Type Systems

Lecture 8 Dec. 8th, 2004

Sebastian Maneth

http://lampwww.epfl.ch/teaching/typeSystems/2004

Important:

The FJ Programming Assignment is only due

tomorrow, Dec. 9th, at 17:00.

→ send code to `burak.emir@epfl.ch`

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.

```
double_int =  λx:int→int.λy:int. x(x(y))
double_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)

- polymorphism
  - Universal (true)
    - parametric
    - inclusion
  - 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.

```
class CPt extends Pt {
  color c;
  CPt(int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

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.

```
class CPt extends Pt {
  color c;
  CPt(int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

Parametric Polymorphism

→ Use type variables

```
f =  λx:int→int.λy:int. x(x(y))
        bool→bool   bool
```

Universal Polymorphism

Inclusion = Subtype Polymorphism

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

```
class CPt extends Pt {
  color c;
  CPt(int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

Parametric Polymorphism

→ Use Type Variables

```
f =  λx:   X    .λy: 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.

```
class CPt extends Pt {
  color c;
  CPt(int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

Parametric Polymorphism

→ Use Type Variables

```
f =  λx:   X    .λy: Y . x(x(y))
          Y→Y        Y
```

"principal type" of `f = λx.λy. x(x(y))`

Parametric Polymorphism

How to find the principal type of `λx.λy. x(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

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

constraints

➔ 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 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 $\Gamma \vdash t : T$ then $\sigma\Gamma \vdash \sigma t : \sigma T$.

Proof. By induction on the structure of term t.

Example. $x{:}X \vdash \lambda y{:}X \to int.\ y\ x : int$ is derivable.

Applying $\sigma = [\ X / bool\ ]$ gives

$x{:}bool \vdash \lambda y{:}bool \to int.\ y\ x : int$

which is also derivable.

2. Type Inference (Reconstruction)

Γ : environment
t : term

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

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

Then $(\ [\ X / Y \to int\ ],\ int\ )$

$(\ [\ X / int \to int,\ Y \to int\ ],\ int\ )$

$(\ [\ X / Y \to Z],\ Z\ )$

$(\ [\ X / Y \to Z,\ Z \to int\ ],\ Z\ )$ are solutions of (Γ, t)

2. Type Inference (Reconstruction)

Γ : environment
t : term

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

Find three different solutions for $\Gamma = \varnothing$ and

$t = \lambda x{:}X.\ \lambda y{:}Y.\ \lambda z{:}Z.\ (x\ z)\ (y\ z)$

2. Type Inference (Reconstruction)

Γ : environment
t : term

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

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   false : Bool $\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$

zero : Nat

$\dfrac{t_1 : Nat}{\text{succ } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{pred } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{isZero } t_1 : Bool}$

$$\dfrac{\Gamma \vdash t_1 : T \;||_U\; C \qquad C' = C \cup \{ T = Nat \}}{\Gamma \vdash \text{succ } t_1 : Nat \;||_U\; C'}$$

2. Type Inference (Reconstruction)

true : Bool   false : Bool $\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$

zero : Nat

$\dfrac{t_1 : Nat}{\text{succ } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{pred } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{isZero } t_1 : Bool}$

$$\dfrac{\Gamma \vdash t_1 : T \;||_U\; C \qquad C' = C \cup \{ T = Nat \}}{\Gamma \vdash \text{pred } t_1 : Nat \;||_U\; C'}$$

2. Type Inference (Reconstruction)

true : Bool   false : Bool $\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$

zero : Nat

$\dfrac{t_1 : Nat}{\text{succ } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{pred } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{isZero } t_1 : Bool}$

$$\dfrac{\Gamma \vdash t_1 : T \;||_U\; C \qquad C' = C \cup \{ T = Nat \}}{\Gamma \vdash \text{isZero } t_1 : Bool \;||_U\; C'}$$

2. Type Inference (Reconstruction)

true : Bool   false : Bool $\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$

zero : Nat

$\dfrac{t_1 : Nat}{\text{succ } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{pred } t_1 : Nat}$ $\dfrac{t_1 : Nat}{\text{isZero } t_1 : Bool}$

$$\dfrac{\begin{array}{ll}\Gamma \vdash t_1 : T_1 \;||_{U1}\; C_1 & \text{U1, U2, U3 pairwise disjoint} \\ \Gamma \vdash t_2 : T_2 \;||_{U2}\; C_2 & \\ \Gamma \vdash t_3 : T_3 \;||_{U3}\; C_3 & C' = C_1 \cup C_2 \cup C_3 \cup \{ T_1 = Bool, T_2 = T_3 \}\end{array}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \;||_{U1 \cup U2 \cup U3}\; C'}$$

2. Type Inference (Reconstruction)

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x{:}T_1 \vdash t{:}T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1{:}T \to R \quad \Gamma \vdash t_2{:}T}{\Gamma \vdash t_1\ t_2 : R}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T \ \|_{\varnothing} \{\}}$$

Variable and Abstraction:

No new constraints!

$$\frac{\Gamma, x{:}T_1 \vdash t{:}T_2 \ \|_U\ C}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2 \ \|_U\ C}$$

2. Type Inference (Reconstruction)

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x{:}T_1 \vdash t{:}T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1{:}T \to R \quad \Gamma \vdash t_2{:}T}{\Gamma \vdash t_1\ t_2 : R}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T \ \|_{\varnothing} \{\}}$$

Variable and Abstraction:

No new constraints!

$$\frac{\Gamma, x{:}T_1 \vdash t{:}T_2 \ \|_U\ C}{\Gamma \vdash \lambda x\quad .t : T_1 \to T_2 \ \|_U\ C}$$

BUT: we can leave out type annotations now!!

2. Type Inference (Reconstruction)

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x{:}T_1 \vdash t{:}T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1{:}T \to R \quad \Gamma \vdash t_2{:}T}{\Gamma \vdash t_1\ t_2 : R}$$

Application:

$$\frac{\begin{array}{ll}\Gamma \vdash t_1 : T_1 \ \|_{U1}\ C_1 & X \text{ fresh} \\ \Gamma \vdash t_2 : T_2 \ \|_{U2}\ C_2 & C' = C_1 \cup C_2 \cup \{ T_1 = T_2 \to X \}\end{array}}{\Gamma \vdash t_1\ t_2 : X \ \|_{U1 \cup U2 \cup \{X\}}\ C'}$$

2. Type Inference (Reconstruction)

Suppose that $\Gamma \vdash t: S \ \|\ C$

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

How to find a solution to a set of constraints??

Unification [Robinson, 1965]

→ Basis to logic programming (e.g., used in Prolog)

→ Linear space algorithm [Martelli,Montanari, 1984]

3. Unification

→ More precisely: syntactic equational unification

→ Define the set of terms
$t ::= x \mid f(t_1, \dots, t_n)$ 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: $f(x,y) \approx f(a,y)$

→ $\sigma_1 = [\, x / a,\ y / b\,]$ is a unifier because $\sigma_1 f(x,y) = \sigma_1 f(a,y)$
$f(a,b) = f(a,b)$

→ $\sigma_2 = [\, x / a\,]$ is principal unifier because $\sigma_2 f(x,y) = \sigma_2 f(a,y)$
$f(a,y) \quad f(a,y)$

$\sigma_1 \leq \sigma_2$ because $[\, y / b\,]\, \sigma_2 = \sigma_1$

3. Unification by Martelli, Montanari

R = set of equations of the form $s \approx t$

$t \approx t, R \mid \sigma \Rightarrow_{MM} R \mid \sigma$

$f(\dots) \approx g(\dots), R \mid \sigma \Rightarrow_{MM} \bot$ if $f \neq g$ or $Arity(f) \neq Arity(g)$

$f(s_1,\dots,s_n) \approx f(t_1,\dots,t_n), R \mid \sigma \Rightarrow_{MM} s_1 \approx t_1, \dots, s_n \approx t_n, R \mid \sigma$

$x \approx t, R \mid \sigma \Rightarrow_{MM} [x / t]\, R \mid [x / t]\, \sigma$ if $x \notin var(t)$
(Self Occurence Check)

$x \approx t, R \mid \sigma \Rightarrow_{MM} \bot$ if $x \in var(t)$

$t \approx x, R \mid \sigma \Rightarrow_{MM} x \approx t, R \mid \sigma$

$\emptyset \mid \sigma \Rightarrow_{MM} \sigma$

Start with: C | [ ]

(C: set of constraints; [ ]: 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 $\Gamma \vdash t: S \ || \ C$

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

→ 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 {..}
  }
}
```


$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

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 {..}
  }
}
```

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

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!

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

4. Let-Polymorphism

How can we 'repair' this?

Should NOT be required to be the same type T_1!

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

→ substitute, and only type check the expanded term

$$\frac{\Gamma \vdash [x \to t_1] t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

4. Let-Polymorphism

How can we 'repair' this?

Should NOT be required to be the same type T_1!

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

→ substitute, and only type check the expanded term

$$\frac{\Gamma \vdash [x \to t_1] t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

… now it works … but, what if x does not occur in t_2??

4. Let-Polymorphism

How can we 'repair' this?

Should NOT be required to be the same type T_1!

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

→ substitute, and only type check the expanded term

$$\frac{\Gamma \vdash [x \to t_1] t_2 : T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{let } x=t_1 \texttt{ in } t_2 : T_2}$$

… now it works … but, what if x does not occur in t_2??

→ t_1 should be typable! Add $t_1 : T_1$ as premise.

4. Let-Polymorphism

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [x \rightarrow t_1]t_2 : T_2}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T_2}$$

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

CAN be typed now!! Because the new let rule creates two copies of double, and the rule for abstraction assigns a *different* type variable to each one.

4. Let-Polymorphism

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [x \rightarrow t_1]t_2 : T_2}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T_2}$$

Problem 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	2^0	0
`fun x => (x,x) in`			
`let val f1 = fun y =>`	∀X1:X1 →(X1*X1)*(X1*X1)	2^2	2
`f0 (f0 y) in`			
`let val f2 = fun y =>`	∀X2:X2→((((X2*X2)*(X2*X2))*	2^4	4
`f1 (f1 y) in`	((X2*X2)*(X2*X2)))*		
`let val f3 = fun y =>`	(((X2*X2)*(X2*X2))*	2^8	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 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

4. Conclusion

Next time: (15.12.)

→ polymorphic lambda-calculus (system F)

→ 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!