Higher-Order Types

Klaus Ostermann
University of Tübingen

Motivation:
Limitations of first-order types in Scala

```
trait Iterable[T] {
  def filter(p: T ⇒ Boolean): Iterable[T]
  def remove(p: T ⇒ Boolean): Iterable[T] = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T] {
  def filter(p: T ⇒ Boolean): List[T]
  override def remove(p: T ⇒ Boolean): List[T]
    = filter (x ⇒ !p(x))
}
```

legend: – copy/paste →
redundant code

From “Generics of a Higher Kind” by Moors et al, 2008

Solution using higher-order types

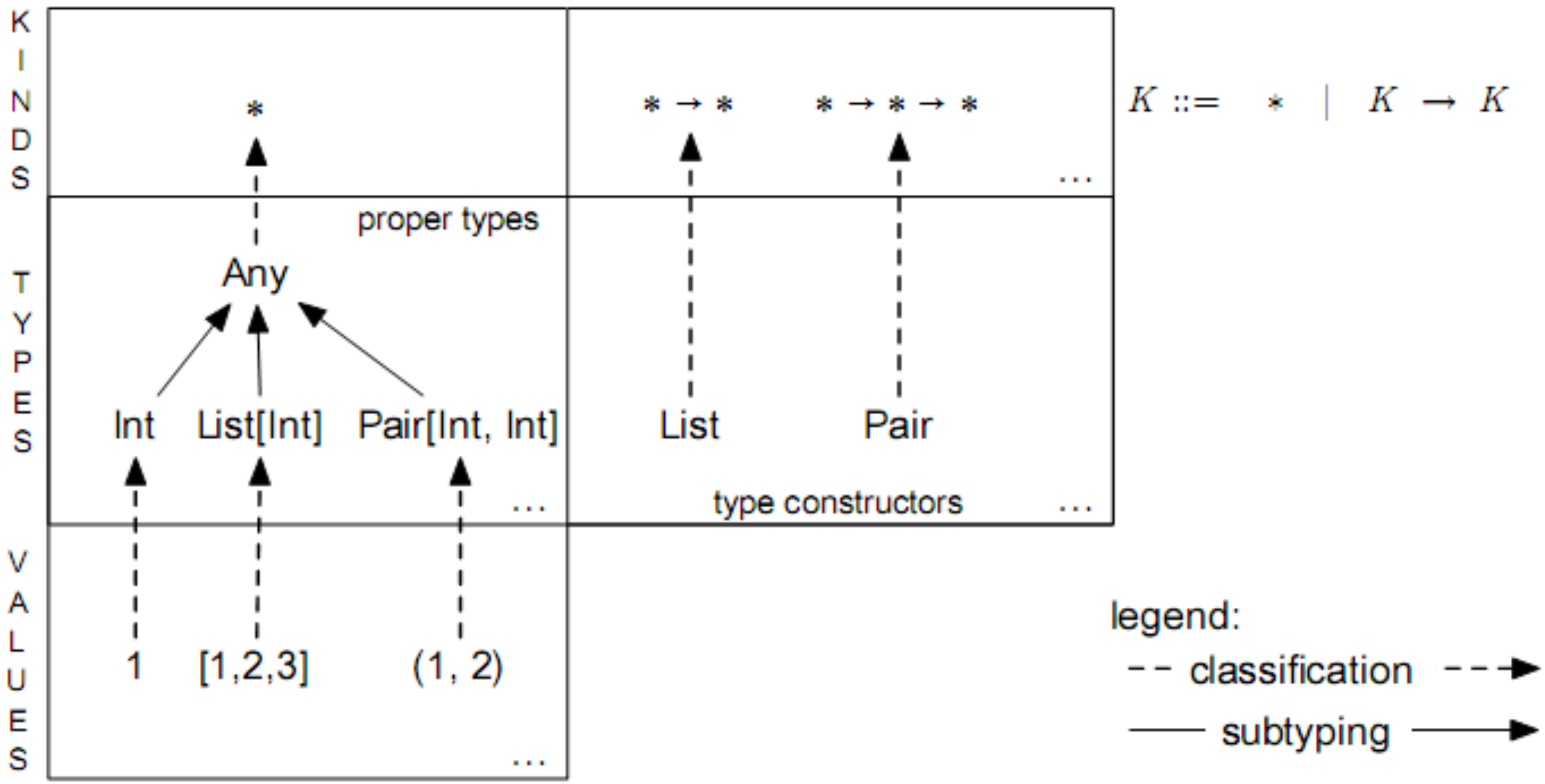
```
trait Iterable[T, Container[X]] {
  def filter(p: T ⇒ Boolean): Container[T]
  def remove(p: T ⇒ Boolean): Container[T] = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T, List]
```

legend: - abstraction

— instantiation

Universes in Scala



Motivation: Higher-Order types in Haskell

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

class Functor f where -- f must have kind *->*
  fmap                  :: (a -> b) -> f a -> f b

instance Functor Tree where
    fmap f (Leaf x)        = Leaf   (f x)
    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)

addone :: Tree Int -> Tree Int
addone t = fmap (+ 1) t

-- instance Functor Integer where → kind error
```

Adding kinds to simply-typed LC

- Syntax
  - Syntax of terms and values unchanged

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*

Evaluation

- Like in simply-typed LC, no changes

Kinding rules

Kinding $\boxed{\Gamma \vdash T :: K}$

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

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \qquad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \qquad \text{(K-ARROW)}$$

This is basically a copy of the STLC “one level up”!

Typing Rules

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}} \quad \text{(T-APP)}$$

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

- We need a notion of type equivalence!
- T-Eq is not syntax-directed, like the subsumption rule in subtyping

Type Equivalence

Type equivalence $\boxed{S \equiv T}$

$$T \equiv T \qquad \text{(Q-Refl)}$$

$$\frac{T \equiv S}{S \equiv T} \qquad \text{(Q-Symm)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \qquad \text{(Q-Trans)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \qquad \text{(Q-Arrow)}$$

$$\frac{S_2 \equiv T_2}{\lambda X{::}K_1 . S_2 \equiv \lambda X{::}K_1 . T_2} \qquad \text{(Q-Abs)}$$

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

$$(\lambda X{::}K_{11} . T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \qquad \text{(Q-AppAbs)}$$

Nice, but…

- Adding kinds to STLC is not really useful.
- A program in this language can trivially be rewritten to STLC w/o kinds by just normalizing every type expression in place.
- To gain real expressive power we need universal types, too.
- Let’s hack System F, then!

Adding kinds to System F – a.k.a. F_ω

- Syntax of terms and values

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX::K.t	*type abstraction*
	t [T]	*type application*
v ::=		*values:*
	λx:T.t	*abstraction value*
	λX::K.t	*type abstraction value*

Adding kinds to System F – a.k.a. F_ω

- Syntax of types, contexts, kinds

T ::=		*types:*
	X	*type variable*
	$T \to T$	*type of functions*
	$\forall X :: K . T$	*universal type*
	$\lambda X :: K . T$	*operator abstraction*
	$T\ T$	*operator application*
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*

Adding kinds to System F – a.k.a. F_ω

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

$$(\lambda X{::}K_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TappTabs)}$$

Adding kinds to System F – a.k.a. F_ω

Kinding $\boxed{\Gamma \vdash T :: K}$

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

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \quad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad \text{(K-ARROW)}$$

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

Adding kinds to System F – a.k.a. F_ω

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

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

$$\frac{\Gamma \vdash t_1 : \forall X :: K_{11} . T_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

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

Adding kinds to System F – a.k.a. F_ω

Type equivalence $\boxed{S \equiv T}$

$$T \equiv T \quad \text{(Q-Refl)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-Symm)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \quad \text{(Q-Trans)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-Arrow)}$$

$$\frac{S_2 \equiv T_2}{\forall X :: K_1 . S_2 \equiv \forall X :: K_1 . T_2} \quad \text{(Q-All)}$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1 . S_2 \equiv \lambda X :: K_1 . T_2} \quad \text{(Q-Abs)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-App)}$$

$$(\lambda X :: K_{11} . T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-AppAbs)}$$

Higher-Order Existentials

- F_ω with existential types has some interesting uses
- Example: Abstract data type for pairs
  - want to hide choice of Pair type constructor

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

Higher-Order Existentials

- Example, continued

```
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
```

Using the Pair ADT:

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

Higher-Order Existentials, formally

New syntactic forms

$T ::= \ldots$ *types:*

$\{\exists X :: K, T\}$ *existential type*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\texttt{let } \{X,x\}=(\{*T_{11}, v_{12}\} \texttt{ as } T_1) \texttt{ in } t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \quad \text{(E-UnpackPack)}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{*T_{11}, t_{12}\} \texttt{ as } T_1 \longrightarrow \{*T_{11}, t'_{12}\} \texttt{ as } T_1} \quad \text{(E-Pack)}$$

New kinding rules $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \{\exists X :: K_1, T_2\} :: *} \quad \text{(K-Some)}$$

New type equivalence rules $\boxed{S \equiv T}$

$$\frac{S_2 \equiv T_2}{\{\exists X :: K_1, S_2\} \equiv \{\exists X :: K_1, T_2\}} \quad \text{(Q-Some)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \qquad \Gamma \vdash \{\exists X :: K_1, T_2\} :: *}{\Gamma \vdash \{*U, t_2\} \texttt{ as } \{\exists X :: K_1, T_2\} : \{\exists X :: K_1, T_2\}} \quad \text{(T-Pack)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X :: K_{11}, T_{12}\} \qquad \Gamma, X :: K_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } \{X,x\}=t_1 \texttt{ in } t_2 : T_2} \quad \text{(T-Unpack)}$$

Algorithmic Type-Checking for F_ω

- Kinding relation is easily decidable (syntax-directed)
- T-Eq must be removed, similarly to T-Sub in the system with subtyping
- Two critical points for the now missing T-Eq rule:
  - First premise of T-App and T-TApp requires type to be of a specific form
  - In the second premise of T-App we must match two types

Algorithmic Type-Checking for F_ω

- Idea: Equivalence checking by normalization
- Normalization = Reduction to normal form
- In our case: Use directed variant of type equivalence relation, reduce until normal form reached
- In practical languages, a slightly weaker form of equivalence checking is used: Normalization to Weak Head Normal Form (WHNF)
- A term is in WHNF if its top-level constructor is not reducible
  - i.e. stop if top-level constructor is not an application