



# 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 2026



# 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:**  $\lambda x:\text{Nat}. x$ ,  $\lambda x:\text{Bool}. x$ ,  $\lambda x:(\text{Nat} \rightarrow \text{Bool}). x$ , ...
- **Double application:**  $\lambda f:(\text{Nat} \rightarrow \text{Nat}). \lambda x:\text{Nat}. f (f x)$ ,  
 $\lambda f:(\text{Nat} \rightarrow \text{Bool}) \rightarrow (\text{Nat} \rightarrow \text{Bool}). \lambda x:(\text{Nat} \rightarrow \text{Bool}). f (f x)$ , ...
- **Composition:**  $\lambda f:(T_2 \rightarrow T_3). \lambda g:(T_1 \rightarrow T_2). \lambda x:T_1. f (g x)$  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 = λ f:Nat→Nat. λ a:Nat. f (f a);  
doubleRcd = λ f:{l:Bool}→{l:Bool}. λ a:{l:Bool}. f (f a);  
doubleFun = λ f:(Nat→Nat)→(Nat→Nat). λ a:Nat→Nat. f (f a);
```

with

```
double = λ X. λ f:X→X. λ a: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 = λ X. λ x: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**: 
$$\frac{\Gamma \vdash t : S \quad S \leq T}{\Gamma \vdash t : T}$$

# System F: Most Powerful Parametric Polymorphism



## Some Historical Accounts

- System F was introduced by Girard (1972) in the context of proof theory.<sup>1</sup>
- System F was independently developed by Reynolds (1974) in the context of programming languages.<sup>2</sup>
- 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.

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

<sup>2</sup>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

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

## Evaluation

$$\frac{t_1 \longrightarrow t'_1}{t_1 [T_2] \longrightarrow t'_1 [T_2]} \text{E-TApp}$$

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

## Example

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

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

# Universal Types: Types, Type Contexts, and Typing



## Types and Type Contexts

$$T ::= X \mid T \rightarrow T \mid \forall X. T$$
$$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, X$$

## Typing

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

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

## Example

$$\frac{\frac{\frac{}{X, x : X \vdash x : X} \text{ T-Var}}{X \vdash \lambda x : X. x : X \rightarrow X} \text{ T-Abs}}{\emptyset \vdash \lambda X. \lambda x : X. x : \forall X. X \rightarrow X} \text{ T-TAbs}}$$



# Universal Types: Type Formation

## Observation

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

## Type Formation

$$\frac{}{\Gamma, X \vdash X \text{ type}}$$

$$\frac{\Gamma \vdash T_1 \text{ type} \quad \Gamma \vdash T_2 \text{ type}}{\Gamma \vdash T_1 \rightarrow T_2 \text{ type}}$$

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

$$\frac{\Gamma \vdash T_1 \text{ type} \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : \forall X. T_{12} \quad \Gamma \vdash T_2 \text{ type}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2] T_{12}} \text{T-TApp}$$

## Question (Regularity)

Prove that if  $\emptyset \vdash t : T$ , then  $\emptyset \vdash T \text{ type}$ .



# 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 :  $\forall X. \text{List } X$   
cons :  $\forall X. X \rightarrow \text{List } X \rightarrow \text{List } X$ 
```

```
isnil :  $\forall X. \text{List } X \rightarrow \text{Bool}$   
head :  $\forall X. \text{List } X \rightarrow X$   
tail :  $\forall X. \text{List } X \rightarrow \text{List } X$ 
```

## Example

```
map =  $\lambda X. \lambda Y. \lambda f: X \rightarrow Y.$   
      (fix ( $\lambda m: (\text{List } X) \rightarrow (\text{List } Y).$   
            $\lambda l: \text{List } X.$   
             if isnil [X] l then nil [Y]  
               else cons [Y] (f (head [X] l)) (m (tail [X] l))));  
► map :  $\forall X. \forall Y. (X \rightarrow Y) \rightarrow \text{List } X \rightarrow \text{List } Y$ 
```



# Example: Polymorphic Lists

## Question (Exercise 23.4.3)

Using `map` as a model, write a polymorphic list-reversing function: `reverse :  $\forall X. \text{List } X \rightarrow \text{List } X$` .

## A Solution

```
rev_append =  $\lambda X. \mathbf{fix} (\lambda ra:(\text{List } X) \rightarrow (\text{List } X) \rightarrow (\text{List } X). \lambda l1:(\text{List } X). \lambda l2:(\text{List } X). \mathbf{if} \text{isnil } [X] \ l1 \ \mathbf{then} \ l2 \ \mathbf{else} \ ra \ (\text{tail } [X] \ l1) \ (\text{cons } [X] \ (\text{head } [X] \ l1) \ l2));$ 
```

► `rev_append :  $\forall X. \text{List } X \rightarrow \text{List } X \rightarrow \text{List } X$`

```
reverse =  $\lambda X. \lambda l:\text{List } X. \text{rev\_append } [X] \ l \ (\text{nil } [X]);$ 
```

► `reverse :  $\forall X. \text{List } X \rightarrow \text{List } X$`



# Example: Polymorphic Lists

## List as a Type Operator

We have assumed the language has the following primitives:

`nil` :  $\forall X. \text{List } X$

`cons` :  $\forall X. X \rightarrow \text{List } X \rightarrow \text{List } X$

`isnil` :  $\forall X. \text{List } X \rightarrow \text{Bool}$

`head` :  $\forall X. \text{List } X \rightarrow X$

`tail` :  $\forall X. \text{List } X \rightarrow \text{List } X$

## Aside

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

`nil` =  $\lambda X. \langle \text{nil}=\text{Unit} \rangle$  **as**  $(\mu T. \langle \text{nil}:\text{Unit}, \text{cons}:\{X, T\} \rangle);$

► `nil` :  $\forall X. \mu T. \langle \text{nil}:\text{Unit}, \text{cons}:\{X, T\} \rangle$

## 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:  $T ::= X \mid T \rightarrow T \mid \forall X. T$ .

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 = λ t. λ f. t;  
fls = λ t. λ f. f;  
test = λ b. λ m. λ n. b m n;
```

$CBool = \forall X. X \rightarrow X \rightarrow X$ ;

$tru = (\lambda X. \lambda t:X. \lambda f:X. t)$  **as**  $CBool$ ;

▶  $tru : CBool$

$fls = (\lambda X. \lambda t:X. \lambda f:X. f)$  **as**  $CBool$ ;

▶  $fls : CBool$

$test = \lambda Y. \lambda b:CBool. \lambda m:Y. \lambda n:Y. b [Y] m n$ ;

▶  $test : \forall Y. CBool \rightarrow Y \rightarrow Y \rightarrow Y$

## Question

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

# Church Encodings: Booleans

## Typing Rules for Booleans

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \text{T-True}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \text{T-False}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{T-If}$$

## Observation

The definition  $\text{CBool} = \forall T. T \rightarrow T \rightarrow T$  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 = λ T. λ t1:CBool. λ t2:T. λ t3:T. t1 [T] t2 t3;
```



# Church Encodings: Booleans

## Question

Can `test` be used as conditional expressions?

## Observation

Under call-by-value, `test [T] t1 t2 t3` (where `T` is the type of `t2, t3`) evaluates **both** `t2` and `t3`.

## A Solution: Dummy Abstractions

`CBool` =  $\forall X. (\text{Unit} \rightarrow X) \rightarrow (\text{Unit} \rightarrow X) \rightarrow X$ ;

`test` =  $\lambda Y. \lambda b:\text{CBool}. \lambda m:(\text{Unit} \rightarrow Y). \lambda n:(\text{Unit} \rightarrow Y). b [Y] m n$ ;

► `test`:  $\forall Y. \text{CBool} \rightarrow (\text{Unit} \rightarrow Y) \rightarrow (\text{Unit} \rightarrow Y) \rightarrow Y$

We can encode `if t1 then t2 else t3` as `test [T] t1 ( $\lambda\_:\text{Unit}. t_2$ ) ( $\lambda\_:\text{Unit}. t_3$ )`.

## Question

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



# Church Encodings: Unit

## Typing Rules for Unit

$$\frac{}{\Gamma \vdash \text{unit} : \text{Unit}} \text{T-Unit}$$

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

## Question

Encode the elimination rule (T-LetUnit) as a polymorphic function type  $\mathcal{C}\text{Unit}$ .

## A Solution

$\mathcal{C}\text{Unit} = \forall X. X \rightarrow X$ ;

$\text{unit} = (\lambda X. \lambda r : X. r)$  **as**  $\mathcal{C}\text{Unit}$ ;

►  $\text{unit} : \mathcal{C}\text{Unit}$

$\text{seq} = \lambda Y. \lambda u : \mathcal{C}\text{Unit}. \lambda m : Y. u [Y] m$ ;

►  $\text{seq} : \forall Y. \mathcal{C}\text{Unit} \rightarrow Y \rightarrow Y$

It is worth noting that  $\text{unit}$  is the **polymorphic identity function**.



# Church Encodings: Products

## Typing Rules for Products

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \text{ T-Pair}$$

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

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

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

## Question

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

## A Solution

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

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



# Church Encodings: Products

$\text{Pair}_{T_1, T_2} = \forall X. (T_1 \rightarrow T_2 \rightarrow X) \rightarrow X;$

$\text{pair}_{T_1, T_2} = \lambda x:T_1. \lambda y:T_2. (\lambda X. \lambda p:(T_1 \rightarrow T_2 \rightarrow X). p \ x \ y) \ \mathbf{as} \ \text{Pair}_{T_1, T_2};$

►  $\text{pair}_{T_1, T_2} : T_1 \rightarrow T_2 \rightarrow \text{Pair}_{T_1, T_2}$

$\text{unpair}_{T_1, T_2} = \lambda Y. \lambda p:\text{Pair}_{T_1, T_2}. \lambda m:(T_1 \rightarrow T_2 \rightarrow Y). p \ [Y] \ m;$

►  $\text{unpair}_{T_1, T_2} : \forall Y. \text{Pair}_{T_1, T_2} \rightarrow (T_1 \rightarrow T_2 \rightarrow Y) \rightarrow Y$

$\text{fst}_{T_1, T_2} = \lambda p:\text{Pair}_{T_1, T_2}. p \ [T_1] \ (\lambda x:T_1. \lambda _:T_2. x);$

►  $\text{fst}_{T_1, T_2} : \text{Pair}_{T_1, T_2} \rightarrow T_1$

$\text{snd}_{T_1, T_2} = \lambda p:\text{Pair}_{T_1, T_2}. p \ [T_2] \ (\lambda _:T_1. \lambda y:T_2. y);$

►  $\text{snd}_{T_1, T_2} : \text{Pair}_{T_1, T_2} \rightarrow T_2$

## Question

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



# Church Encodings: Sums

## Question

Recall that with sum types, we can define the Boolean type as `Unit + Unit` and Boolean 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 **eliminating** sum types.

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

## A Solution

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

$\text{inl}_{T_1, T_2} = \lambda v : T_1. (\lambda S. \lambda l : (T_1 \rightarrow S). \lambda r : (T_2 \rightarrow S). l \ v) \text{ as } \text{Sum}_{T_1, T_2};$

►  $\text{inl}_{T_1, T_2} : T_1 \rightarrow \text{Sum}_{T_1, T_2}$

$\text{inr}_{T_1, T_2} = \lambda v : T_2. (\lambda S. \lambda l : (T_1 \rightarrow S). \lambda r : (T_2 \rightarrow S). r \ v) \text{ as } \text{Sum}_{T_1, T_2};$

►  $\text{inr}_{T_1, T_2} : T_2 \rightarrow \text{Sum}_{T_1, T_2}$



# Church Encodings: Sums

$\text{Sum}_{T_1, T_2} = \forall X. (T_1 \rightarrow X) \rightarrow (T_2 \rightarrow X) \rightarrow X;$

$\text{inl}_{T_1, T_2} = \lambda v:T_1. (\lambda X. \lambda l:(T_1 \rightarrow S). \lambda r:(T_2 \rightarrow S). l\ v) \text{ as } \text{Sum}_{T_1, T_2};$

►  $\text{inl}_{T_1, T_2} : T_1 \rightarrow \text{Sum}_{T_1, T_2}$

$\text{inr}_{T_1, T_2} = \lambda v:T_2. (\lambda X. \lambda l:(T_1 \rightarrow S). \lambda r:(T_2 \rightarrow S). r\ v) \text{ as } \text{Sum}_{T_1, T_2};$

►  $\text{inr}_{T_1, T_2} : T_2 \rightarrow \text{Sum}_{T_1, T_2}$

$\text{test} = \lambda Y. \lambda b:\text{Sum}_{T_1, T_2}. \lambda m:(T_1 \rightarrow Y). \lambda n:(T_2 \rightarrow Y). b\ [Y]\ m\ n;$

►  $\text{test} : \forall Y. \text{Sum}_{T_1, T_2} \rightarrow (T_1 \rightarrow Y) \rightarrow (T_2 \rightarrow Y) \rightarrow Y$

## Question

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

## A Solution

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



# Church Encodings: Natural Numbers

## Remark (Church Numerals)

$$\begin{aligned}
c_0 &= \lambda s. \lambda z. z; \\
c_1 &= \lambda s. \lambda z. s z; \\
c_2 &= \lambda s. \lambda z. s (s z); \\
&\dots
\end{aligned}$$

## 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

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

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



# Church Encodings: Natural Numbers

$\mathbb{C}\text{Nat} = \forall X. ((\text{Unit} + X) \rightarrow X) \rightarrow X;$

$\mathbb{C}\text{Nat} = \forall X. ((\text{Unit} \rightarrow X) \times (X \rightarrow X)) \rightarrow X;$

$\mathbb{C}\text{Nat} = \forall X. (X \times (X \rightarrow X)) \rightarrow X;$

$\mathbb{C}\text{Nat} = \forall X. (X \rightarrow X) \rightarrow X \rightarrow X;$

## Remark

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

$c_0 = (\lambda X. \lambda s : X \rightarrow X. \lambda z : X. z) \mathbf{as} \mathbb{C}\text{Nat};$

▶  $c_0 : \mathbb{C}\text{Nat}$

$c_1 = (\lambda X. \lambda s : X \rightarrow X. \lambda z : X. s z) \mathbf{as} \mathbb{C}\text{Nat};$

▶  $c_1 : \mathbb{C}\text{Nat}$

$c_2 = (\lambda X. \lambda s : X \rightarrow X. \lambda z : X. s (s z)) \mathbf{as} \mathbb{C}\text{Nat};$

▶  $c_2 : \mathbb{C}\text{Nat}$

...



# Church Encodings: Natural Numbers

$\mathbb{C}\text{Nat} = \forall X. (X \rightarrow X) \rightarrow X \rightarrow X;$

$\text{zero} = (\lambda X. \lambda s:X \rightarrow X. \lambda z:X. z) \text{ as } \mathbb{C}\text{Nat};$

▶  $\text{zero} : \mathbb{C}\text{Nat}$

$\text{succ} = \lambda n:\mathbb{C}\text{Nat}. (\lambda X. \lambda s:X \rightarrow X. \lambda z:X. s (n [X] s z)) \text{ as } \mathbb{C}\text{Nat};$

▶  $\text{succ} : \mathbb{C}\text{Nat} \rightarrow \mathbb{C}\text{Nat}$

$\text{plus} = \lambda m:\mathbb{C}\text{Nat}. \lambda n:\mathbb{C}\text{Nat}. m [\mathbb{C}\text{Nat}] \text{succ } n;$

▶  $\text{plus} : \mathbb{C}\text{Nat} \rightarrow \mathbb{C}\text{Nat} \rightarrow \mathbb{C}\text{Nat}$

## 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 `List T` as a primitive type or as a recursive type. Can we encode it in the “vanilla” System F?

## Remark (Iterating over Lists)

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

$$\begin{aligned} \text{List}_{T_{11}} &= \forall S. ((\text{Unit} + T_{11} \times S) \rightarrow S) \rightarrow S; \\ \text{List}_{T_{11}} &= \forall S. ((\text{Unit} \rightarrow S) \times (T_{11} \times S \rightarrow S)) \rightarrow S; \\ \text{List}_{T_{11}} &= \forall S. (S \times (T_{11} \rightarrow S \rightarrow S)) \rightarrow S; \\ \text{List}_{T_{11}} &= \forall S. (T_{11} \rightarrow S \rightarrow S) \rightarrow S \rightarrow S; \end{aligned}$$



# Church Encodings: Lists

$\text{List}_T = \forall X. (T \rightarrow X \rightarrow X) \rightarrow X \rightarrow X;$

$\text{nil}_T = (\lambda X. \lambda c:(T \rightarrow X \rightarrow X). \lambda n:X. n) \text{ as List}_T;$

►  $\text{nil}_T : \text{List}_T$

$\text{cons}_T = \lambda \text{hd}:T. \lambda \text{tl}:\text{List}_T. (\lambda X. \lambda c:(T \rightarrow X \rightarrow X). \lambda n:X. c \text{ hd } (\text{tl } [X] \text{ c } n)) \text{ as List}_T;$

►  $\text{cons}_T : T \rightarrow \text{List}_T \rightarrow \text{List}_T$

$\text{isnil}_T = \lambda l:\text{List}_T. l [\text{Bool}] (\lambda _:T. \lambda _:Bool. \text{false}) \text{ true};$

►  $\text{isnil}_T : \text{List}_T \rightarrow \text{Bool}$

$\text{head}_T = \lambda l:\text{List}_T. l [T] (\lambda \text{hd}:T. \lambda _:T. \text{hd}) \text{ error};$

►  $\text{head}_T : \text{List}_T \rightarrow T$

## Question

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



# Church Encodings: Inductive Types

## Remark (Iteration)

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

## Principle

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

$$\mathbf{Ind}_{X. T} = \forall S. ([X \mapsto S]T \rightarrow S) \rightarrow S;$$

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

$$\blacktriangleright \mathbf{fold}_{X. T} : [X \mapsto \mathbf{Ind}_{X. T}]T \rightarrow \mathbf{Ind}_{X. T}$$

## Question

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



# Church Encodings: Streams

## Remark (Generation of Streams)

Previously, we define `Stream` as a coinductive type `coi(X. Nat × X)`.

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

## 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 → Nat × S`.

## Observation

From the perspective of **elimination**, one can use `S` and `S → Nat × S` to produce a value of some other type `T`.



# Church Encodings: Streams

## An Encoding of Streams

$\text{Stream} = \forall T. (\forall S. S \rightarrow (S \rightarrow \text{Nat} \times S) \rightarrow T) \rightarrow T;$

$\text{unfold}_{\text{Stream}} = \lambda v:\text{Stream}. v [\text{Nat} \times \text{Stream}]$   
 $(\lambda S. \lambda s:S. \lambda g:(S \rightarrow \text{Nat} \times S).$   
 $\quad \mathbf{let} \ v' = g \ s \ \mathbf{in}$   
 $\quad \{v'.1, (\lambda T. \lambda f:(\forall S. S \rightarrow (S \rightarrow \text{Nat} \times S) \rightarrow T). f [S] \ v'.2 \ g)\})$

►  $\text{unfold}_{\text{Stream}} : \text{Stream} \rightarrow \text{Nat} \times \text{Stream}$

## Question

Encode the generation rule [T-Gen-Stream] as  $\text{gen}_{\text{Stream}} : \forall S. S \rightarrow (S \rightarrow \text{Nat} \times S) \rightarrow \text{Stream}.$



# Church Encodings: Coinductive Types

## Remark (Generation)

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

## Principle

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

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

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

$$\blacktriangleright \mathbf{unfold}_{X.T} : \mathbf{Coi}_{X.T} \rightarrow [X \mapsto \mathbf{Coi}_{X.T}]T$$



# 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 and 23.5.2: prove preservation and progress of System F.

# Parametricity

## Observation

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

- If  $\emptyset \vdash t : \forall X. X \rightarrow X$ , then  $t$  **is** (essentially) the identity function.
- If  $\emptyset \vdash t : \forall X. X \rightarrow X \rightarrow X$ , then  $t$  **is** (essentially) either  $\text{tru}$  (i.e.,  $\lambda X. \lambda t:X. \lambda f:X. t$ ) or  $\text{fls}$  (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  $id : \forall X. X \rightarrow X$ , 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  $t \rightarrow t'$  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  $b : \forall X. X \rightarrow X \rightarrow X$ , 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 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  $\delta$  is a **type substitution** that assigns a closed type  $\delta(X)$  to each type variable  $X \in \Gamma$ . A type substitution  $\delta$  induces a substitution  $\hat{\delta}$  on types  $\hat{\delta}(T) \stackrel{\text{def}}{=} [X_1 \mapsto \delta(X_1), \dots, X_n \mapsto \delta(X_n)]T$ .
- The judgment  $\eta : \delta$  states that  $\eta$  is an **admissible property assignment** on  $\delta : \Gamma$  that assigns an admissible property  $\eta(X) : \delta(X)$  to each  $X \in \Gamma$ .

## Definition $\{t \in T [\eta : \delta]\}$

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

# Parametricity: The Unary Case

## Definition

- The judgment  $\gamma : \Gamma$  states that  $\gamma$  is a **term substitution** that assigns a closed term  $\gamma(x) : \Gamma(x)$  to each variable  $x \in \Gamma$ . A term substitution  $\gamma$  induces a substitution  $\hat{\gamma}$  on terms  $\hat{\gamma}(t) \stackrel{\text{def}}{=} [x_1 \mapsto \gamma(x_1), \dots, x_n \mapsto \gamma(x_n)]t$ .
- The judgment  $\gamma \in \Gamma [\eta : \delta]$  states that  $\gamma$  and  $\Gamma$  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  $id : \forall X. X \rightarrow X$ , 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  $id : \forall X. X \rightarrow X$ , 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  $id [T'] t' : T'$ .

## Proposition (A Free Theorem)

Let  $g : T \rightarrow T'$  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.,

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

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  $T = \forall X. X \rightarrow X$  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-AppTabs) 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

$$p = \{*\text{Nat}, \{a=5, f=\lambda x:\text{Nat}. \text{succ}(x)\}\}$$

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

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

## Example

The same pair  $p$  also has the type  $\{\exists X, \{a : X, f : X \rightarrow \text{Nat}\}\}$ .

In general, the typechecker cannot decide **how much information should be hidden**.

$$p = \{*\text{Nat}, \{a=5, f=\lambda x:\text{Nat}. \text{succ}(x)\}\} \text{ as } \{\exists X, \{a:X, f:X\rightarrow X\}\};$$

$$\blacktriangleright p : \{\exists X, \{a:X, f:X\rightarrow X\}\}$$

$$p1 = \{*\text{Nat}, \{a=5, f=\lambda x:\text{Nat}. \text{succ}(x)\}\} \text{ as } \{\exists X, \{a:X, f:X\rightarrow \text{Nat}\}\};$$

$$\blacktriangleright p1 : \{\exists X, \{a:X, f:X\rightarrow \text{Nat}\}\}$$



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

## Typing

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

## Example

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

$p_4 = \{*\text{Nat}, \{a=0, f=\lambda x:\text{Nat}. \text{succ}(x)\}\} \text{ as } \{\exists X, \{a:X, f:X \rightarrow \text{Nat}\}\};$

▶  $p_4 : \{\exists X, \{a:X, f:X \rightarrow \text{Nat}\}\}$

$p_5 = \{*\text{Bool}, \{a=\text{ture}, f=\lambda x:\text{Bool}. \text{if } x \text{ then } 1 \text{ else } 0\}\} \text{ as } \{\exists X, \{a:X, f:X \rightarrow \text{Nat}\}\};$

▶  $p_5 : \{\exists X, \{a:X, f:X \rightarrow \text{Nat}\}\}$



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

## Typing

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

## Example

`p4 = {*Nat, {a=0, f=λ x: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 (λ y:X. x.f y) x.a;`

▶ `1 : Nat`



# Subtlety of Elimination Rule

## Example

$p4 = \{*\text{Nat}, \{a=0, f=\lambda x:\text{Nat}. \text{succ}(x)\}\}$  **as**  $\{\exists X, \{a:X, f:X\rightarrow\text{Nat}\}\};$

►  $p4 : \{\exists X, \{a:X, f:X\rightarrow\text{Nat}\}\}$

**let**  $\{X,x\}=p4$  **in**  $\text{succ}(x.a);$

► Error: argument of  $\text{succ}$  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:

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

# Existential Types: Syntax and Evaluation



## Syntax

$$t ::= \dots \mid \{^*T, t\} \text{ as } T \mid \text{let } \{X, x\} = t \text{ in } t$$
$$v ::= \dots \mid \{^*T, v\} \text{ as } T$$
$$T ::= \dots \mid \{\exists X, T\}$$

## Evaluation

$$\frac{}{\text{let } \{X, x\} = (\{^*T_{11}, v_{12}\} \text{ as } T_1) \text{ in } t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2} \text{E-UnpackPack}$$
$$\frac{t_{12} \longrightarrow t'_{12}}{\{^*T_{11}, t_{12}\} \text{ as } T_1 \longrightarrow \{^*T_{11}, t'_{12}\} \text{ as } T_1} \text{E-Pack}$$
$$\frac{t_1 \longrightarrow t'_1}{\text{let } \{X, x\} = t_1 \text{ in } t_2 \longrightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2} \text{E-Unpack}$$

# 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  $\rightarrow$  Nat,

inc : Counter  $\rightarrow$  Counter;

**operations**

new = 1,

get =  $\lambda i:\text{Nat}. i$ ,

inc =  $\lambda i:\text{Nat}. \text{succ}(i)$ ;



# Translating ADTs to Existentials

```
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}}
```

  

```
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  =  $\lambda c:\text{Counter}.$  iseven (counter.get c),  
     toggle =  $\lambda c:\text{Counter}.$  counter.inc c,  
     reset =  $\lambda c:\text{Counter}.$  counter.new}}  
  as { $\exists$ FlipFlop,  
     {new:    FlipFlop, read: FlipFlop $\rightarrow$ Bool,  
       toggle: FlipFlop $\rightarrow$ FlipFlop, reset: FlipFlop $\rightarrow$ 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 = λ i:{x:Nat}. i.x,
    inc = λ i:{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**):

$$\text{Counter} = \{\exists X, \{\text{state}:X, \text{methods}: \{\text{get}:X \rightarrow \text{Nat}, \text{inc}:X \rightarrow X\}\}\};$$

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



# Existential Objects

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

▶ 5 : Nat

```
sendget =  $\lambda$  c:Counter.
```

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

▶ sendget : Counter  $\rightarrow$  Nat

```
let {X,body} = c in body.methods.inc(body.state);
```

▶ Error: scoping error!

```
sendinc =  $\lambda$  c:Counter.
```

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

▶ sendinc : Counter  $\rightarrow$  Counter



# ADTs vs. Objects

## ADTs

$\text{CounterADT} = \{\exists \text{Counter}, \{\text{new}:\text{Counter}, \text{get}:\text{Counter} \rightarrow \text{Nat}, \text{inc}:\text{Counter} \rightarrow \text{Counter}\}\}$

“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

$\text{Counter} = \{\exists X, \{\text{state}:X, \text{methods}:\{\text{get}:X \rightarrow \text{Nat}, \text{inc}:X \rightarrow X\}\}\}$

“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 =  $\lambda c1:Counter. \lambda c2:Counter. nat\_eq (counter.get\ c1) (counter.get\ c2)$   
in <rest of program>
```

## Object Style

```
let counter_eq =  $\lambda c1:Counter. \lambda c2:Counter.$   
  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

$$\text{NatSetADT} = \{\exists \text{NatSet}, \{\dots, \text{union}:\text{NatSet} \rightarrow \text{NatSet} \rightarrow \text{NatSet}\}\}$$

## Object Style

$$\text{NatSet} = \{\exists X, \{\text{state}:X, \text{methods}:\{\dots, \text{union}:X \rightarrow \text{NatSet} \rightarrow X\}\}\}$$

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

$$\text{NatSet} = \{\exists X, \{\text{state}:X, \text{methods}\{\dots, \text{union}:X \rightarrow X \rightarrow X\}\}\}$$

instead?

## Answer

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

```
sendunion = λ s1:NatSet. λ s2: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

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

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

$$\mathbf{*}S, t\} \mathbf{as} \{\exists X, T\} \stackrel{\text{def}}{=} \lambda S. \lambda f : (\forall X. T \rightarrow S). f [S] t \\ \mathbf{let} \{X, x\} = t_1 \mathbf{in} t_2 \stackrel{\text{def}}{=} t_1 [S] (\lambda X. \lambda x : T. t_2)$$

# Homework



## 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  $t \longrightarrow t'$ .

**And also:**

## Question

Show that under the encodings of existentials in System F, we have the following evaluation relation:

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