CS 476 – Programming Language Design

William Mansky
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
Universal Polymorphism

• What’s the inferred type for fun x -> x?
  — \( \tau_1 \rightarrow \tau_1 \), with no constraints
  — In other words, \( \tau_1 \rightarrow \tau_1 \) for any \( \tau_1 \)!

let id = fun x -> x in
(id 1 = 1) && (id true)
Universal Polymorphism

• What’s the inferred type for fun x -> x?
  — \( \tau_1 \rightarrow \tau_1 \), with no constraints
  — In other words, \( \tau_1 \rightarrow \tau_1 \) for any \( \tau_1 \)!

let id = fun x -> x;;
(* val id : 'a -> 'a = <fun> *)

• ‘a is OCaml’s way of writing a type variable
let id = fun 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 *)
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

d update (gamma : context) x t update (r : env) x v ...
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

\[ \Gamma[x \mapsto \tau_1] \vdash l : \tau_2 \mid C \quad \tau_1 \text{ fresh} \]

\[ \Gamma \vdash \text{fun } x \rightarrow l : \tau_1 \rightarrow \tau_2 \mid C \]

— So the function will do the same thing with an input of any type
— Contrast with OO polymorphism
Universal Polymorphism

\[
\Gamma \vdash \text{let } \text{id} = \text{fun } x \rightarrow x \text{ in (id 1 = 1) } \&\& \text{ (id true) : ?}
\]

\[
\text{... } \Gamma[id \mapsto a \rightarrow a] \vdash (\text{id 1 = 1}) \&\& \text{(id true)} : ?
\]
Universal Polymorphism

\[
\begin{align*}
\ldots & \quad \Gamma[id \mapsto a \rightarrow a] \vdash (id \ 1 = 1) \ & \& \ (id \ true) : C \\
\Gamma & \vdash \text{let } id = \text{fun } x \rightarrow x \text{ in } (id \ 1 = 1) \ & \& \ (id \ true) : C
\end{align*}
\]

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 local definitions

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

\[
\frac{\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall a_1 \ldots a_n. \tau_1] \vdash l_2 : \tau}{\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau}
\]
Let-Polymorphism: Typing

What variables should we quantify?

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

\[
\begin{align*}
\text{id} : & \forall a. \ a \rightarrow a \\
\text{let id = fun x -> x in (id 1 = 1) && (id bool)}
\end{align*}
\]
Let-Polymorphism: Typing

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

$$
\begin{align*}
\Gamma &\vdash l_1 : \tau_1 \\
\Gamma[x \mapsto \forall a_1 \ldots a_n. \tau_1] &\vdash l_2 : \tau \\
\Gamma &\vdash \text{let } x = l_1 \text{ in } l_2 : \tau
\end{align*}
$$

$$
\begin{align*}
a \to a \\
id : \forall a. a \to a
\end{align*}
$$

let id = fun x -> x in (id 1 = 1) && (id bool)
Let-Polymorphism: Typing

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

\[
\begin{array}{c}
\Gamma \vdash l_1 : \tau_1 \\
\Gamma [x \mapsto \forall a_1 \ldots a_n. \tau_1] \vdash l_2 : \tau \\
\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau
\end{array}
\]

\( \Gamma(y) = a \quad b \to a \quad f : \forall a \ b.\ b \to a \)

fun y -> (let f = fun x -> y in y + f 3)
Let-Polymorphism: Typing

where $fv(\tau_1) = a_1, \ldots, a_n$

\[
\Gamma \vdash l_1 : \tau_1 \quad \Gamma[x \mapsto \forall a_1 \ldots a_n. \tau_1] \vdash l_2 : \tau
\]

\[
\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau
\]

\[
\Gamma(y) = a \quad b \rightarrow a \quad f : \forall b. b \rightarrow a
\]

fun y -> (let f = fun x -> y in y + f 3)
Let-Polymorphism: Typing

where $\text{fv}(\tau_1) - \text{fv}(\Gamma) = a_1, ..., a_n$

\[
\frac{
\Gamma \vdash l_1 : \tau_1 \\
\Gamma \vdash \forall a_1 \ldots a_n. \tau_1 \vdash l_2 : \tau \\
\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau
}{\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau}
\]

\[
\Gamma(y) = a \quad b \rightarrow a \quad f : \forall b. b \rightarrow a
\]

fun y -> (let f = fun x -> y in y + f 3)
Let-Polymorphism: Typing

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

\[ \Gamma(x) = \tau \]
\[ \Gamma \vdash x : \tau \]
Let-Polymorphism: Typing

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

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

• We can have polytypes in \( \Gamma \), but in \( \Gamma \vdash e : \tau \), \( \tau \) is a monotype
Let-Polymorphism: Type Inference

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

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

• We can have polytypes in \( \Gamma \), but in \( \Gamma \vdash e : \tau \), \( \tau \) is a monotype
Let-Polymorphism: Type Inference

Let \( \Gamma \vdash l_1 : \tau_1 \mid C_1 \)  \( \text{fv}(\tau_1) - \text{fv}(\Gamma) = a_1, \ldots, a_n \)

\[ \Gamma \vdash x \mapsto \forall a_1 \ldots a_n. \tau_1 \mid l_2 : \tau \mid C_2 \]

\[ \Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau \mid C_1 \cup C_2 \]

\[ \Gamma(x) = \forall a_1 \ldots a_n. \tau \text{ b}_1, \ldots, b_n \text{ fresh} \]

\[ \Gamma \vdash x : [a_1 \mapsto b_1, \ldots, a_n \mapsto b_n] \tau \mid \{\} \]

- We can have polytypes in \( \Gamma \), but in \( \Gamma \vdash e : \tau \), \( \tau \) is a monotype
More Universal Polymorphism

• With let-poly-polymorphism, we can have polytypes in $\Gamma$, but when $\Gamma \vdash e : \tau$, $\tau$ is a monotype
• This is what OCaml does

```ocaml
let g f x y = (f x = 1) && (f y); ;
(* Type error: f takes an int, not a bool *)
```

• In let-poly-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

\[ T ::= T \to T \mid \text<tident> \mid \forall \text<tident>. \; T \]
\[ L ::= \lambda \text<ident>:T. \; L \mid L \; L \mid \text<ident> \mid \Lambda \text<tident>. \; L \mid L \; [T] \]

let id = \Lambda a. \lambda x: a. x  
\hspace{1cm} (id \; [\text{int}] \; 1 = 1) \; \&\& \; (id \; [\text{bool}] \; \text{true})

\[ \Gamma \vdash l : \tau \quad \Rightarrow \quad \Gamma \vdash \Lambda a. \; l : \forall a. \tau \]
\[ \Gamma \vdash l : \forall a. \tau_1 \quad \Rightarrow \quad \Gamma \vdash l \; [\tau] : [a \mapsto \tau]\tau_1 \]

- Used in some versions of Haskell, dependently-typed languages
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
• More general universal polymorphism is possible, but if we go too general, we lose 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