

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 = λx: int→int. λy: int. x(x(y))
doubl e_bool = λx: bool →bool .λy: bool x(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

1. What is Polymorphism?

Generally: Idea that an operation can be applied to values of different types. ('poly'='many')

Can be achieved in many ways..

According to Strachey (1967, "Fundamental Concepts in PLs") and Cardelli/Wegner (1985, survey)

parametric

universal (true)

polymorphism

Ad hoc (apparent)

overloading

coercion

```
Ad Hoc Polymorphism

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

→ one name for different functions

→ only a conveniant syntax abbreviation

→ example: +: 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 = λx: int→int. λy: 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

I ass CPt extends Pt {
col or c;
CPt(int x, int y, col or c) {
super (x, y);
this. c = c;
} col or getc () { return this.c; }

Y . 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 = \( \lambda x \)

#principal type" of f = \( \lambda x \) \( \lambda x \) \( \lambda x \)

#principal type" of f = \( \lambda x \) \( \lamb
```

```
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 \(\lambda x: X. \lambda y: Y. \times (x(y)) \)??

→ 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.
```

```
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 for 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 = f a

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)
```

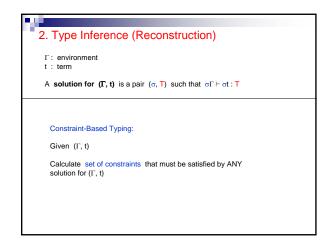
```
2. Type Inference (Reconstruction)

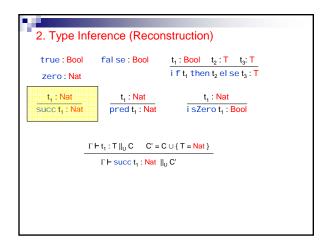
Γ: environment
t: term

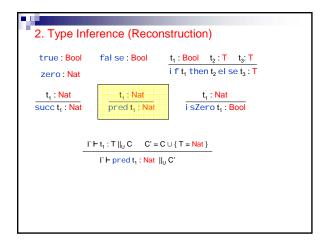
A solution for (Γ, t) is a pair (σ, Τ) such that σΓ ⊢ σt: Τ

Find three different solutions for Γ = Ø and

t = λx: X. λy: Y. λz: Z. (x z) (y z)
```



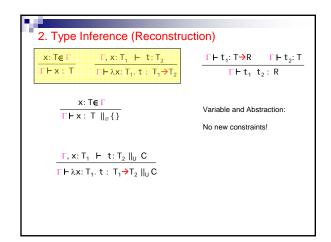


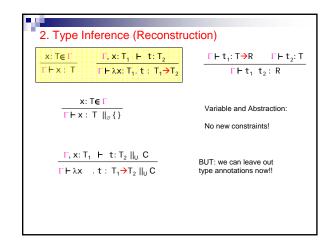


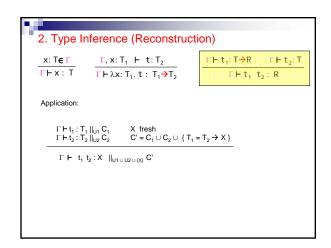
```
2. Type Inference (Reconstruction)

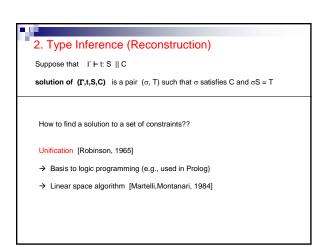
true : Bool \qquad false : Bool \qquad \underline{t_1 : Bool} \qquad \underline{t_2 : T \quad t_3 : T}
zero : Nat \qquad \qquad \underline{t_1 : Nat} \qquad \underline{t_1 : Nat} \qquad \underline{t_1 : Nat}
\underline{t_1 : Nat} \qquad \underline{t_1 : Nat} \qquad \underline{t_1 : Nat} \qquad \underline{t_1 : Nat}
\underline{succ \ t_1 : Nat} \qquad \underline{pred \ t_1 : Nat} \qquad \underline{i \ sZero \ t_1 : Bool}
\underline{\Gamma \vdash t_1 : T \mid_{U} C \quad C' = C \cup \{T = Nat\}}
\overline{\Gamma \vdash i \ sZero \ t_1 : Bool} \mid_{U} C'
```

```
2. Type Inference (Reconstruction)
 true: Bool
                                      fal se: Bool
                                                                                 t_1: Bool t_2: T t_3: T
                                                                                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_1: Nat
succ t<sub>1</sub>: Nat
                                           pred t<sub>1</sub>: Nat
                                                                                          i sZero t<sub>1</sub>: Bool
              \begin{array}{c|c} \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 = \frac{Bool}{T_1}, 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{U}1 \, \cup \, \mathsf{U}2 \, \cup \, \mathsf{U}3} \, \mathsf{C}'
```









3. Unification

→ More precisely: syntactic equational unification

→ Define the set of terms $t := x \mid f(t_1, ..., t_n)$ with $x \in Var$ and $f \in FuncSymbols$ → Given an equation $s \approx t$ we look for substitution σ such that $\sigma s \approx \sigma t$ (σ is called unifier for $s \approx t$) σ_1 more general than σ_2 iff $\exists \sigma$ such that $\sigma \sigma_1 = \sigma_2$ Write $\sigma_1 \leq \sigma_2$ (σ_2 can be obtained from σ_1 !)

Principal Unifier of $s \approx t$ is unifier σ s.t. for all unifiers σ : $\sigma \leq \sigma$ '

Unification Theorem: $s \approx t$ has principal unifier, if it is unifiable!

3. Unification

Example: $f(x,y) \approx f(a,y)$ $\Rightarrow \sigma_1 = [x/a, y/b]$ is a unifier because $\sigma_1 f(x,y) = \sigma_1 f(a,y)$ f(a,b) = f(a,b) $\Rightarrow \sigma_2 = [x/a]$ is principal unifier because $\sigma_2 f(x,y) = \sigma_2 f(a,y)$ f(a,y) f(a,y) $\sigma_1 \leq \sigma_2$ because $[y/b] \sigma_2 = \sigma_1$

```
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, ..., 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

z \in S_{MM} = S_{MM} = S_{MM} \times S_{MM}
```

```
3. Unification by Martelli, Montanari

Examples:

C1 = {X = int, Y = X \rightarrow X}

C2 = {int \rightarrow int = X \rightarrow Y}

C3 = {X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W}

C4 = {int = int \rightarrow Y}

C5 = {Y = int \rightarrow 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 = \( \lambda x: Y \rightarrow Y. \lambda y: X \rightarrow (x \rightarrow y) \) in {
| let a = double (\( \lambda x: \text{int.} \limbda x + 2) \) 2 in {
| let b = double (\( \lambda x: \text{bool.} \limbda x) \) false in {...}
| }
| }
| \[
\frac{\Gamma + \text{t_1: T_1} \quad \Gamma, x: T_1 \rightarrow t_2: T_2}{\Gamma + \text{let } x = t_1 \text{in } t_2: T_2}
\]
```

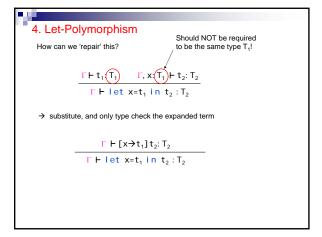
```
4. Let-Polymorphism

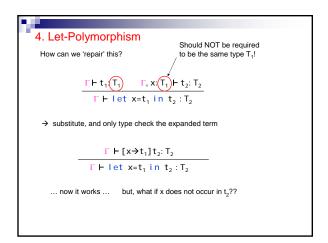
Let us now try to use this parametric function:

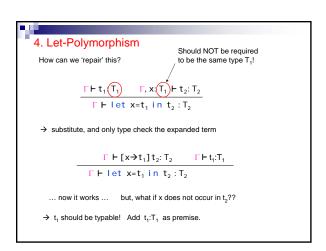
| let double = λx: Y→Y. λy: Y. x(x(y)) in {
| let a = double (λx: int. x+2) 2 in {
| let b = double (λx: bool. x) false in {..} }
| }
| \[
\frac{\Gamma \Gamma \text{t}_1: T_1 \quad \Gamma \text{x}: T_1 \quad \text{t}_2: T_2}{\Gamma \Gamma \text{let} \text{x} = \text{t}_1 \text{ in } \text{t}_2: T_2}

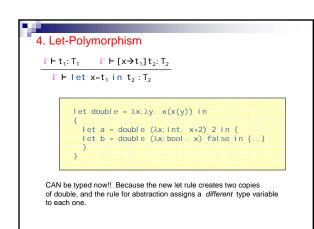
| Can NOT be typed!
| constraints: Y→Y = int→int AND Y→Y = bool→bool
```

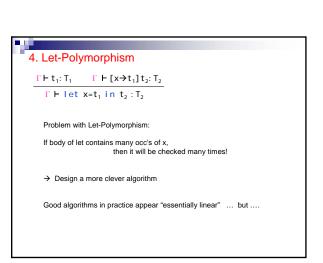
4. Let-Polymorphism How can we 'repair' this? Should NOT be required to be the same type $T_1!$ $\frac{\Gamma \vdash t_1(T_1) \qquad \Gamma, x(T_1) \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$











```
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

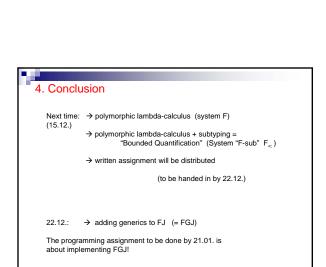
Let val f4 = fun y => f3 (f3 y) in

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
fl (fl y) in	((X2*X2)*(X2*X2)))*		
let val f3 = fun y =>	(((X2*X2)*(X2*X	(2))* 28	8
f2 (f2 y) in	((X2*X2)*(X2*X2))))		
let val f4 = fun y =>		2 ¹⁶	16
f3 (f3 y) in	()		
f4 (fun z => z)			
end end end end			

4. 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 → Simple form of polymorphism → Introduced by [Milner 1978] in ML → 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. λ-bound values cannot be used polymorphically. Example: let f = λg. ... g(1)... g(true)... in f(λx. x) is not typable: when typechecking the definition of f, g has type X (a fresh type variable) which is then constrained by X = int→Y and X = bool→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