

# Today ... into Polymorphism .. 1. What is Polymorphism? 2. Type Inference (Reconstruction) 3. Unification 4. Let-Polymorphism 5. Conclusion

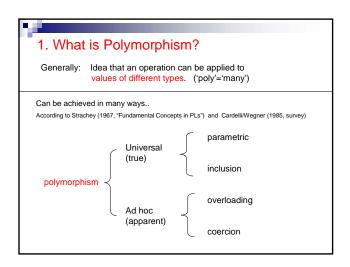
```
A Critique of Statically Typed PLs

→ Types are obtrusive: they overwhelm the code

→ Types inhibit code re-use: one version for each type.

doubl e_int = \( \lambda x \cdot \int \rightarrow \int \lambda y \cdot \int \lambda (x(y)) \\
doubl e_bool = \( \lambda x \cdot \int \rightarrow \int \lambda y \cdot \int \rightarrow \int \lambda (x(y))
```

## A Critique of Statically Typed PLs → Types are obtrusive: they overwhelm the code → Type Inference (Reconstruction) → Types inhibit code re-use: one version for each type. → Polymorphism



```
Ad Hoc Polymorphism

Overloading (resolved at compile-time. -- Overridden methods at run-time)

→ one name for different functions

→ only a conveniant syntax abbreviation

→ exampl e: +: int → int 1 + 2
+: real → real 1.0 + 2.0

Coercion (= compile away subtyping by run-time coercions)

((real 1) + 1.0 or 1 + 1.0
```

```
Universal Polymorphism

Inclusion = Subtype Polymorphism

→ One object belongs to many classes.
E.g., a colored point can be seen as a point.

Parametric Polymorphism

→ Use type variables

f = \( \lambda x : \int \rightarrow int. \lambda y : int. \( x(x(y)) \)
```

```
Universal Polymorphism

Inclusion = Subtype Polymorphism

→ One object belongs to many classes.
E.g., a colored point
can be seen as a point.

Parametric Polymorphism

→ Use type variables

f = \(\lambda x: int \rightarrow int. \lambda y: int. \(x) = x(x(y)) \)
bool \(\rightarrow bool
bool
```

```
Universal Polymorphism

Inclusion = Subtype Polymorphism

→ One object belongs to many classes.
E.g., a colored point
can be seen as a point.

Parametric Polymorphism

→ Use Type Variables

f = \( \lambda x \). \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \) \( \lambda y \): Y \( \lambda x \): Y
```

```
Universal Polymorphism

Inclusion = Subtype Polymorphism

One object belongs to many classes.
E.g., a colored point can be seen as a point.

Parametric Polymorphism

Use Type Variables

f = \lambda x: \quad X \quad . \lambda y: \quad Y \quad x(x(y))

"principal type" of f = \lambda x. \lambda y. \quad x(x(y))
```

```
Parametric Polymorphism

How to find the principal type of \(\lambda \times \lambda \times \lambda (x(y)) \)??

→ type check and accumulate constraints about the types of the variables
```

### Parametric Polymorphism How to find the principal type of λx: X. λy: Y. x(x(y)) ?? → type check and accumulate constraints about the types of the variables Type Variables Type checking x(y) requires that X = Y → Z Type checking x(x(y)) requires that X = Z → W

```
Parametric Polymorphism

How to find the principal type of λx: X. λy: Y. x(x(y)) ??

→ type check and accumulate constraints about the types of variables

Type Parameters

Type checking x(y) requires that X = Y → Z

Type checking x(x(y)) requires that X = Z → W

→ Z = Y and X = Y → Y (and result type is Y)

This process is called type inference or type reconstruction.
```

```
Parametric Polymorphism

How to find the principal type of λx: X. λy: Y. x(x(y)) ??

→ type check and accumulate constraints about the types of variables Type Parameters

Type checking x(y) requires that X = Y → Z constraints

Type checking x(x(y)) requires that X = Z → W

→ Z = Y and X = Y → Y (and result type is Y) smallest solution

This process is called type inference or type reconstruction.
```

```
2. Type Inference (Reconstruction)
For simply typed lambda calculus (with base types, Int and Bool)
A Type Substitution is a mapping from type variables to types.
E.g. σ = [X / bool, Y / X → X]
then σ X = bool and σ Y = X → X (applied simultaneously)
Composition σ ∘ γ "sigma after gamma"
(σ ∘ γ) S = σ(γ S)
σ ∘ γ := [X/σ(T) for X/T in γ, and X/T in σ with X ∉ dom(γ)]
```

```
2. Type Inference (Reconstruction)

Extend type substitution to environments Γ and terms t.

Lemma. Type substitution preserves typing:

if Γ ⊢ t: T then σΓ ⊢ σt: σT.

Proof. By induction on the structure of term t.

Example. x:X ⊢ λy:X→int. y x: int is derivable.

Applying σ = [ X / bool ] gives

x:bool ⊢ λy:bool→int. y x: int

which is also derivable.
```

```
2. Type Inference (Reconstruction)

\Gamma: environment t: term

A solution for (\Gamma, t) is a pair (\sigma, T) such that \sigma\Gamma \vdash \sigma t : T

Example: \Gamma = f : X, a : Y and t = fa

Then ([X/Y \rightarrow int], int)

([X/int \rightarrow int, Y \rightarrow int], int)

([X/Y \rightarrow Z], Z)

([X/Y \rightarrow Z, Z \rightarrow int], Z) are solutions of (\Gamma, t)
```

```
2. Type Inference (Reconstruction)

\Gamma: environment t: term

A solution for (\Gamma, t) is a pair (\sigma, T) such that \sigma\Gamma \vdash \sigma t : T

Find three different solutions for \Gamma = \varnothing and

t = \lambda x : X. \lambda y : Y. \lambda z : Z. (x \in Z) (x \in Z)
```

```
2. Type Inference (Reconstruction)

Γ: environment
t: term

A solution for (Γ, t) is a pair (σ, Τ) such that σΓ ⊢ σt: Τ

Constraint-Based Typing:
Given (Γ, t)

Calculate set of constraints that must be satisfied by ANY solution for (Γ, t)
```

```
2. Type Inference (Reconstruction)

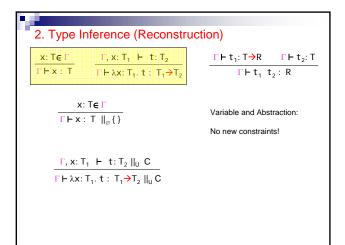
true: Bool \quad false: Bool \quad \underbrace{t_1: Bool \quad t_2: T \quad t_3: T}_{i \text{ } f \text{ } t_1 \text{ } then \text{ } t_2 \text{ } el \text{ } se \text{ } t_3: T}
\underline{t_1: Nat} \quad \underline{t_1: Nat} \quad \underline{t_1: Nat} \quad \underline{t_1: Nat} \quad \underline{i \text{ } sZero \text{ } t_1: Bool}
\underline{\Gamma \vdash t_1: T \mid_U C \quad C' = C \cup \{T = Nat\}}
\Gamma \vdash succ t_1: Nat \mid_U C'
```

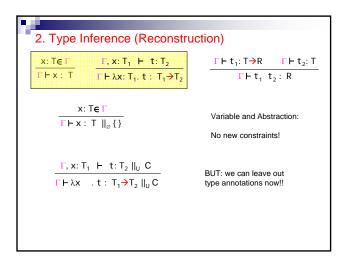
```
2. Type Inference (Reconstruction)

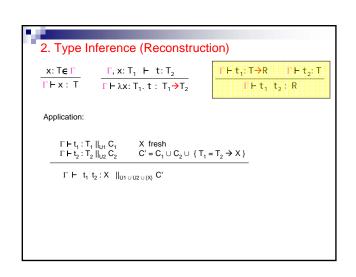
true: Bool false: Bool t_1: Bool t_2: T t_3: T

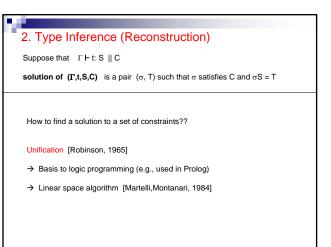
zero: Nat t_1: Nat t_2: t_3: t_4: Nat t_1: Nat t_2: t_3: t_4: Nat t_1: Nat t_2: t_3: t_4: Nat t_4:
```

```
2. Type Inference (Reconstruction)
                                      fal se: Bool
                                                                                  t_1: Bool t_2: T t_3: T
  true: Bool
                                                                                  ift<sub>1</sub> then t<sub>2</sub> else t<sub>3</sub>: T
  zero: Nat
                                                                                                     t<sub>1</sub>: Nat
     t<sub>1</sub>: Nat
                                                 t<sub>1</sub>: Nat
succ t<sub>1</sub>: Nat
                                            pred t<sub>1</sub> : Nat
                                                                                          i sZero t<sub>1</sub> : Bool
               \begin{array}{l} \Gamma \vdash t_1 : T_1 \mid\mid_{U1} C_1 \\ \Gamma \vdash t_2 : T_2 \mid\mid_{U2} C_2 \\ \Gamma \vdash t_3 : T_3 \mid\mid_{U3} C_3 \end{array}
                                                             U1, U2, U3 pairwise disjoint
                                                            C' = C_1 \cup C_2 \cup C_3 \cup \{ T_1 = Bool, T_2 = T_3 \}
                       \Gamma \, \vdash \, \mathsf{ift}_1 \, \, \mathsf{then} \, \mathsf{t_2} \, \mathsf{else} \, \mathsf{t_3} \, \colon \, \mathsf{T_2} \, \parallel_{\mathsf{U1} \, \cup \, \mathsf{U2} \, \cup \, \mathsf{U3}} \, \mathsf{C'}
```

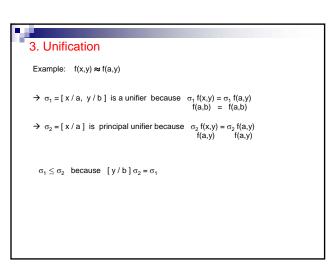








## 3. Unification → More precisely: syntactic equational unification → Define the set of terms t := x | f(t<sub>1</sub>, ..., t<sub>n</sub>) with x∈ Var and f∈ FuncSymbols → Given an equation s ≈ t we look for substitution σ such that σs ≈ σt (σ is called unifier for s ≈ t) σ₁ more general than σ₂ iff ∃ σ such that σ σ₁ = σ₂ Write σ₁ ≤ σ₂ (σ₂ can be obtained from σ₁!) Principal Unifier of s ≈ t is unifier σ s.t. for all unifiers σ⁺: σ ≤ σ⁺ Unification Theorem: s ≈ t has principal unifier, if it is unifiable!



```
3. Unification by Martelli, Montanari

R = \text{set of equations of the form } s \approx t
t \approx t, R \mid \sigma \Rightarrow_{MM} R \mid \sigma
f(...) \approx g(...), R \mid \sigma \Rightarrow_{MM} \bot \text{ if } f \neq g \text{ or Arity}(f) \neq \text{Arity}(g)
f(s_1,...,s_n) \approx f(t_1,...,t_n), R \mid \sigma \Rightarrow_{MM} s_1 \approx t_1, \ldots, s_n \approx t_n, R \mid \sigma
x \approx t, R \mid \sigma \Rightarrow_{MM} [x/t] R \mid [x/t] \sigma \text{ if } x \notin \text{var}(t)
(Self Occurence Check)
x \approx t, R \mid \sigma \Rightarrow_{MM} \bot \text{ if } x \in \text{var}(t)
t \approx x, R \mid \sigma \Rightarrow_{MM} x \approx t, R \mid \sigma
\emptyset \mid \sigma \Rightarrow_{MM} \sigma
set of constraints

Start with: C \mid []
empty substitution
```

```
3. Unification by Martelli, Montanari

Examples:

C1 = {X = int, Y = X → X}

C2 = {int→int = X → Y}

C3 = {X → Y = Y → Z, Z = U → W}

C4 = {int = int→Y}

C5 = {Y = int→Y}
```

```
3. Unification by Martelli, Montanari

Suppose that Γ⊢t: S || C

solution of (Γ,t,S,C) is a pair (σ, T) such that σ satisfies C and σS = T

→ Use MM - unification algorithm on C | []

→ If this returns substitution σ,

then σS is the principal type of t under Γ.
```

```
4. Let-Polymorphism

Let us now try to use this parametric function:

| let double = λx: Y → Y. λy: Y. x(x(y)) in {
| let a = double (λx: int. x+2) 2 in {
| let b = double (λx: bool. x) false in {..}
| }
| }
```

```
4. Let-Polymorphism

Let us now try to use this parametric function:

let double = λx: Y->Y. λy: Y. x(x(y)) in {
  let a = double (λx: int. x+2) 2 in {
  let b = double (λx: bool. x) false in {...}
  }
}

Γ + t<sub>1</sub>: T<sub>1</sub>  Γ, x: T<sub>1</sub> + t<sub>2</sub>: T<sub>2</sub>

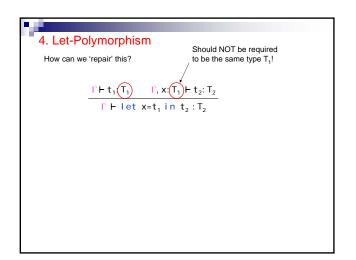
Γ + let x=t<sub>1</sub> in t<sub>2</sub>: T<sub>2</sub>
```

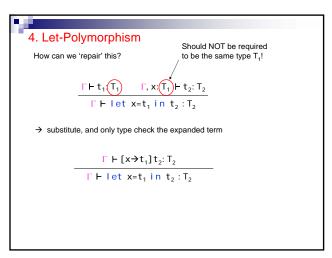
```
4. Let-Polymorphism

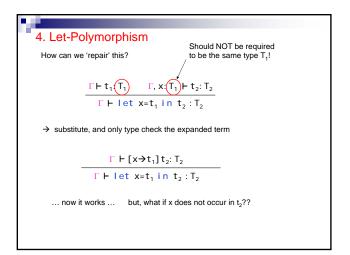
Let us now try to use this parametric function:

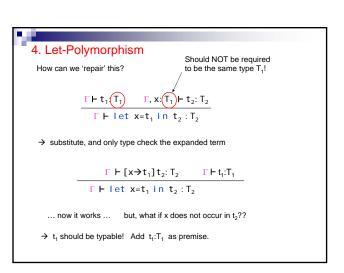
| let double = λx: Y→Y. λy: Y. x(x(y)) in {
| let a = double (λx: int. x+2) 2 in {
| let b = double (λx: bool. x) false in {...}
| }
| }
| Γ⊢ t<sub>1</sub>: T<sub>1</sub>  Γ, x: T<sub>1</sub> ⊢ t<sub>2</sub>: T<sub>2</sub>
| Γ⊢ let x=t<sub>1</sub> in t<sub>2</sub>: T<sub>2</sub>

Can NOT be typed!
| constraints: Y→Y = int→int AND Y→Y = bool→bool
```









```
4. Let-Polymorphism

Frequency

Frequency

Frequency

Frequency

Frequency

Frequency

Froblem with Let-Polymorphism:

If body of let contains many occ's of x, then it will be checked many times!

→ Design a more clever algorithm

Good algorithms in practice appear "essentially linear" ... but ....
```

```
4. Let-Polymorphism

... this OCaml program ..

let val f0 = fun x => (x, x) in

let val f1 = fun y => f0 (f0 y) in

let val f2 = fun y => f1 (f1 y) in

let val f3 = fun y => f2 (f2 y) in

let val f4 = fun y => f3 (f3 y) in

f4 (fun z => z)

.. is well-typed, but takes a **LONG** time to type check!
```

| 4. Let-Polymorphism          |                           |           |             |
|------------------------------|---------------------------|-----------|-------------|
| Program                      | Derived Type              | Type Size | Constraints |
| let val f0 =                 | ∀X0:X0→X0*X0              | 20        | 0           |
| fun $x \Rightarrow (x,x)$ in |                           |           |             |
| let val f1 = fun y =         | > ∀X1:X1 →(X1*X1)*(X1*X1) | ) 22      | 2           |
| f0 (f0 y) in                 |                           |           |             |
| let val f2 = fun y =         | > ∀X2:X2→((((X2*X2)*(X2*X | 2))* 24   | 4           |
| f1 (f1 y) in                 | ((X2*X2)*(X2*X            | (2)))*    |             |
| let val f3 = fun y =         | > (((X2*X2)*(X2*X         | 2))* 28   | 8           |
| f2 (f2 y) in                 | ((X2*X2)*(X2*X            | (2))))    |             |
| let val f4 = fun y =         | >                         | 216       | 16          |
| f3 (f3 y) in                 | ()                        |           |             |
| f4 (fun z => z)              |                           |           |             |
| end end end end              |                           |           |             |
|                              |                           |           |             |

### Conclusion

In simply-typed lambda-calculus, we can leave out ALL type annotations:

- → insert new type variables→ do type reconstruction (using unification)

In this way, changing the let-rule, we obtain

### Let-Polymorphism

- $\rightarrow$  Simple form of polymorphism
- → Introduced by [ Milner 1978 ] in ML
- ightarrow also known as Damas-Milner polymorphism
- → in ML, basis of powerful generic libraries (e.g., lists, arrays, trees, hash tables, ...)

### 4. Conclusion

With let-polymorphism, only let-bound values can be used polymorphically.  $\lambda$ -bound values cannot be used polymorphically.

Let  $f = \lambda g$ . ... g(1)... g(true)... in  $f(\lambda x. x)$ Example:

is not typable: when typechecking the definition of  $\mathbf{f}$ , g has type X (a fresh type variable) which is then constrained by  $X = int \rightarrow Y$  and  $X = bool \rightarrow Z$ 

Functions cannot take polymorphic functions as parameters. This is the key limitation of let-polymorphism.

→ Can this be fixed/generalized?? YES: System F (next time)! Polymorphic Lambda Calculus

### 4. Conclusion

Next time: → polymorphic lambda-calculus (system F) (15.12.)

ightarrow polymorphic lambda-calculus + subtyping = "Bounded Quantification" (System "F-sub"  $F_{<:}$ )

→ written assignment will be distributed

(to be handed in by 22.12.)

22.12.: → adding generics to FJ (= FGJ)

The programming assignment to be done by 21.01. is about implementing FGJ!