle : \forall R::\text{row}. \forall X::*. X \rightarrow \langle (\ell : X \mid R) \rangle$$

- **Embedding:** We write  $\langle \ell \mid v \rangle$  to embed a variant  $v$  in a type that also allows label  $\ell$ .

$$\langle \ell \mid \_ \rangle : \forall R::\text{row}. \forall X::*. \langle R \rangle \rightarrow \langle (\ell : X \mid R) \rangle$$

- **Decomposition:** We write  $\ell \in v ? t_1 : t_2$  to decompose a variant  $v$  and check if it is labeled with  $\ell$ .

$$(\ell \in \_ ? \_ : \_) : \forall R::\text{row}. \forall X::*. \forall Y::*. \langle (\ell : X \mid R) \rangle \rightarrow (X \rightarrow Y) \rightarrow (\langle R \rangle \rightarrow Y) \rightarrow Y$$



# Type-Level Labels

## Question

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

## Principle

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

$T ::= X \mid \lambda X :: K. T \mid T T \mid T \rightarrow T \mid \forall X :: K. T \mid \{\exists X :: K, T\} \mid () \mid (T : T \mid T) \mid \{T\} \mid \langle T \rangle \mid \#l$

$$\frac{}{\Gamma \vdash \#l :: \text{label}} \qquad \frac{\Gamma \vdash T_1 :: \text{label} \quad \Gamma \vdash T_2 :: * \quad \Gamma \vdash T_3 :: \text{row}}{\Gamma \vdash (T_1 : T_2 \mid T_3) :: \text{row}}$$



# 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.

$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{\exists X::K, T\} \mid () \mid (T : T \mid T) \mid \{T\} \mid \langle T \rangle \mid \#l \mid \text{map}$

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

## Example

Consider  $\text{Meta} = \lambda T. \{() \#name:String, \#show:(T \rightarrow String) \}$ .

Then  $\text{map Meta } () \#x:Nat, \#y:Bool$  is equivalent to  $() \#x:(\text{Meta Nat}), \#y:(\text{Meta Bool})$ .



# Example: A Generic Table Formatter

Meta =  $\lambda T. \{ \{ \#name:String, \#show:(T \rightarrow String) \} \};$

► Meta :: \*  $\Rightarrow$  \*

Folder =  $\lambda R::row. \forall TF::(row \Rightarrow *).$

$(\forall L::label. \forall T. \forall R::row. TF\ R \rightarrow TF\ (\{ L : T \mid R \}) \rightarrow TF\ () \rightarrow TF\ R;$

► Folder :: row  $\Rightarrow$  \*

► mk\_table :  $\forall R::row. Folder\ R \rightarrow \{ \text{map Meta } R \} \rightarrow \{ R \} \rightarrow String$

mk\_table =  $\lambda R::row. \lambda fl:(Folder\ R). \lambda mr:\{\text{map Meta } R\}. \lambda x:\{R\}.$

$fl\ (\lambda R::row. \{\text{map Meta } R\} \rightarrow \{R\} \rightarrow String)$

$(\lambda L::label. \lambda T. \lambda R::row.$

$\lambda acc:(\{\text{map Meta } R\} \rightarrow \{R\} \rightarrow String).$

$\lambda mr:\{\text{map Meta } (\{ L : T \mid R \})\}.$

$\lambda x:\{(\{ L : T \mid R \})\}.$

$"<tr><th>" \wedge mr.L.name \wedge "</th><td>" \wedge mr.L.show\ x.L \wedge "</td></tr>" \wedge acc\ (mr-L)\ (x-L))$

$(\lambda _:\{\text{map Meta } ()\}. \lambda _:\{()\}. "")\ mr\ x$



# The Essence of $\lambda$ : 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.,  $M = \{(\emptyset \vdash * : \square)\}$ , meaning “ $*$  is a kind for (proper) types.”

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



# The Essence of $\lambda$ : Abstraction

## Principle

- In  $\lambda_{\rightarrow}$ , we use  $\lambda x:T. t$  to abstract **terms** out of **terms**.
- In  $\lambda_{\omega}$ , 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.,  $S = \{*, \square\}$ . Let  $M$  be a set of **axioms**, e.g.,  $M = \{(\emptyset \vdash * : \square)\}$ .

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

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

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



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

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash A \rightsquigarrow_{s_2}^{s_1} B : s_2} \text{ Arrow}$$

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

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

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

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

$$\frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightsquigarrow_*^* T_2 : *}$$

means “if  $T_1, T_2$  are types, then  $T_1 \rightarrow T_2$  is a type”

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \quad \Gamma \vdash T_1 \rightsquigarrow_*^* T_2 : *}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightsquigarrow_*^* T_2}$$

means the typing rule (T-Abs)

$$\frac{\Gamma \vdash t_1 : T_{11} \rightsquigarrow_*^* T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

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

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash A \rightsquigarrow_{s_2}^{s_1} B : s_2} \text{ Arrow}$$

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

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

## $\lambda_\omega$ : Abstracting Types out of Types

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

$$\frac{\Gamma \vdash K_1 : \square \quad \Gamma \vdash K_2 : \square}{\Gamma \vdash K_1 \rightsquigarrow_\square K_2 : \square}$$

means “if  $K_1, K_2$  are kinds, then  $K_1 \Rightarrow K_2$  is a kind”

$$\frac{\Gamma, X : K_1 \vdash T_2 : K_2 \quad \Gamma \vdash K_1 \rightsquigarrow_\square K_2 : \square}{\Gamma \vdash \lambda X : K_1. T_2 : K_1 \rightsquigarrow_\square K_2}$$

means the typing rule (K-Abs)

$$\frac{\Gamma \vdash T_1 : K_{11} \rightsquigarrow_\square K_{12} \quad \Gamma \vdash T_2 : K_{11}}{\Gamma \vdash T_1 T_2 : K_{12}}$$

means the typing rule (K-App)

# The Essence of $\lambda$ : 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**  $(X::*) \Rightarrow T$ : the domain is a **kind** and the range is a **type**.

In System F $_{\omega}$ , there is a generalized form  $\forall X::K. T$ , or as a dependent arrow  $(X::K) \Rightarrow T$ .

## 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

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

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



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

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

## System F: Abstracting Terms out of Types

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

$$\frac{\Gamma \vdash K_1 : \square \quad \Gamma, X : K_1 \vdash T_2 : *}{\Gamma \vdash (X : K_1) \rightsquigarrow_*^\square T_2 : *} \quad \text{means "if } K_1 \text{ is a kind and } T_2 \text{ is a type, then } \forall X::K_1. T_2 \text{ is a type"}$$

$$\frac{\Gamma, X : K_1 \vdash t_2 : T_2 \quad \Gamma \vdash (X:K_1) \rightsquigarrow_*^\square T_2 : *}{\Gamma \vdash \lambda X:K_1. t_2 : (X:K_1) \rightsquigarrow_*^\square T_2} \quad \text{means the typing rule (T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : (X:K_{11}) \rightsquigarrow_*^\square T_{12} \quad \Gamma \vdash T_2 : K_{11}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2]T_{12}} \quad \text{means the typing rule (T-TApp)}$$



# The Essence of $\lambda$ : Abstraction

## Aside (Pure Type Systems, Part IV)

$\lambda_{\rightarrow}$	abstract <b>terms</b> out of <b>terms</b>	$\{(*, *)\}$
$F$	abstract <b>terms</b> out of <b>types</b>	$\{(*, *), (\square, *)\}$
$\lambda_{\omega}$	abstract <b>types</b> out of <b>types</b>	$\{(*, *), (\square, \square)\}$
$F_{\omega}$	$F + \lambda_{\omega}$	$\{(*, *), (\square, *), (\square, \square)\}$

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  $\lambda x:T. T$ ?”

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

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



$$K ::= * \mid (x:T) \rightsquigarrow_{\square}^* K$$

$$T ::= \text{Nat} \mid \lambda x:T. T \mid T[t] \mid (x:T) \rightsquigarrow_*^* T$$

$$t ::= \text{zero} \mid \text{succ}(t) \mid x \mid \lambda x:T. t \mid t t$$

$$\frac{\Gamma, x : T_1 \vdash T_2 :: K_2 \quad \Gamma \vdash T_1 :: *}{\Gamma \vdash \lambda x:T_1. T_2 :: (x:T_1) \rightsquigarrow_{\square}^* K_2} \text{K-VAbs}$$

$$\frac{\Gamma \vdash T_1 :: (x:T_{11}) \rightsquigarrow_{\square}^* K_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash T_1[t_2] :: [x \mapsto t_2]K_{12}} \text{K-VApp}$$

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

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

## Example (Dependent Types)

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

$$\text{NatList} :: \text{Nat} \rightsquigarrow_{\square}^* *$$

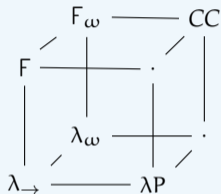
$$\text{nil} : \text{NatList} [\text{zero}]$$

$$\text{cons} : (n:\text{Nat}) \rightsquigarrow_*^* \text{Nat} \rightsquigarrow_*^* \text{NatList} [n] \rightsquigarrow_*^* \text{NatList} [\text{succ}(n)]$$

# The Essence of $\lambda$ : The Lambda Cube



## Aside (Pure Type Systems, Part V)



$\lambda_{\rightarrow}$	simply-typed lambda-calculus	$\{(*, *)\}$
F	parametric polymorphism	$\{(*, *), (\square, *)\}$
$\lambda_{\omega}$	type operators	$\{(*, *), (\square, \square)\}$
$\lambda P$	dependent types	$\{(*, *), (*, \square)\}$
$F_{\omega}$	higher-order polymorphism	$\{(*, *), (\square, *), (\square, \square)\}$
CC	calculus of constructions	$\{(*, *), (\square, *), (\square, \square), (*, \square)\}$

## Question

Extend System  $F_{\omega}$  with local type definition as follows.

$$t ::= \dots \mid \mathbf{let} X = T \mathbf{in} t$$
$$\Gamma ::= \dots \mid \Gamma, X :: K = T$$

For example, the term **let**  $X = \text{Nat}$  **in**  $(\lambda x : X. x + 1) 4$  evaluates to 5.

Extend the rules for context formation  $\Gamma \text{ ctx}$ , type equivalence  $\Gamma \vdash S \equiv T :: K$ , kinding  $\Gamma \vdash T :: K$ , typing  $\Gamma \vdash t : T$ , and evaluation  $t \longrightarrow t'$ .