



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

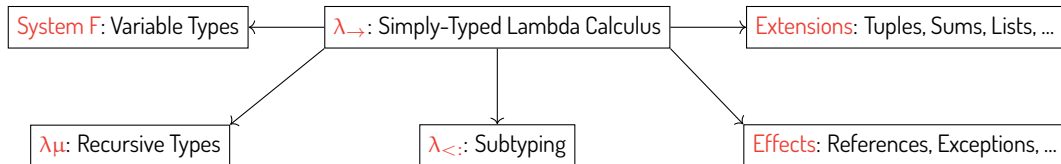
类型层计算

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 + λ_{μ} 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 λ_{\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_{T₁,T₂} = $\forall X. (T_1 \rightarrow T_2 \rightarrow X) \rightarrow X$;

Sum_{T₁,T₂} = $\forall X. (T_1 \rightarrow X) \rightarrow (T_2 \rightarrow X) \rightarrow X$;

List_T = $\forall x. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X$;

Observation

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

Pair = $\lambda T_1. \lambda T_2. (\forall X. (T_1 \rightarrow T_2 \rightarrow X) \rightarrow X)$;

Sum = $\lambda T_1. \lambda T_2. (\forall X. (T_1 \rightarrow X) \rightarrow (T_2 \rightarrow X) \rightarrow X)$;

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

Type-Level Computation

Principle (Type-Level Computation)

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

```
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 =  $\lambda T. (\forall x. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X)$ ;
```

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:

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

Type-Level Abstraction & Application

Syntax

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

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: $S \equiv T$

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

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

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

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

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

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

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

Issue 2: Errors in Type-Level Computation

Example

Consider $(\lambda X. X X) \text{Nat}$. The type evaluates to 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**.

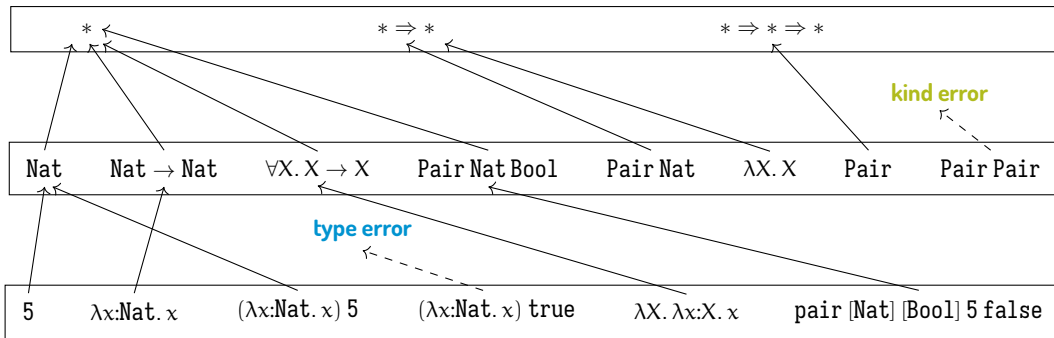
What can characterize **types**?

Kinds: “Types of Types”

Kinds characterize **types**.

- $*$ proper types (e.g., 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 Γ ”

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

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

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

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

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

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

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 ”

$$\begin{array}{c}
 \frac{}{T \Rightarrow T} \qquad \frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 \rightarrow S_2 \Rightarrow T_1 \rightarrow T_2} \qquad \frac{S_2 \Rightarrow T_2}{\lambda X::K_1. S_2 \Rightarrow \lambda X::K_1. T_2} \qquad \frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 S_2 \Rightarrow T_1 T_2} \\
 \\
 \frac{S_{12} \Rightarrow T_{12} \quad S_2 \Rightarrow T_2}{(\lambda X::K_{11}. S_{12}) S_2 \Rightarrow [X \mapsto T_2] T_{12}}
 \end{array}$$

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

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

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 =  $\lambda Y::^*.$  ( $\mu X.$  <nil:Unit,cons:{Y,X}>);
```

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

Parallel Reduction

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

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_ω : The Combination of System F and λ_ω



Syntax

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

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

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

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

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

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

Kinding and Type Equivalence

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

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

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

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

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**!

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

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


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_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 :

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

Example

- System F_1 is the simply-typed lambda-calculus λ_{\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 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

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

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

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

Natural-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} :

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

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

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: $\Gamma \vdash S \equiv T :: K$

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

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

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

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

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

$$\begin{aligned} & \equiv \\ & [Y \mapsto \text{ITER } T_0 \text{ WITH ZERO} \Rightarrow T_1 \mid \text{SUCC} \Rightarrow Y. T_2] T_2 :: K \end{aligned}$$

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

`tc`case M of `ZERO` \Rightarrow `t1` | `SUCC` M' \Rightarrow `t2`

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

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

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

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

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

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**.

$$S ::= \{\alpha :: \mathbb{N} \mid \emptyset\} \mid \{\emptyset\}$$

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

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

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

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

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 ℓ and term t by $\{\ell = t \mid r\}$.
$$\text{origin} = \{x = 0 \mid \{y = 0 \mid \{\}\}\};$$
$$\text{origin3} = \{z = 0 \mid \text{origin}\};$$
$$\text{named} = \lambda s. \lambda r. \{\text{name} = s \mid r\};$$
- **Selection:** The selection operation $r.\ell$ selects the value of a label ℓ from a record r .
$$\text{distance} = \lambda p. \text{sqrt} ((p.x * p.x) + (p.y * p.y));$$
$$\text{distance} (\text{named } "2d" \text{ origin}) + \text{distance } \text{origin3};$$
- **Restriction:** The restriction operation $r - \ell$ removes a label ℓ from a record r .
$$\text{update_name} = \lambda r. \lambda s. \{\text{name} = s \mid r - \text{name}\};$$
$$\text{rename_name_nn} = \lambda r. \{\text{nn} = r.\text{name} \mid r - \text{name}\};$$

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.

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

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

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:

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

Well-Typed Record Operations

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

Row Equivalence

Question

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

Type-Level Row Equivalence

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

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

Example

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

$$\frac{}{\emptyset \vdash \{x = 0 \mid \{y = \text{true} \mid \{\}\}\}.y : \text{Bool}}$$

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 ℓ 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 ℓ .

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

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

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

Type-Level Labels



Question

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

Principle

$$K ::= * \mid K \Rightarrow K \mid \text{row} \mid \text{label}$$
$$T ::= X \mid \lambda X::K. T \mid T T \mid T \rightarrow T \mid \forall X::K. T \mid \{\exists X::K, T\} \mid () \mid (\textcolor{red}{T} : T \mid T) \mid \{T\} \mid \langle T \rangle \mid \textcolor{red}{\#} \ell$$

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

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

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

Example: A Generic Table Formatter

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

► Meta :: $\ast \Rightarrow \ast$

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

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

► Folder :: $row \Rightarrow \ast$

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

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

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

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

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

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

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

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

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

Question

Extend System F_{ω} 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 $t \longrightarrow t'$.