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

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

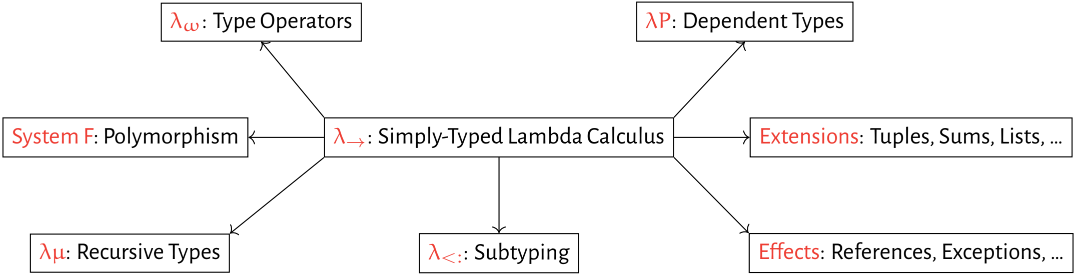
Chap 30: Higher-Order Polymorphism

System F_ω

Examples

Properties

We Have Studied ...



Remark

- Different combinations of features lead to different languages.
- Some combinations turn out to be very **tricky**!
- This chapter studies the combination of polymorphism and type operators.

F_ω

The Combination of System F and λ_ω

Syntax and Evaluation

Syntax

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

Evaluation

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

Typing, Kinding, and Type Equivalence

Typing

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

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

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

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

Kinding and Type Equivalence

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

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \{\exists X{::}K_1, T_2\} :: *} \text{ K-SOME}$$

$$\frac{S_2 \equiv T_2}{\forall X{::}K_1 . S_2 \equiv \forall X{::}K_1 . T_2} \text{ Q-ALL}$$

$$\frac{S_2 \equiv T_2}{\{\exists X{::}K_1, S_2\} \equiv \{\exists X{::}K_1, T_2\}} \text{ Q-SOME}$$

Examples

Review: Abstract Data Types (ADTs)

Definition

An abstract data type (ADT) consists of

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

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

Abstract Type Operators

Question

We want to implement an ADT of pairs.

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

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

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

Abstract Type Operators

Example

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

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

More Examples

Option: Combination with Variants

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

List: Combination with Variants, Tuples, and Recursive Types

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

More Examples

Queue: Implementing a Queue using Two Lists

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

Properties

Type Equivalence and Reduction

Review: Parallel Reduction ($S \Rrightarrow T$)

$$\frac{}{T \Rrightarrow T} \text{ QR-Refl} \qquad \frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1 \to S_2 \Rrightarrow T_1 \to T_2} \text{ QR-Arrow} \qquad \frac{S_2 \Rrightarrow T_2}{\lambda X{::}K_1 . S_2 \Rrightarrow \lambda X{::}K_1 . T_2} \text{ QR-Abs}$$

$$\frac{S_1 \Rrightarrow T_1 \qquad S_2 \Rrightarrow T_2}{S_1\ S_2 \Rrightarrow T_1\ T_2} \text{ QR-App} \qquad \frac{S_{12} \Rrightarrow T_{12} \qquad S_2 \Rrightarrow T_2}{(\lambda X{::}K_{11} . S_{12})\ S_2 \Rrightarrow [X \mapsto T_2]T_{12}} \text{ QR-AppAbs}$$

$$\frac{S_2 \Rrightarrow T_2}{\forall X{::}K_1 . S_2 \Rrightarrow \forall X{::}K_1 . T_2} \text{ QR-All} \qquad \frac{S_2 \Rrightarrow T_2}{\{\exists X{::}K_1 , S_2\} \Rrightarrow \{\exists X{::}K_1 , T_2\}} \text{ QR-Some}$$

Proposition

- If $S \Rrightarrow^* U$ and $T \Rrightarrow^* U$ for some U, then $S \equiv T$. (Corollary of Lemma 30.3.5)
- If $S \equiv T$, then there is some U such that $S \Rrightarrow^* U$ and $T \Rrightarrow^* U$. (Corollary 30.3.11)

Preservation

Observation

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

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

Preservation of Shapes (for Arrows)

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

Inversion (for Arrows)

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

Theorem (30.3.14)

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

Progress

Canonical Forms (for Arrows)

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

Theorem (30.3.16)

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

Decidability

Observation

The kinding relation is decidable, because kinding is a "simply-typed lambda-calculus" at the type level.

Suppose that we remove the one structural rule (T-Eq).

Example

$$\frac{\Gamma \vdash t_1 : (\lambda X{::}{*}.\, X \to X)\ \mathsf{Nat} \qquad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash t_1\ t_2 : \mathsf{Nat}}\ \textsc{T-App}$$

We need to rewrite the type of t_1 to bring an arrow to the outside.

Solution

We can **reduce** the type of t_1 to a normal form, e.g., $(\lambda X{::}{*}.\, X \to X)\ \mathsf{Nat} \Rrightarrow^* \mathsf{Nat} \to \mathsf{Nat}$.
Parallel reduction always normalizes for well-kinded types, by a similar argument for the normalization of simply-typed lambda-calculus (Chapter 12).

Decidability

Aside (Weak-Head Reduction)

$$\frac{T_1 \Rrightarrow_{wh} T_1'}{T_1\ T_2 \Rrightarrow_{wh} T_1'\ T_2}\ \text{WH-App} \qquad \frac{}{(\lambda X{::}K_{11}.\,T_{12})\ T_2 \Rrightarrow_{wh} [X \mapsto T_2]T_{12}}\ \text{WH-AppAbs}$$

Weak-head reduction only reduces leftmost, outermost redexes and stops at a concrete constructor (e.g., arrows).

$$\begin{aligned} & (\lambda X{::}*.\,\mathrm{Id}\,(X \to X))\ (\mathrm{Id}\,\mathrm{Nat}) \\ \Rrightarrow_{wh}\ & \mathrm{Id}\,((\mathrm{Id}\,\mathrm{Nat}) \to (\mathrm{Id}\,\mathrm{Nat})) \\ =\ & (\lambda Y{::}*.\,Y)\,((\mathrm{Id}\,\mathrm{Nat}) \to (\mathrm{Id}\,\mathrm{Nat})) \\ \Rrightarrow_{wh}\ & (\mathrm{Id}\,\mathrm{Nat}) \to (\mathrm{Id}\,\mathrm{Nat}) \\ \not\Rrightarrow_{wh}\ & \cdot \end{aligned}$$

Decidability

Example

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-App}$$

We need to check the equivalence between T_2 and T_{11}.

Solution

We can again **reduce** both T_2 and T_{11} to their normal forms.
For example, $T_2 \Rrightarrow^* S_1$ and $T_{11} \Rrightarrow^* S_2$ where S_1 and S_2 are identical (modulo the names of bound variables).

Fragments of 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**:

$$\mathcal{K}_1 = \varnothing$$
$$\mathcal{K}_{i+1} = \{*\} \cup \{J \Rightarrow K \mid J \in \mathcal{K}_i \wedge K \in \mathcal{K}_{i+1}\}$$
$$\mathcal{K}_\omega = \bigcup_{1 \leqslant i} \mathcal{K}_i$$

Example

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

Design Principles of Programming Languages

Key Takeaways

Principle

- The uses of type systems **go far beyond** their role in detecting errors.
- Type systems offer **crucial support** for programming: **abstraction, safety, efficiency, ...**
- Language design shall go **hand-in-hand** with type-system design.