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

Variable Types
变量类型

Monomorphic Types

Observation

So far in the course, every well-typed closed term has a **unique** type.
However, we often want to implement the same behavior for different types.

- **Identity function:** $\lambda x{:}\mathtt{Nat}.\ x$, $\lambda x{:}\mathtt{Bool}.\ x$, $\lambda x{:}(\mathtt{Nat} \to \mathtt{Bool}).\ x$, ...
- **Double application:** $\lambda f{:}(\mathtt{Nat} \to \mathtt{Nat}).\ \lambda x{:}\mathtt{Nat}.\ f\ (f\ x)$,
  $\lambda f{:}((\mathtt{Nat} \to \mathtt{Bool}) \to (\mathtt{Nat} \to \mathtt{Bool})).\ \lambda x{:}(\mathtt{Nat} \to \mathtt{Bool}).\ f\ (f\ x)$, ...
- **Composition:** $\lambda f{:}(T_2 \to T_3).\ \lambda g{:}(T_1 \to T_2).\ \lambda x{:}T_1.\ f\ (g\ x)$ for every triple T_1, T_2, T_3 of types

Observation

Albeit with different types, the terms with the same behavior are **almost identical**.

Question

How can a programming language capture such a pattern once and for all?

Polymorphic Types

Principle (Abstraction)

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

Example

Replace

```
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);
```

with

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

Question

Can you think of different kinds of polymorphic types?

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"`
- **Typeclass**: `(+) :: 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: Most Powerful Parametric Polymorphism

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 $\lambda_{\rightarrow}$.

- In $\lambda_{\rightarrow}$, 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.

Universal Types: Syntax and Evaluation

Syntax

$$
\begin{aligned}
t &::= \ldots \mid \lambda X.\, t \mid t\ [T] \\
v &::= \ldots \mid \lambda X.\, t
\end{aligned}
$$

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

Let us define $id \stackrel{\text{def}}{=} \lambda X.\, \lambda x{:}X.\, x$.

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

Universal Types: 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

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

Universal Types: Type Formation

Observation

Not all syntactically well-formed types are semantically well-formed, e.g., $\forall X.\, Y \to X$.

Type Formation

$$\frac{}{\Gamma, X \vdash X \text{ type}} \qquad \frac{\Gamma \vdash T_1 \text{ type} \qquad \Gamma \vdash T_2 \text{ type}}{\Gamma \vdash T_1 \to T_2 \text{ type}} \qquad \frac{\Gamma, X \vdash T_1 \text{ type}}{\Gamma \vdash \forall X.\, T_1 \text{ type}}$$

$$\frac{\Gamma \vdash T_1 \text{ type} \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs} \qquad \frac{\Gamma \vdash t_1 : \forall X.\, T_{12} \qquad \Gamma \vdash T_2 \text{ type}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \text{ T-TApp}$$

Question (Regularity)

Prove that if $\varnothing \vdash t : T$, then $\varnothing \vdash T$ type.

Example: 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
```

Example: Polymorphic Lists

`List` as a Type Operator

We assume 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
```

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
```

Example: Polymorphic Lists

Question (Exercise 23.4.3)

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

A 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
```

Example: Polymorphic Lists

List as a Type Operator

We have assumed 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
```

Aside

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

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

Question

Implement polymorphic binary trees with System F + recursive types.

Expressiveness of System F

Question

Consider the "vanilla" System F whose types only have three forms: $\mathsf{T} ::= \mathsf{X} \mid \mathsf{T} \to \mathsf{T} \mid \forall \mathsf{X}.\, \mathsf{T}$.
How expressive can it be?
Can it express Booleans, natural numbers, lists, products, sums, inductive/coinductive types, etc.?
Can it express fixed points?

Remark (Church Encodings)

In Chapter 5, we saw that untyped lambda calculus can express all of the notions above.
Let us see if those encodings are well-typed terms in System F.

Church Encodings: Booleans

Remark (Church Booleans)

```
tru = λt. λf. t;
fls = λt. λf. f;
test = λb. λm. λn. b m n;
```

```
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
test = λY. λb:CBool. λm:Y. λn:Y. b [Y] m n;
▶ test : ∀Y. CBool → Y → Y → Y
```

Question

Why does the polymorphic function type `CBool` characterize Booleans?

Church Encodings: Booleans

Typing Rules for Booleans

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

Observation

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

Principle

Encode typing rules for **elimination forms** as polymorphic function types.

Example

Using Booleans are directly applying their polymorphic functions with respect to **the elimination typing rule**.

```
test = λT. λt1:CBool. λt2:T. λt3:T. t1 [T] t2 t3;
```

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.

A Solution: Dummy Abstractions

```
CBool = ∀X. (Unit→X)→(Unit→X)→X;
test = λY. λb:CBool. λm:(Unit→Y). λn:(Unit→Y). b [Y] m n;
▶ 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: Unit

Typing Rules for Unit

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}} \text{ T-Unit} \qquad \frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{let unit} = t_1 \texttt{ in } t_2 : T} \text{ T-LetUnit}$$

Question

Encode the elimination rule (T-LetUnit) as a polymorphic function type `CUnit`.

A Solution

```
CUnit = ∀X. X→X;
unit = (λX. λr:X. r) as CUnit;
▶ unit : CUnit
seq = λY. λu:CUnit. λm:Y. u [Y] m;
▶ seq : ∀Y. CUnit → Y → Y
```

It is worth noting that `unit` is the **polymorphic identity function**.

Church Encodings: Products

Typing Rules for Products

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \text{ T-Pair} \qquad \frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \text{ T-Proj1} \qquad \frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \text{ T-Proj2}$$

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

Question

How to encode the elimination rule (T-LetPair) as a polymorphic function type?

A Solution

$\mathtt{Pair}_{T_{11},T_{12}}$ = $\forall$S. ($T_{11} \to T_{12} \to$ S) $\to$ S;

We will later see how to extend the type system to support **type operators** like `Pair`.

Church Encodings: Products

```
Pair_{T1,T2} = ∀X. (T1→T2→X) → X;

pair_{T1,T2} = λx:T1. λy:T2. (λX. λp:(T1→T2→X). p x y) as Pair_{T1,T2};
▶ pair_{T1,T2} : T1 → T2 → Pair_{T1,T2}

unpair_{T1,T2} = λY. λp:Pair_{T1,T2}. λm:(T1→T2→Y). p [Y] m;
▶ unpair_{T1,T2} : ∀Y. Pair_{T1,T2} → (T1→T2→Y) → Y

fst_{T1,T2} = λp:Pair_{T1,T2}. p [T1] (λx:T1. λ_:T2. x);
▶ fst_{T1,T2} : Pair_{T1,T2} → T1
snd_{T1,T2} = λp:Pair_{T1,T2}. p [T2] (λ_:T1. λy:T2. y);
▶ snd_{T1,T2} : Pair_{T1,T2} → T2
```

Question

Use `unpair` to define `fst` and `snd`.

Church Encodings: Sums

Question

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

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

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x_1 : T_1 \vdash t_1 : S \qquad \Gamma, x_2 : T_2 \vdash t_2 : S}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 : S}\ \text{T-Case}$$

A Solution

$\texttt{Sum}_{T_1,T_2}$ = $\forall$S. $(T_1 \to S) \to (T_2 \to S) \to S$;

$\texttt{inl}_{T_1,T_2}$ = λv:T_1. (λS. λl:($T_1 \to$S). λr:($T_2 \to$S). l v) **as** $\texttt{Sum}_{T_1,T_2}$;
▶ $\texttt{inl}_{T_1,T_2}$: $T_1 \to \texttt{Sum}_{T_1,T_2}$
$\texttt{inr}_{T_1,T_2}$ = λv:T_2. (λS. λl:($T_1 \to$S). λr:($T_2 \to$S). r v) **as** $\texttt{Sum}_{T_1,T_2}$;
▶ $\texttt{inr}_{T_1,T_2}$: $T_2 \to \texttt{Sum}_{T_1,T_2}$

Church Encodings: Sums

```
Sum_{T1,T2} = ∀X. (T1→X) → (T2→X) → X;

inl_{T1,T2} = λv:T1. (λX. λl:(T1→S). λr:(T2→S). l v) as Sum_{T1,T2};
▶ inl_{T1,T2} : T1 → Sum_{T1,T2}
inr_{T1,T2} = λv:T2. (λX. λl:(T1→S). λr:(T2→S). r v) as Sum_{T1,T2};
▶ inl_{T1,T2} : T2 → Sum_{T1,T2}

test = λY. λb:Sum_{T1,T2}. λm:(T1→Y). λn:(T2→Y). b [Y] m n;
▶ test : ∀Y. Sum_{T1,T2} → (T1→Y) → (T2→Y) → Y
```

Question

How to encode $\texttt{case}\ t_0\ \texttt{of inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2$?

A Solution

$\texttt{test}\ [T]\ t_0\ (\lambda x_1{:}T_1.\, t_1)\ (\lambda x_2{:}T_2.\, t_2)$, where T is the type of t_1 and t_2.

Church Encodings: Natural Numbers

Remark (Church Numerals)

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

Question

To repeat the practice, we need a typing rule for **eliminating** natural numbers.
Hint: we shall view the type of natural numbers as an **inductive type**.

A Solution

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat} \qquad \Gamma, x : \mathtt{Unit} + S \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [\mathtt{Nat}]\ t_1\ \mathbf{with}\ x.\, t_2 : S} \text{ T-Iter-Nat}$$

Thus, we can extract a possible encoding $\forall S.\ ((\mathtt{Unit} + S) \to S) \to S$.

Church Encodings: Natural Numbers

```
CNat = ∀X. ((Unit + X) → X) → X;
CNat = ∀X. ((Unit → X) × (X → X)) → X;
CNat = ∀X. (X × (X → X)) → X;
CNat = ∀X. (X → X) → X → X;
```

Remark

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma \vdash t_2 : S \qquad \Gamma, x : S \vdash t_3 : S}{\Gamma \vdash \mathbf{iter}\ [\mathsf{Nat}]\ t_1\ \mathbf{with}\ \mathsf{zero} \Rightarrow t_2 \mid \mathsf{succ} \Rightarrow x.\, t_3 : S} \text{ T-Iter-Nat}$$

```
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
c2 = (λX. λs:X→X. λz:X. s (s z)) as CNat;
▶ c2 : CNat
...
```

Church Encodings: Natural Numbers

```
CNat = ∀X. (X → X) → X → X;

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

plus = λm:CNat. λn:CNat. m [CNat] succ n;
▶ plus : CNat → CNat → CNat
```

Question

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

Observation

We do **not** need recursion to define `plus` and `mult`. **How can it be possible?**

Church Encodings: Lists

Question

We have seen $\mathtt{List}\ \mathtt{T}$ as a primitive type or as a recursive type. Can we encode it in the “vanilla” System F?

Remark (Iterating over Lists)

$$\frac{\Gamma \vdash t_1 : \mathtt{List}\ T_{11} \qquad \Gamma, x : \mathtt{Unit} + T_{11} \times S \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [\mathtt{List}\ T_{11}]\ t_1\ \mathbf{with}\ x.\,t_2 : S} \text{ T-Iter-List}$$

$$\mathtt{List}_{T_{11}} = \forall S.\ ((\mathtt{Unit} + T_{11} \times S) \to S) \to S;$$
$$\mathtt{List}_{T_{11}} = \forall S.\ ((\mathtt{Unit} \to S) \times (T_{11} \times S \to S)) \to S;$$
$$\mathtt{List}_{T_{11}} = \forall S.\ (S \times (T_{11} \to S \to S)) \to S;$$
$$\mathtt{List}_{T_{11}} = \forall S.\ (T_{11} \to S \to S) \to S \to S;$$

Church Encodings: Lists

```
ListT = ∀X. (T→X→X) → X → X;

nilT = (λX. λc:(T→X→X). λn:X. n) as ListT;
▶ nilT : ListT
consT = λhd:T. λtl:ListT. (λX. λc:(T→X→X). λn:X. c hd (tl [X] c n)) as ListT;
▶ consT : T → ListT → ListT

isnilT = λl:ListT. l [Bool] (λ_:T. λ_:Bool. false) true;
▶ isnilT : ListT → Bool
headT = λl:ListT. l [T] (λhd:T. λ_:T. hd) error;
▶ headT : ListT → T
```

Question

- The definition above for $\texttt{head}_\texttt{T}$ does not work under call-by-value. Can you make it work?
- Can you define a function $\texttt{sum} : \texttt{List}_\texttt{Nat} \to \texttt{Nat}$ **without** using recursion?

Church Encodings: Inductive Types

Remark (Iteration)

$$\frac{\Gamma \vdash t_1 : \mathtt{ind}(X.\,T) \qquad \Gamma, x : [X \mapsto S]T \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [X.\,T]\ t_1\ \mathbf{with}\ x.\,t_2 : S}\ \text{T-Iter}$$

Principle

For every inductive type $\mathtt{ind}(X.\,T)$, its encoding in System F could be the following:

$$\mathtt{Ind}_{X.\,T} = \forall S.\ ([X \mapsto S]T \to S) \to S;$$

$$\mathtt{fold}_{X.\,T} = \lambda v{:}[X \mapsto \mathtt{Ind}_{X.\,T}]T.\ (\lambda S.\ \lambda f{:}([X \mapsto S]T \to S).\ \mathbf{map}\ [X.\,T]\ v\ \mathbf{with}\ x.\ f\ x)\ \mathbf{as}\ \mathtt{Ind}_{X.\,T};$$

▶ $\mathtt{fold}_{X.\,T} : [X \mapsto \mathtt{Ind}_{X.\,T}]T \to \mathtt{Ind}_{X.\,T}$

Question

Can we encode **coinductive types** in a similar way?

Church Encodings: Streams

Remark (Generation of Streams)

Previously, we define $\texttt{Stream}$ as a coinductive type $\texttt{coi}(X.\,\texttt{Nat} \times X)$.

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : \texttt{Nat} \times S}{\Gamma \vdash \textbf{gen}\ [X.\,\texttt{Nat} \times X]\ t_1\ \textbf{with}\ x.\,t_2 : \texttt{coi}(X.\,\texttt{Nat} \times X)}\ \text{T-Gen-Stream}$$

Observation

The parameter type S does **NOT** appear in the conclusion part!
We need a notion to say that there **exists** some type S, such that a stream consists of an "internal state" of type S and a "generator" of type $S \to \texttt{Nat} \times S$.

Observation

From the perspective of **elimination**, one can use S and $S \to \texttt{Nat} \times S$ to produce a value of some other type T.

Church Encodings: Streams

An Encoding of Streams

```
Stream = ∀T. (∀S. S → (S → Nat × S) → T) → T;

unfold_Stream = λv:Stream. v [Nat × Stream]
                  (λS. λs:S. λg:(S→Nat×S).
                    let v' = g s in
                    {v'.1,(λT. λf:(∀S. S→(S→Nat×S)→T). f [S] v'.2 g)})
```

- $\text{unfold}_{\texttt{Stream}} : \texttt{Stream} \to \texttt{Nat} \times \texttt{Stream}$

Question

Encode the generation rule (T-Gen-Stream) as $\texttt{gen}_{\texttt{Stream}} : \forall S.\, S \to (S \to \texttt{Nat} \times S) \to \texttt{Stream}$.

Church Encodings: Coinductive Types

Remark (Generation)

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : [X \mapsto S]T}{\Gamma \vdash \textbf{gen}\ [X.\,T]\ t_1\ \textbf{with}\ x.\,t_2 : \mathtt{coi}(X.\,T)}\ \text{T-Gen}$$

Principle

For every coinductive type $\mathtt{coi}(X.\,T)$, its encoding in System F could be the following:

$$\mathtt{Coi}_{X.T} = \forall Y.\ (\forall S.\ S \to (S \to [X \mapsto S]T) \to Y) \to Y;$$

$$\begin{aligned}\mathtt{unfold}_{X.T} = \lambda v{:}\mathtt{Coi}_{X.T}.\ &v\ [[X \mapsto \mathtt{coi}(X.\,T)]T]\\ &(\lambda S.\ \lambda s{:}S.\ \lambda g{:}(S \to [X \mapsto S]T).\\ &\quad \textbf{let}\ v' = g\ s\ \textbf{in}\\ &\quad \textbf{map}\ [X.\,T]\ v'\ \textbf{with}\ x.\ (\lambda Y.\ \lambda f{:}(\forall S.\ S\to(S\to[X \mapsto S]T)\to Y).\ f\ [S]\ x\ g));\end{aligned}$$

- $\mathtt{unfold}_{X.T} : \mathtt{Coi}_{X.T} \to [X \mapsto \mathtt{Coi}_{X.T}]T$

Properties of System F

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` (i.e., $\lambda X.\, \lambda t{:}X.\, \lambda f{:}X.\, t$) or `fls` (i.e., $\lambda X.\, \lambda t{:}X.\, \lambda f{:}X.\, f$).

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 Unary Case

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 Unary Case

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 $\mathsf{m} : \mathsf{T}$ and of $\mathsf{n} : \mathsf{T}$, then $\mathcal{P}$ holds of $b\ [\mathsf{T}]\ \mathsf{m}\ \mathsf{n}$.

Example

Fix $\mathsf{t}_0 : \mathsf{T}$ and $\mathsf{t}_1 : \mathsf{T}$. Consider $\mathcal{P}_{\mathsf{t}_0,\mathsf{t}_1}$ that holds of $\mathsf{t}_2 : \mathsf{T}$ iff t_2 is equivalent to either t_0 or t_1.
Obviously $\mathcal{P}_{\mathsf{t}_0,\mathsf{t}_1}$ holds of both t_0 and t_1.
By the proposition above, $\mathcal{P}_{\mathsf{t}_0,\mathsf{t}_1}$ holds of $b\ [\mathsf{T}]\ \mathsf{t}_0\ \mathsf{t}_1$.
Thus, $b\ [\mathsf{T}]\ \mathsf{t}_0\ \mathsf{t}_1$ is equivalent to either t_0 or t_1.

Parametricity: The Unary Case

Definition

- The judgment $\mathcal{P} : T$ states that $\mathcal{P}$ is a **admissible property** for type T, i.e., $\mathcal{P}$ is a set of closed terms of type T closed under head expansion.
- The judgment $\delta : \Gamma$ states that δ is a **type substitution** that assigns a closed type $\delta(X)$ to each type variable $X \in \Gamma$. A type substitution δ induces a substitution $\hat{\delta}$ on types $\hat{\delta}(T) \stackrel{\text{def}}{=} [X_1 \mapsto \delta(X_1), \ldots, X_n \mapsto \delta(X_n)]T$.
- The judgment $\eta : \delta$ states that η is an **admissible property assignment** on $\delta : \Gamma$ that assigns an admissible property $\eta(X) : \delta(X)$ to each $X \in \Gamma$.

Definition ($t \in T\ [\eta : \delta]$)

$$
\begin{array}{rcl}
t \in X\ [\eta : \delta] & \text{iff} & \eta(X)(t) \\
t \in \mathtt{Bool}\ [\eta : \delta] & \text{iff} & t \longrightarrow^* \mathtt{true} \text{ or } t \longrightarrow^* \mathtt{false} \\
t \in T_1 \to T_2\ [\eta : \delta] & \text{iff} & t_1 \in T_1\ [\eta : \delta] \text{ implies } t\ t_1 \in T_2\ [\eta : \delta] \\
t \in \forall X.\, T\ [\eta : \delta] & \text{iff} & \text{for every type } S \text{ and admissible property } \mathcal{P} : S,\ t\ [S] \in T\ [(\eta, X : S) : (\delta, X : \mathcal{P})]
\end{array}
$$

Parametricity: The Unary Case

Definition

- The judgment $\gamma : \Gamma$ states that γ is a **term substitution** that assigns a closed term $\gamma(x) : \Gamma(x)$ to each variable $x \in \Gamma$. A term substitution γ induces a substitution $\hat{\gamma}$ on terms $\hat{\gamma}(t) \stackrel{\text{def}}{=} [x_1 \mapsto \gamma(x_1), \ldots, x_n \mapsto \gamma(x_n)]t$.
- The judgment $\gamma \in \Gamma\ [\eta : \delta]$ states that γ and Γ covers the same set of variables and for each such variable x it holds that $\gamma(x) \in \Gamma(x)\ [\eta : \delta]$.
- The judgment $\Gamma \vdash t \in T$ states that for every type substitution $\delta : \Gamma$, every admissible property assignment $\eta : \delta$, and every term substitution $\gamma : \Gamma$, if $\gamma \in \Gamma\ [\eta : \delta]$, then $\hat{\gamma}(\hat{\delta}(t)) \in T\ [\eta : \delta]$.

Theorem (Parametricity)

If $\Gamma \vdash t : T$, then $\Gamma \vdash t \in T$.

Proof Sketch

By induction on the derivation of $\Gamma \vdash t : T$.

Parametricity: Beyond The Unary Case

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 $t : \mathrm{T}$, then $\mathcal{P}$ holds of $id\ [\mathrm{T}]\ t : \mathrm{T}$.

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** $t : \mathrm{T}$ to $t' : \mathrm{T}'$, then $\mathcal{R}$ **relates** $id\ [\mathrm{T}]\ t : \mathrm{T}$ to $id\ [\mathrm{T}']\ t' : \mathrm{T}'$.

Proposition (A Free Theorem)

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

Impredicativity

Remark (Russell's Paradox)

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

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

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**.

Two Views of Universal Type $\forall X.\, T$

Logical Intuition

- An element of $\forall X.\, T$ is a value of type $[X \mapsto S]T$ **for all** choices of S.
- The identify function $\lambda X.\, \lambda x{:}X.\, x$ erases to $\lambda x.\, x$, mapping a value of any type S to a value of the same type.

Operational Intuition

- An element of $\forall X.\, T$ is a **function** mapping **any** type S to a specialized term with type $[X \mapsto S]T$.
- In the (E-TappTabs) rule, the reduction of a type application is an actual computation step.

Question

We have already seen universal quantifiers $\forall$. What about existential quantifiers $\exists$?

Two Views of Existential Type $\exists X.\ T$

Logical Intuition

An element of $\exists X.\ T$ is a value of type $[X \mapsto S]T$ **for some** type S.

Operational Intuition

An element of $\exists X.\ T$ is a **pair** of **some** type S and a term of type $[X \mapsto S]T$.

Remark

We will focus on the operational view of existential types.
The essence of existential types is that they **hide information** about the packaged type.

Notations

We write $\{\exists X, T\}$ (instead of $\exists X.\ T$) to emphasize the operational view.
The pair of type $\{\exists X, T\}$ is written $\{{}^*S, t\}$ of a type S and a term t of type $[X \mapsto S]T$.

A Simple Example

Example

The pair

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}}
```

has the existential type $\{\exists X, \{\mathtt{a} : X, \mathtt{f} : X \to X\}\}$.

- The type component of p is Nat.
- The value component is a record containing of field a of type X and a field f of type $X \to X$, **for some X**.

Example

The same pair p also has the type $\{\exists X, \{\mathtt{a} : X, \mathtt{f} : X \to \mathtt{Nat}\}\}$.
In general, the typechecker cannot decide **how much information should be hidden**.

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▶ p : {∃X, {a:X, f:X→X}}
p1 = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p1 : {∃X, {a:X, f:X→Nat}}
```

Introduction Rule for $\{\exists X, T\}$

Typing

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

Example

Pairs with **different** hidden representation types can inhabit the **same** existential type.

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
 ▶ p4 : {∃X, {a:X, f:X→Nat}}
p5 = {*Bool, {a=ture, f=λx:Bool. if x then 1 else 0}} as {∃X, {a:X, f:X→Nat}};
 ▶ p5 : {∃X, {a:X, f:X→Nat}}
```

Elimination Rule for $\{\exists X, T\}$

Typing

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

Example

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p4 : {∃X, {a:X, f:X→Nat}}

let {X,x}=p4 in (x.f x.a);
▶ 1 : Nat
let {X,x}=p4 in (λy:X. x.f y) x.a;
▶ 1 : Nat
```

Subtlety of Elimination Rule

Example

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p4 : {∃X, {a:X, f:X→Nat}}

let {X,x}=p4 in succ(x.a);
▶ Error: argument of succ is not a number

let {X,x}=p4 in x.a;
▶ Error: scoping error!
```

Aside

A simple solution for the scoping problem is to add a well-formedness check as a premise:

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

Existential Types: Syntax and Evaluation

Syntax

$$
\begin{aligned}
t &::= \ldots \mid \{{}^*T, t\} \text{ as } T \mid \texttt{let}\ \{X, x\} = t \texttt{ in } t \\
v &::= \ldots \mid \{{}^*T, v\} \text{ as } T \\
T &::= \ldots \mid \{\exists X, T\}
\end{aligned}
$$

Evaluation

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

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

$$
\frac{t_1 \longrightarrow t'_1}{\texttt{let}\ \{X, x\} = t_1 \texttt{ in } t_2 \longrightarrow \texttt{let}\ \{X, x\} = t'_1 \texttt{ in } t_2} \text{ E-Unpack}
$$

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.

```
ADT counter =
  type Counter
  representation Nat
  signature
     new : Counter,
     get : Counter→Nat,
     inc : Counter→Counter;
```

```
operations
   new = 1,
   get = λi:Nat. i,
   inc = λi:Nat. succ(i);
```

Translating ADTs to Existentials

```
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}}

let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);
▶ 2 : Nat
```

ADTs and Modules / Packages

Observation

An element of an existential type can be seen as a **module** or a **package**, in the following sense:

```
let {Counter,counter} = <counter module / counter package> in
<rest of program that uses the module / package>
```

```
let {Counter,counter} = counterADT in
let {FlipFlop,flipflop} =
     {*Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop, read: FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop, reset: FlipFlop→FlipFlop}} in
flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
▶ false : Bool
```

Representation Independence

Observation

We can substitute an alternative implementation of the `Counter` ADT and the program will remain typesafe.

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

let {Counter,counter} = counterADT in
let {FlipFlop,flipflop} = ...
```

Existential Objects

Idea

We choose a **purely functional** style, i.e., when we need to change the object's internal state, we instead build a fresh object.

A counter object consists of (i) a number (**its internal state**) and (ii) a pair of methods (**its external interface**):

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};

c = {*Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
    as Counter;
▶ c : Counter
```

Existential Objects

```
let {X,body} = c in body.methods.get(body.state);
▶ 5 : Nat

sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▶ sendget : Counter → Nat

let {X,body} = c in body.methods.inc(body.state);
▶ Error: scoping error!

sendinc = λc:Counter.
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter
```

ADTs vs. Objects

ADTs

`CounterADT = {∃Counter, {new:Counter,get:Counter→Nat,inc:Counter→Counter}}`

"The abstract type of counters" refers to the (hidden) type `Nat`, i.e., simple numbers.

ADTs are usually used in a **pack-and-then-open** manner, leading to a **unique** internal representation type.

Objects

`Counter = {∃X, {state:X, methods:{get:X→Nat,inc:X→X}}}`

"The abstract type of counters" refers to the whole package, including the number and the implementations.

Objects are kept closed as long as possible and each object carries its **own** representation type.

Observation

The object style is convenient in the presence of **subtyping** and **inheritance**.

ADTs vs. Objects

Question

What about implementing **binary** operations on the same abstract type?

Let us consider a simple case: we want to implement an equality operation for counters.

ADT Style

```
let {Counter,counter} = counterADT in
let counter_eq = λc1:Counter. λc2.Counter. nat_eq (counter.get c1) (counter.get c2)
in <rest of program>
```

Object Style

```
let counter_eq = λc1:Counter. λc2:Counter.
  let {X1,body1} = c1 in
  let {X2,body2} = c2 in
  nat_eq body1.methods.get(body1.state) body2.methods.get(body2.state);
```

ADTs vs. Objects

Remark

The equality operation can be implemented outside the abstraction boundary.

Let us consider implementing an abstraction for sets of numbers.
The concrete representation is labeled trees and is **NOT** exposed to the outside.
We'd implement a `union` operation that needs to view the **concrete representation of both** arguments.

ADT Style

$$\texttt{NatSetADT} = \{\exists\,\texttt{NatSet},\ \{\ldots,\ \texttt{union}:\texttt{NatSet}\to\texttt{NatSet}\to\texttt{NatSet}\}\}$$

Object Style

$$\texttt{NatSet} = \{\exists\,\texttt{X},\ \{\texttt{state}:\texttt{X},\ \texttt{methods}:\{\ldots,\ \texttt{union}:\texttt{X}\to\texttt{NatSet}\to\texttt{X}\}\}\}$$

Problems: (i) we need recursive types, and (ii) `union` **cannot access the concrete structure of its 2nd argument**.

ADTs vs. Objects

Question (Exercise 24.2.5)

Why can't we use the type

```
NatSet = {∃X, {state:X, methods:{..., union:X→X→X}}}
```

instead?

Answer

We cannot send a `union` message to a `NatSet` object, with another `NatSet` object as an argument of the message:

```
sendunion = λs1:NatSet. λs2:NatSet.
              let {X1,body1} = s1 in
              let {X2,body2} = s2 in
              ... body1.methods.union body1.state body2.state ...
```

Another explanation: objects allow different internal representations, thus `union:X→X → X` is not safe.

Question

In C++, Java, etc., it's not difficult to implement such a `union` operation. How does that work?

Encoding Existentials in System F

The Elimination Rule for Existentials

$$\frac{\Gamma \vdash t_1 : \{\exists X, T\} \qquad \Gamma, X, x : T \vdash t_2 : S}{\Gamma \vdash \mathtt{let}\ \{X, x\} = t_1\ \mathtt{in}\ t_2 : S}\ \text{T-Unpack}$$

$$\{\exists\, X, T\} \stackrel{\text{def}}{=} \forall S.\ (\forall X.\ T \to S) \to S$$

$$\{{*}S, t\}\ \textbf{as}\ \{\exists\, X, T\} \stackrel{\text{def}}{=} \lambda S.\ \lambda f{:}(\forall X.\ T \to YS.\ f\ [S]\ t$$

$$\textbf{let}\ \{X, x\} = t1\ \textbf{in}\ t2 \stackrel{\text{def}}{=} t1\ [S]\ (\lambda X.\ \lambda x{:}T.\ t2)$$

Homework

Do **one** of them!

Question (Exercise 23.5.1)

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

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 $t \longrightarrow t'$.