# CS 476 – Programming Language Design

William Mansky

## Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

# From Typed Lambda Calculus to OCaml

- User-friendly syntax
- Basic types, tuples, records
- Inductive datatypes and pattern-matching
- Local declarations
- References
- Type inference

➢Generics/polymorphism

# Constraint-Based Type Inference

- Step 1: gather constraints, outputs pair $(\tau, C)$ such that if $C$ can be solved, $\tau$ is the type of the expression
- Step 2: unify constraints $C$, obtain solving substitution $\sigma$
- Step 3: apply $\sigma$ to $\tau$ to get the type of the expression

```
let type_of (gamma : context) (e : exp) =
    let (t, c) = get_constraints gamma e in
    let s = unify c in apply_subst s t
```

# Constraint-Based Type Inference: Example

$$\{\} \vdash (\texttt{fun f -> fun x -> f x + f 3}) : \tau_1 \rightarrow \tau_2 \rightarrow \text{int} \mid C$$

$$C = \{\tau_3 = \text{int}, \tau_4 = \text{int}, \tau_1 = \tau_2 \rightarrow \tau_3, \tau_1 = \text{int} \rightarrow \tau_4\}$$

$$\sigma = \{\tau_3 \mapsto \text{int}, \tau_4 \mapsto \text{int}, \tau_1 \mapsto \text{int} \rightarrow \text{int}, \tau_2 \mapsto \text{int}\}$$

$$[\sigma](\tau_1 \rightarrow \tau_2 \rightarrow \text{int}) = (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$$

$$\{\} \vdash (\texttt{fun f -> fun x -> f x + f 3}) : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$$

# Universal Polymorphism

- What happens when we do type inference and end up with variables in the final type?

inferred type for `fun x -> 5` : $\tau_1 \rightarrow \mathbf{int}$

inferred type for `fun x -> x` : $\tau_1 \rightarrow \tau_1$

- What should the type checker do in this case?

# Universal Polymorphism

- What happens when we do type inference and end up with variables in the final type?

inferred type for `fun x -> 5` : $\tau_1 \rightarrow \mathbf{int}$

inferred type for `fun x -> x` : $\tau_1 \rightarrow \tau_1$

- We could fail, and ask the user to specify the type of the argument

# Universal Polymorphism

- What happens when we do type inference and end up with variables in the final type?

inferred type for `fun x -> 5` : $\tau_1 \rightarrow \mathbf{int}$

inferred type for `fun x -> x` : $\tau_1 \rightarrow \tau_1$

- We could let the user apply `fun x -> x` to any input!

```
(fun x -> x) 1 = 1          (fun x -> x) true = true
(fun x -> x) (fun y -> y) = fun y -> y
```

# Universal Polymorphism

- What happens when we do type inference and end up with variables in the final type?

```
let id = fun x -> x;;
val id : ‘a -> ‘a = <fun>
```

- `‘a` is OCaml’s way of writing a type variable
- Read as “type inference has inferred that this function can be generic”

# Universal Polymorphism: Examples

```
let id x = x;;
val id : ‘a -> ‘a = <fun>

let f x y = y;;
val f : ‘a -> ‘b -> ‘b = <fun>

let g f x = f x;;
val g : (‘a -> ‘b) -> ‘a -> ‘b = <fun>
```

# Universal Polymorphism: Examples

```
let id = fun x -> x;;
id : ∀a. a -> a

let f x y = y;;
f : ∀a, b. a -> b -> b

let g f x = f x;;
g : ∀a, b. (a -> b) -> a -> b
```

# Universal Polymorphism: Examples

```
let update f x v = fun y ->
    if y = x then Some v else f y;;
```

(* update : ('a -> 'b option) -> 'a -> 'b -> ('a -> 'b option) *)

```
type context = ident -> typ option
type env = ident -> value option

update (gamma : context) x t
update (r : env) x v
```

## Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

# Universal Polymorphism

- Universal polymorphism (also *generic*, or *parametric*): a type can have any number of *universally quantified* variables
- A function can be applied at any *instantiation* of its type
- Happens when a function *doesn't care* about the type of an argument

$$\frac{\Gamma[x \mapsto \tau_1] \vdash l : \tau_2 \mid C \quad \tau_1 \text{ fresh}}{\Gamma \vdash \texttt{fun } x \texttt{ -> } l : \tau_1 \to \tau_2 \mid C}$$

and we end up with no constraints on $\tau_1$ in $C$

  — So the function will do the same thing with an input of any type
  — Compare to generics in C/Java, contrast with OO polymorphism

# Universal Polymorphism

$$\frac{?}{\Gamma \vdash \texttt{let id = fun x -> x in (id 1 = 1) \&\& (id true)} : ?}$$

# Universal Polymorphism

$$\frac{\ldots \quad \Gamma[\texttt{id} \mapsto a \to a] \vdash \texttt{(id 1 = 1) \&\& (id true)} : ?}{\Gamma \vdash \texttt{let id = fun x -> x in (id 1 = 1) \&\& (id true)} : ?}$$

# Universal Polymorphism

- We said "we learn type constraints from the ways variables are used", but that's not true for polymorphic functions!

$$\frac{\ldots \quad \Gamma[\texttt{id} \mapsto a \rightarrow a] \vdash \texttt{(id 1=1) \&\& (id true)} : \text{bool} \mid C}{\Gamma \vdash \texttt{let id = fun x -> x in (id 1=1) \&\& (id true)} : \text{bool} \mid C}$$

where $C = \{a \rightarrow a = \text{int} \rightarrow \text{int}, a \rightarrow a = \text{bool} \rightarrow \text{bool}\}$

Unsolvable!

# Universal Polymorphism: Typing

- There are now two kinds of types:
  - A *monomorphic* type, or *monotype*, doesn't have quantifiers
  - A *polymorphic* type, or *polytype*, is $\forall a_1 \ldots a_n . \tau$ where $\tau$ is a monotype (that uses $a_1 \ldots a_n$)
- When should we assign a polytype to a term?
- *Let-polymorphism*: only at `let` definitions

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

# Universal Polymorphism: Typing

- There are now two kinds of types:
  - A *monomorphic* type, or *monotype*, doesn't have quantifiers
  - A *polymorphic* type, or *polytype*, is $\forall a_1 \dots a_n . \tau$ where $\tau$ is a monotype (that uses $a_1 \dots a_n$)
- When should we assign a polytype to a term?
- *Let-polymorphism*: only at let definitions

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall a_1 \dots a_n . \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

# Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

# Let-Polymorphism: Typing

What variables should we quantify?

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall \overbrace{a_1 \ldots a_n}. \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = l_1\ \mathtt{in}\ l_2 : \tau}$$

$a \to a$ (type of `fun x -> x`)

$\mathrm{id} : \forall a. a \to a$ (in `(id 1 = 1) && (id bool)`)

```
let id = fun x -> x in (id 1 = 1) && (id bool)
```

# Let-Polymorphism: Typing

where $\mathrm{vars}(\tau_1) = a_1, \ldots, a_n$

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall \overbrace{a_1 \ldots a_n}. \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

$a \to a$ (annotating `fun x -> x`)

$\mathrm{id} : \forall a. a \to a$ (annotating `(id 1 = 1) && (id bool)`)

```
let id = fun x -> x in (id 1 = 1) && (id bool)
```

# Let-Polymorphism: Typing

where $\text{vars}(\tau_1) = a_1, \ldots, a_n$

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall \underbrace{a_1 \ldots a_n}. \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

- The type of y has to be int, not generic

$\Gamma(y) = a$ $\quad b \to a$ $\quad \mathrm{f} : \forall a\ b. b \to a$

```
fun y -> (let f = fun x -> y in y + f 3)
```

# Let-Polymorphism: Typing

where $\text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n$

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall \overbrace{a_1 \ldots a_n}.\tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

- The type of **y** has to be int, not generic

$\Gamma(y) = a$ $\qquad$ $b \to a$ (for `fun x -> y`) $\qquad$ $\text{f} : \forall b. b \to a$ (for `y + f 3`)

```
fun y -> (let f = fun x -> y in y + f 3)
```

# Let-Polymorphism: Typing

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n \qquad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

# Let-Polymorphism: Typing

$$\frac{\Gamma \vdash l_1 : \tau_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \dots, a_n \qquad \Gamma[x \mapsto \forall a_1 \dots a_n . \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau}$$

$$\frac{\Gamma(x) = \forall a_1 \dots a_n . \tau \quad [a_1 \mapsto \tau_1, \dots, a_n \mapsto \tau_n]\tau = \tau'}{\Gamma \vdash x : \tau'}$$

- We can have polytypes in $\Gamma$, but in $\Gamma \vdash e : \tau$, $\tau$ is a monotype

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n \quad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\Gamma(x) = \forall a_1 \ldots a_n . \tau \quad [a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n]\tau = \tau'}{\Gamma \vdash x : \tau'}$$

- We can have polytypes in $\Gamma$, but in $\Gamma \vdash e : \tau$, $\tau$ is a monotype

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n \quad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\Gamma(x) = \forall a_1 \ldots a_n . \tau \quad b_1, \ldots, b_n \text{ fresh}}{\Gamma \vdash x : [a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\tau \mid \{\}}$$

- We can have polytypes in $\Gamma$, but in $\Gamma \vdash e : \tau$, $\tau$ is a monotype

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \mathrm{vars}(\tau_1) - \mathrm{vars}(\Gamma) = a_1, \ldots, a_n \qquad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{}{\{\mathrm{y} : \mathrm{a}\} \vdash \texttt{let f = fun x -> y in y + f 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \mathrm{vars}(\tau_1) - \mathrm{vars}(\Gamma) = a_1, \ldots, a_n \qquad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\{\mathtt{y} : \mathtt{a}\} \vdash \texttt{fun x -> y} : ? \mid ?}{\{\mathtt{y} : \mathtt{a}\} \vdash \texttt{let f = fun x -> y in y + f 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \dots, a_n \qquad \Gamma[x \mapsto \forall a_1 \dots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\{\mathrm{y} : \mathrm{a}\} \vdash \texttt{fun x -> y} : \mathrm{b} \to \mathrm{a} \mid \{\}}{\{\mathrm{y} : \mathrm{a}\} \vdash \texttt{let f = fun x -> y in y + f 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n \qquad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\{y : a\} \vdash \texttt{fun x -> y} : b \to a \mid \{\} \qquad \{y : a, f : ?\} \vdash \texttt{y + f 3} : ? \mid ?}{\{y : a\} \vdash \texttt{let f = fun x -> y in y + f 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \mathrm{vars}(\tau_1) - \mathrm{vars}(\Gamma) = a_1, \dots, a_n \qquad \Gamma[x \mapsto \forall a_1 \dots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\{\mathrm{y} : \mathrm{a}\} \vdash \texttt{fun x -> y} : \mathrm{b} \to \mathrm{a} \mid \{\} \qquad \{\mathrm{y} : \mathrm{a}, \mathrm{f} : \forall \mathrm{b}. \mathrm{b} \to \mathrm{a}\} \vdash \texttt{y + f 3} : ? \mid ?}{\{\mathrm{y} : \mathrm{a}\} \vdash \texttt{let f = fun x -> y in y + f 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma \vdash l_1 : \tau_1 \mid C_1 \quad \text{vars}(\tau_1) - \text{vars}(\Gamma) = a_1, \ldots, a_n \qquad \Gamma[x \mapsto \forall a_1 \ldots a_n . \tau_1] \vdash l_2 : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ x = l_1\ \texttt{in}\ l_2 : \tau \mid C_1 \cup C_2}$$

$$\frac{\ldots \qquad \dfrac{\dfrac{\{\mathrm{y} : \mathrm{a}, \mathtt{f} : \forall \mathrm{b}. \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{f} : ?}{\ldots}}{\{\mathrm{y} : \mathrm{a}, \mathtt{f} : \forall \mathrm{b}. \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{y\ +\ f\ 3} : ? \mid ?}}{\{\mathrm{y} : \mathrm{a}\} \vdash \mathtt{let\ f\ =\ fun\ x\ \text{->}\ y\ in\ y\ +\ f\ 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma(x) = \forall a_1 \dots a_n.\tau \quad b_1, \dots, b_n \text{ fresh}}{\Gamma \vdash x : [a_1 \mapsto b_1, \dots, a_n \mapsto b_n]\tau \mid \{\}}$$

$$\frac{\cdots \qquad \dfrac{\dfrac{\{\mathrm{y} : \mathrm{a}, \mathtt{f} : \forall \mathrm{b}.\, \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{f} : ?}{\cdots}}{\{\mathrm{y} : \mathrm{a}, \mathtt{f} : \forall \mathrm{b}.\, \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{y\ +\ f\ 3} : ? \mid ?}}{\{\mathrm{y} : \mathrm{a}\} \vdash \mathtt{let\ f\ =\ fun\ x\ \text{->}\ y\ in\ y\ +\ f\ 3} : ? \mid ?}$$

# Let-Polymorphism: Type Inference

$$\frac{\Gamma(x) = \forall a_1 \ldots a_n . \tau \quad b_1, \ldots, b_n \text{ fresh}}{\Gamma \vdash x : [a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\tau \mid \{\}}$$

a and c will be int, but b stays quantified

$$\frac{\ldots \quad \dfrac{\dfrac{\{\mathrm{y} : \mathrm{a}, \mathrm{f} : \forall \mathrm{b}.\, \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{f} : \mathrm{c} \to \mathrm{a} \mid \{\}}{\ldots}}{\{\mathrm{y} : \mathrm{a}, \mathrm{f} : \forall \mathrm{b}.\, \mathrm{b} \to \mathrm{a}\} \vdash \mathtt{y\ +\ f\ 3} : ? \mid ?}}{\{\mathrm{y} : \mathrm{a}\} \vdash \mathtt{let\ f\ =\ fun\ x\ ->\ y\ in\ y\ +\ f\ 3} : ? \mid ?}$$

# Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

# More Universal Polymorphism

- With let-polymorphism, we can have polytypes in $\Gamma$, but when $\Gamma \vdash e : \tau$, $\tau$ is a monotype

```
let id = fun x -> x in (id 1 = 1) && (id true);;
let g f = (f 1 = 1) && (f true);;
(* Type error: f takes an int, not a bool *)
```

- In let-polymorphism, a polytype never appears as an *argument*

# More Universal Polymorphism

- With let-polymorphism, we can have polytypes in $\Gamma$, but when $\Gamma \vdash e : \tau$, $\tau$ is a monotype
- With full universal polymorphism, polytypes are first-class types

```
let g f x y = (f x = 1) && (f y);;
```

(* g : $(\forall a. a \rightarrow a) \rightarrow \text{int} \rightarrow \text{bool} \rightarrow \text{bool}$ *)

- There is no type inference algorithm for full universal polymorphism!
- We need to explicitly instantiate the polytypes at each use

# System F

$T$ ::= $T \to T$ | <tident> | ∀<tident>. $T$

$L$ ::= $\lambda$<ident>:$T$. $L$ | $L$ $L$ | <ident> | Λ<tident>. $L$ | $L$ [$T$]

```
let id = Λa.λx:a.x
(id [int] 1 = 1) && (id [bool] true)
```

$$\frac{\Gamma \vdash l : \tau}{\Gamma \vdash \Lambda a. l : \forall a. \tau} \qquad \frac{\Gamma \vdash l : \forall a. \tau_1}{\Gamma \vdash l\,[\tau] : [a \mapsto \tau]\tau_1}$$

- Used in some versions of Haskell, dependently-typed languages

## Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

# Universal Polymorphism

- In OCaml, a function with free type variables gets a *universal* type, and can be used at any *instantiation* of its type
- Happens when a function *doesn't care* about the type of an argument, and will do the same thing with an input of any type
- Requires only a small change to type checking/inference to automatically infer when a function can be generic
- More general universal polymorphism is possible, but if we go too general, we lose automatic type inference!

# From Typed Lambda Calculus to OCaml

- User-friendly syntax
- Basic types, tuples, records
- Inductive datatypes and pattern-matching
- Local declarations
- References
- Type inference
- Generics/polymorphism