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 23: Universal Types

Polymorphism

System F

Examples

Properties

Type Reconstruction

Abstraction Principle

Example

Suppose we want to define a function `double` that applies its 1st argument twice to its 2nd:

```
doubleNat = λf:Nat→Nat. λa:Nat. f (f a);
doubleRcd = λf:{l:Bool}→{l:Bool}. λa:{l:Bool}. f (f a);
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λa:Nat→Nat. f (f a);
```

They share the same behavior and the same body term.

Principle (Abstraction)

Each significant piece of functionality in a program should be implemented in just one place in the source code.

```
double = λX. λf:X→X. λa:X. f (f a);
```

Polymorphism

Parametric Polymorphism

Allow a single piece of code to be typed "generically" using **type variables**.

```
id = λX. λx:X. x;
▶ id : ∀X. X → X
```

Ad-hoc Polymorphism

Allow a polymorphic value to exhibit **different behaviors** when "viewed" at different types.

- **Overloading**: `1+2` `1.0+2.0` `"we"+"you"`
- **Typeclasses**: `(+) :: Num a => a -> a -> a`

Subtype Polymorphism

Allow a single term to have many types using the rule of **subsumption**: $\dfrac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$.

System F

The Most Powerful Form of Parametric Polymorphism

System F

Some Historical Accounts

- System F was introduced by Girard (1972) in the context of proof theory.[1]
- System F was independently developed by Reynolds (1974) in the context of programming languages.[2]
- Reynolds called System F the **polymorphic lambda-calculus**.

Principle

System F is a straightforward extension of $\lambda_{\to}$.

- In $\lambda_{\to}$, we use $\lambda x{:}T.\ t$ to abstract **terms** out of terms.
- In System F, we introduce $\lambda X.\ t$ to abstract **types** out of terms.

[1] J.-Y. Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis. Université Paris 7.

[2] J. C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, 408–423. DOI: 10.1007/3-540-06859-7_148.

Syntax and Evaluation

Syntax

$$t ::= \ldots \mid \lambda X.\, t \mid t\,[T]$$
$$v ::= \ldots \mid \lambda X.\, t$$

Evaluation

$$\frac{t_1 \longrightarrow t_1'}{t_1\,[T_2] \longrightarrow t_1'\,[T_2]}\ \text{E-TApp} \qquad \frac{}{(\lambda X.\, t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]t_{12}}\ \text{E-TappTabs}$$

Example

Recall that we define $id \stackrel{\text{def}}{=} \lambda X.\, \lambda x{:}X.\, x$. Thus

$$id\,[\text{Nat}] \longrightarrow [X \mapsto \text{Nat}](\lambda x{:}X.\, x) = \lambda x{:}\text{Nat}.\, x$$

Types, Type Contexts, and Typing

Types and Type Contexts

$$T ::= X \mid T \to T \mid \forall X.T$$
$$\Gamma ::= \varnothing \mid \Gamma, x : T \mid \Gamma, X$$

Typing

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.\, t_2 : \forall X.T_2}\ \text{T-TAbs} \qquad \frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}\ \text{T-TApp}$$

Example

$$\cfrac{\cfrac{\cfrac{}{X, x : X \vdash x : X}\ \text{T-Var}}{X \vdash \lambda x{:}X.\, x : X \to X}\ \text{T-Abs}}{\varnothing \vdash \lambda X.\, \lambda x{:}X.\, x : \forall X.X \to X}\ \text{T-TAbs}$$

Examples

Polymorphic Functions

Polymorphic Lists

Church Encodings

Polymorphic Functions

```
id = λX. λx:X. x;
▶ id : ∀X. X → X
id [Nat] 0;
▶ 0 : Nat

double = λX. λf:X→X. λa:X. f (f a);
▶ double : ∀X. (X→X) → X → X
double [Nat] (λx:Nat. succ(succ(x))) 3;
▶ 7 : Nat

selfApp = λx:∀X.X→X. x [∀X.X→X] x;
▶ selfApp : (∀X. X→X) → (∀X. X→X)

quadruple = λX. double [X→X] (double [X]);
▶ quadruple : ∀X. (X→X) → X → X
```

Polymorphic Lists

List as a Type Constructor

We assume the language has the following primitives:

```
nil : ∀X. List X
cons : ∀X. X → List X → List X
```

```
isnil : ∀X. List X → Bool
head : ∀X. List X → X
tail : ∀X. List X → List X
```

Example

```
map = λX. λY. λf: X→Y.
        (fix (λm: (List X) → (List Y).
                λl: List X.
                  if isnil [X] l then nil [Y]
                                 else cons [Y] (f (head [X] l)) (m (tail [X] l))));
▶ map : ∀X. ∀Y. (X→Y) → List X → List Y
```

Polymorphic Lists

Question (Exercise 23.4.3)

Using `map` as a model, write a polymorphic list-reversing function: `reverse : ∀X. List X → List X`.

Solution

```
rev_append = λX. fix (λra:(List X)→(List X)→(List X). λl1:(List X). λl2:(List X).
                       if isnil [X] l1 then l2
                       else ra (tail [X] l1) (cons [X] (head [X] l1) l2));
▶ rev_append : ∀X. List X → List X → List X

reverse = λX. λl: List X. rev_append [X] l (nil [X]);
▶ reverse : ∀X. List X → List X
```

Polymorphic Lists

List as a Type Constructor

We have assumed the language has the following primitives:

```
nil  : ∀X. List X                     isnil : ∀X. List X → Bool
cons : ∀X. X → List X → List X        head  : ∀X. List X → X
                                      tail  : ∀X. List X → List X
```

Aside

We can use recursive types to implement List, e.g.,

```
nil = λX. <nil=Unit> as (μT. <nil:Unit, cons:{X,T}>);
▶ nil : ∀X. μT. <nil:Unit, cons:{X,T}>
```

Church Encodings: Booleans

Remark

In Chapter 5.2, we saw that booleans, numbers, lists, etc. can be encoded as functions.

```
tru = λt. λf. t;            fls = λt. λf. f;
```

```
CBool = ∀X. X→X→X;

tru = (λX. λt:X. λf:X. t) as CBool;
▶ tru : CBool
fls = (λX. λt:X. λf:X. f) as CBool;
▶ fls : CBool
```

Question

Why does the definition `CBool` characterize booleans?

Church Encodings: Booleans

Typing Rules for Booleans

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ \textsc{T-False} \qquad \frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}\ \textsc{T-If}$$

Observation

The definition `CBool = ∀X. X→X → X` encodes the typing rule (T-IF).

PRINCIPLE

Encode typing rules for **destructors** as polymorphic function types.

Example

Using booleans are **directly applying** their corresponding polymorphic functions.

```
test = λY. λt1:CBool. λt2:Y. λt3:Y. t1 [Y] t2 t3;
▶ test : ∀Y. CBool → Y → Y → Y
```

Church Encodings: Booleans

Question

Can `test` be used as conditional expressions?

Observation

Under call-by-value, `test` [T] t_1 t_2 t_3 (where T is the type of t_2, t_3) evaluates **both** t_2 and t_3.

Solution: Dummy Abstractions

```
CBool = ∀X. (Unit→X)→(Unit→X)→X;
test = λY. λt1:CBool. λt2:(Unit→Y). λt3:(Unit→Y). t1 [Y] t2 t3;
▶ test: ∀Y. CBool → (Unit→Y) → (Unit→Y) → Y
```

We can encode `if` t_1 `then` t_2 `else` t_3 as `test` [T] t_1 (λ_:Unit. t_2) (λ_:Unit. t_3).

Question

Write down the encodings for `true` and `false` with dummy abstractions.

Church Encodings: Sums

Question

Recall that with sum types, we can define the boolean type as Unit + Unit and literals as inl unit, inr unit. Can you define the encodings of general sum types $T_1 + T_2$?

Hint: write down the typing rule for **using** sum types.

Solution

Let the type constructor $T_1 + T_2$ be defined as

$$\forall X.\ (T_1 \to X) \to (T_2 \to X) \to X$$

Then the constructors and the destructor for $T_1 + T_2$ can be defined as follows:

```
inl = λv:T1. (λX. λl:(T1→X). λr:(T2→X). l v) as (T1+T2);
▶ inl : T1 → (T1+T2)
inr = λv:T2. (λX. λl:(T1→X). λr:(T2→X). r v) as (T1+T2);
▶ inr : T2 → (T1+T2)
```

Church Encodings: Sums

```
test = λY. λt1:(T₁ + T₂). λt2:(T₁→Y). λt3:(T₂→Y). t1 [Y] t2 t3;
▶ test : ∀Y. (T₁ + T₂) → (T₁→Y) → (T₂→Y) → Y
```

Question

How to encode `case` t_1 `of` `inl` $x \Rightarrow t_2$ | `inr` $x \Rightarrow t_3$?

Solution

`test` $[T]\ (\lambda x{:}T_1 .\ t_2)\ (\lambda x{:}T_2 .\ t_3)$.

Church Encodings: Numbers

Remark (Church Numerals)

```
c₀ = λs. λz. z;                c₂ = λs. λz. s (s z);
c₁ = λs. λz. s z;              c₃ = λs. λz. s (s (s z));
```

```
CNat = ∀X. (X→X) → X → X;
c₀ = (λX. λs:X→X. λz:X. z) as CNat;
▶ c₀ : CNat
c₁ = (λX. λs:X→X. λz:X. s z) as CNat;
▶ c₁ : CNat
```

Question

What are the typing rules for **using** numbers, with respect to the polymorphic type CNat?

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma, y : T \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \textbf{usenat}\ t_1\ \textbf{with}\ \mathtt{succ}(y) \Rightarrow t_2 \mid \mathtt{zero} \Rightarrow t_3 : T}$$

Church Encodings: Numbers

```
csucc = λn:CNat. (λX. λs:X→X. λz:X. s (n [X] s z)) as CNat;
▶ csucc : CNat → CNat

cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
▶ cplus : CNat → CNat → CNat
```

Remark

We do **not** use recursion to define `cplus`!

Question

Define a function `cmult` that calculates the product of two numbers.

Church Encodings: Lists

Remark

We have seen `List T` as a primitive type or as a recursive type. Can we encode it in pure System F?

Principle

Encode typing rules for **destructors** as polymorphic function types.

$$\frac{\Gamma \vdash t_1 : \mathtt{List}\,T \qquad \Gamma, h : T, y : S \vdash t_2 : S \qquad \Gamma \vdash t_3 : S}{\Gamma \vdash \textbf{uselist}\; t_1\; \textbf{with}\; \mathtt{cons}(h, y) \Rightarrow t_2 \mid \mathtt{nil} \Rightarrow t_3 : S}$$

```
List T = ∀X. (T→X→X) → X → X;

nil = λT. (λX. λc:(T→X→X). λn:X. n) as List T;
▶ nil : ∀T. List T
cons = λT. λhd:T. λtl:(List T).
          (λX. λc:(T→X→X). λn:X. c hd (tl [X] c n)) as List T;
▶ cons : ∀T. T → List T → List T
```

Church Encodings: Lists

```
isnil = λT. λl:(List T). l [Bool] (λ_:T. λ_:Bool. false) true;
▶ isnil : ∀T. List T → Bool
head = λT. λl:(List T). l [T] (λhd:T. λ_:T. hd) (diverge [T] unit);
▶ head : ∀T. List T → T
```

Question

Can you define a function `sum : List Nat -> Nat` **without** using **`fix`**?

Solution

```
sum = λl:(List Nat). l [Nat] (λhd:Nat. λtl:Nat. hd + tl) 0;
▶ sum : List Nat → Nat
```

Church Encodings: Inductive Types

Aside

Recall the rule for iteration on `NatList` (from the lecture on recursive types):

$$\frac{\Gamma \vdash t_1 : \texttt{NatList} \qquad \Gamma, x : \texttt{<nil} : \texttt{Unit}, \texttt{cons} : \{\texttt{Nat}, S\}\texttt{>} \vdash t_2 : S}{\Gamma \vdash \textbf{iter}\ [\texttt{NatList}]\ t_1\ \textbf{with}\ x.t_2 : S}$$

The rule is very similar to the aforementioned rule that is **internalized** by $\forall$ X. (Nat→X → X) → X → X:

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T \qquad \Gamma, h : T, y : S \vdash t_2 : S \qquad \Gamma \vdash t_3 : S}{\Gamma \vdash \textbf{uselist}\ t_1\ \textbf{with}\ \texttt{cons}(h, y) \Rightarrow t_2 \mid \texttt{nil} \Rightarrow t_3 : S}$$

In a similar way, we can encode general inductive (and also coinductive) types in System F.

Church Encodings: Pairs

Typing Rules for Pairs

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \text{ T-Proj1} \qquad \frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \text{ T-Proj2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12} \qquad \Gamma, x : T_{11}, y : T_{12} \vdash t_2 : S}{\Gamma \vdash \texttt{let}\ \{x, y\} = t_1\ \texttt{in}\ t_2 : S} \text{ T-LetPair}$$

```
Pair T1 T2 = ∀X. (T1→T2→X) → X
```

Church Encodings: Pairs

```
Pair T1 T2 = ∀X. (T1→T2→X) → X;

pair = λT1. λT2. λx:T1. λy:T2. (λX. λp:(T1→T2→X). p x y) as Pair T1 T2;
▶ pair : ∀T1. ∀T2. T1 → T2 → Pair T1 T2

fst = λT1. λT2. λp:(Pair T1 T2). p [T1] (λx:T1. λ_:T2. x);
▶ fst : ∀T1. ∀T2. Pair T1 T2 → T1
snd = λT1. λT2. λp:(Pair T1 T2). p [T2] (λ_:T1. λy:T2. y);
▶ snd : ∀T1. ∀T2. Pair T1 T2 → T2
```

Properties

Preservation, Progress, Normalization, Parametricity, Impredicativity

Basic Properties

Theorem (Preservation)

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

Theorem (Progress)

If t is a closed, well-typed term, then either t is a value or there is some t' with $t \longrightarrow t'$.

Theorem (Normalization)

Well-typed System-F terms are normalizing, i.e., the evaluation of every well-typed term terminates.

Question (Homework)

Exercises 23.5.1 or 23.5.2: prove preservation or progress of System F.

Parametricity

Observation

Polymorphic types severely **constrain** the behavior of their elements.

- If $\varnothing \vdash t : \forall X.X \to X$, then t **is** (essentially) the identity function.
- If $\varnothing \vdash t : \forall X.X \to X \to X$, then t **is** (essentially) either `tru` ($\lambda X.\ \lambda t{:}X.\ \lambda f{:}X.\ t$) or `fls` ($\lambda X.\ \lambda t{:}X.\ \lambda f{:}X.\ f$).

Definition (Parametricity)

Properties of a term that can be proved **knowing only its type** are called parametricity.
Such properties are often called **free theorems** as they come from typing **for free**.

Aside (Read More)

- J. C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing*, 513–523.
- P. Wadler. 1989. Theorems for free! In *Functional Programming Languages and Computer Architecture* (FPCA'89), 347–359. DOI: 10.1145/99370.99404.

Parametricity: The Idea

Proposition

For any closed term $id : \forall X.X \to X$, for any type T and any property $\mathcal{P}$ of the type T, if $\mathcal{P}$ holds of $t : T$, then $\mathcal{P}$ holds of $id\ [T]\ t : T$.

Remark

$\mathcal{P}$ needs to be closed under **head expansion**, i.e., if $t \longrightarrow t'$ and $\mathcal{P}$ holds of $t' : T$, then $\mathcal{P}$ also holds of $t : T$.

Example

Fix $t_0 : T$. Consider $\mathcal{P}_{t_0}$ that holds of $t_1 : T$ iff t_1 is equivalent to t_0 (i.e., $t_1 =_\beta t_0$).
Obviously $\mathcal{P}_{t_0}$ holds of t_0 itself.
By the proposition above, $\mathcal{P}_{t_0}$ holds of $id\ [T]\ t_0$.
Thus, $id\ [T]\ t_0$ is equivalent to t_0.

Parametricity: The Idea

Proposition

For any closed term $b : \forall X.X \to X \to X$, for any type T and any property $\mathcal{P}$ of type T, if $\mathcal{P}$ holds of $m : T$ and of $n : T$, then $\mathcal{P}$ holds of $b\ [T]\ m\ n$.

Example

Fix $t_0 : T$ and $t_1 : T$. Consider $\mathcal{P}_{t_0,t_1}$ that holds of $t_2 : T$ iff t_2 is equivalent to either t_0 or t_1.
Obviously $\mathcal{P}_{t_0,t_1}$ holds of both t_0 and t_1.
By the proposition above, $\mathcal{P}_{t_0,t_1}$ holds of $b\ [T]\ t_0\ t_1$.
Thus, $b\ [T]\ t_0\ t_1$ is equivalent to either t_0 or t_1.

Parametricity: The Idea

Proposition (Unary)

For any closed term $id : \forall X.X \to X$, for any type T and any property $\mathcal{P}$ of the type T, if $\mathcal{P}$ holds of $t : T$, then $\mathcal{P}$ holds of $id\ [T]\ t : T$.

Proposition (Binary)

For any closed term $id : \forall X.X \to X$, for any types T, T' and any **binary relation** $\mathcal{R}$ between T and T', if $\mathcal{R}$ **relates** $t : T$ to $t' : T'$, then $\mathcal{R}$ **relates** $id\ [T]\ t : T$ to $id\ [T']\ t' : T'$.

Example (A Free Theorem from $id : \forall X.X \to X$)

Let $g : T \to T'$ be an arbitrary function. For any $t : T$, it holds that $id\ [T']\ (g\ t)$ is equivalent to $g\ (id\ [T]\ t)$.

Impredicativity

Remark (Russell's Paradox)

Let R be the set of sets that are not a member of themselves, i.e.,

$$R \overset{\text{def}}{=} \{x \mid x \notin x\},$$

then we can see that $R \in R \iff R \notin R$, which yields a paradox.

Observation

The paradox comes of letting the x be the very "set" R that is being defined by the membership condition. Intuitively, impredicativity means **self-referencing definitions**.

System F is Impredicative

The type variable X in the type $T = \forall X.X \to X$ ranges over all types, **including** T **itself**. Fortunately, Girard shows that System F is **logically consistent**.

Type Reconstruction

Erasure & Type Reconstruction

$$erase(x) \stackrel{\text{def}}{=} x$$

$$erase(\lambda x{:}T_1\,.\,t_2) \stackrel{\text{def}}{=} \lambda x.\, erase(t_2)$$

$$erase(t_1\ t_2) \stackrel{\text{def}}{=} erase(t_1)\ erase(t_2)$$

$$erase(\lambda X.\, t_2) \stackrel{\text{def}}{=} erase(t_2)$$

$$erase(t_1\ [T_2]) \stackrel{\text{def}}{=} erase(t_1)$$

Definition (Type Reconstruction)

Given an untyped term m, whether we can find some well-typed term t such that $erase(t) = m$.

Theorem (Wells, 1994[3])

Type reconstruction for System F is **undecidable**.

[3] J. B. Wells. 1994. Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable. In *Logic in Computer Science* (LICS'94), 176–185. DOI: 10.1109/LICS.1994.316068.

Partial Erasure & Type Reconstruction

$$erase_p(x) \stackrel{\text{def}}{=} x$$

$$erase_p(\lambda x{:}T_1\,.\,t_2) \stackrel{\text{def}}{=} \lambda x{:}T_1\,.\,erase_p(t_2)$$

$$erase_p(\lambda X.\,t_2) \stackrel{\text{def}}{=} \lambda X.\,erase_p(t_2)$$

$$erase_p(t_1\ [T_2]) \stackrel{\text{def}}{=} erase_p(t_1)\ [\,]$$

Theorem (Boehm 1985[4], 1989[5])

It is **undecidable** whether, given a closed term s in which type applications are marked but the arguments are omitted, there is some well-typed System-F term t such that $erase_p(t) = s$.

Question

Is this the end of the story?

---

[4] H.-J. Boehm. 1985. Partial Polymorphic Type Inference is Undecidable. In *Symp. on Foundations of Computer Science* (SFCS'85), 339–345. DOI: 10.1109/SFCS.1985.44.

[5] H.-J. Boehm. 1989. Type Inference in the Presence of Type Abstraction. In *Prog. Lang. Design and Impl.* (PLDI'89), 192–206. DOI: 10.1145/73141.74835.

Fragments of System F

Prenex Polymorphism

- Type variables range only over quantifier-free types (**monotypes**).
- Quantified types (**polytypes**) are not allows to appear on the left-hand sides of arrows.

Rank-2 Polymorphism

A type is said to be of rank 2 if no path from its root to a $\forall$ quantifier passes to the left of 2 or more arrows.

$$(\forall X.X \to X) \to \mathsf{Nat} \quad ✓$$

$$\mathsf{Nat} \to ((\forall X.X \to X) \to (\mathsf{Nat} \to \mathsf{Nat})) \quad ✓$$

$$((\forall X.X \to X) \to \mathsf{Nat}) \to \mathsf{Nat} \quad ✗$$

Remark

Prenex polymorphism is a predicative and rank-1 fragment of System F.
Type reconstruction for ranks 2 and lower is **decidable**!

Homework

Do **one** of them!

Question (Exercise 23.5.1)

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

Question (Exercise 23.5.2)

If t is a closed, well-typed term, then either t is a value or else there is some t' with $\mathsf{t} \longrightarrow \mathsf{t}'$.