Type Inference I: An Introduction

CAS CS 320: Principles of Programming Languages

November 19, 2024 (Lecture 21)

Practice Problem

```
fun f -> fun x -> f (x + 1)
```

What is the type of the above OCaml Expression?

Today

- Discuss type inference and polymorphism with an eye towards Hindley-Milner type systems
- Look at a set of typing rules for constraint-based inference in Hindley-Minler (light) (HM^{-})
- Briefly discuss let-polymorphism
- Walk through some examples of type inference using our typing rules

Learning Objectives

- Derive the type of an expression given the rules for HM$^-$ or something similar
- Define the constraint-based inference rule for a language construct (e.g., pairs)
- Determine the type scheme of any OCaml expression

Outline

Explicit Typing

In mini-project 2, we're implementing a programming language with **explicit typing**:

```
let add (x : int) (y : int) : int = x + y
let k (x : int) (y : bool) : int = x
let _ : unit = assert(add 2 3 = k 5 false)
```

Every function argument and let expression is annotated with typing information. This closer to what is done in a language like Java

Implicit Typing

We rarely have to specify types in OCaml:

```
let add x y = x + y
let k x y = x
let _ = assert(add 2 3 = k 5 false)
```

Type inference, or type reconstruction is process of determining what types we *could* have annotated our program with.

But what type should we give k ?

Parametric Polymorphism

```
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

In OCaml we can write polymorphic functions which are *agnostic* in some way to their input type

They are parameterized by the type of the input, in that they must behave exactly the same way on all inputs, independent of the type itself. In particular, we can't "dispatch" according to type

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The parameters in our types are called type variables

The variables are *instantiated* when we apply our function to an argument which means we need to define substitution for types(!)

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

Just like with expression variables, we don't like *unbound* type variables (if they're unbound it means we don't know what they refer to)

Types variables in OCaml are quantified (and the syntax is hidden)

We read this as "`id` has type `t -> t` for any type `t`". As we will see, we'll also write $\forall \alpha . \alpha \rightarrow \alpha$ for `’a .  ’a -> ’a`

Aside: Second-Order Lambda Calculus

```
(* NOT VALID OCAML *)
let id_int : int -> int = fun x -> x
let id : 'a . 'a -> 'a =
  fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F (or the second-order lambda calculus) was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s.

The basic idea: One way of allowing polymorphism is to allow the introduction of types into the language which we can pass as arguments to functions

The problem: Without type annotations type checking is undecidable

Hindley-Milner Type Systems

Hindley-Milner type systems (HM) are variants of the lambda calculus with parametric polymorphism

They underlie nearly all functional programming languages currently in use (like OCaml, Haskell, and Elm)

They allow for a restricted form of type quantification, in which quantifiers always appear in the "outermost" position.

Type inference is decidable and (fairly) efficient.

Type Inference (High-Level)

Infering the type of an expresion e in a context Γ follows the rough procedure:

1. Derive $\Gamma \vdash e : \tau$ relative to some constraints $\mathcal{C}$
2. Use the constraints to determine τ' the "actual" type of e in Γ
3. Make τ' into a polymorphic type by quantifying over the free type variables in τ' to get $\forall\alpha_1 \ldots, \forall\alpha_k.\tau'$
4. We then conclude $\Gamma \vdash e : \forall\alpha_1 \ldots, \forall\alpha_k.\tau'$

Example (by Intuition)

```
fun f -> fun x -> f (x + 1)
```

Outline

Syntax (BNF)

```
<expr> ::= fun <var> -> <expr> | <expr> <expr>
         |  let <var> = <expr> in <expr>
         |  if <expr> then <expr> else <expr>
         |  <expr> + <expr> | <expr> = <expr>
         |  <int> | <var>
 <mty> ::= int  | bool | <tyvar> | <mty> -> <mty>
  <ty> ::= <tyvar>.<ty> | <mty>
```

The syntax of HM$^-$ is similar to our previous languages but with the addition of type variables and type quantification

Syntax (Mathematical)

$$
\begin{array}{rcl}
e & ::= & \lambda x.e \mid ee \\
 & \mid & \text{let } x = e \text{ in } e \\
 & \mid & \text{if } e \text{ then } e \text{ else } e \\
 & \mid & e + e \mid e = e \\
 & \mid & num \mid x \\
\sigma & ::= & \text{int} \mid \text{bool} \mid \sigma \rightarrow \sigma \mid \alpha \\
\tau & ::= & \sigma \mid \forall \alpha . \tau
\end{array}
$$

As usual, we'll often use concise mathematical notation for writing down inference rules and derivations.

Type Variables and Type Schemes

$$\begin{array}{rcl} \sigma & ::= & \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid \alpha \\ \tau & ::= & \sigma \mid \forall \alpha . \tau \end{array}$$

σ represents monotypes, types with no quantification. A type is monomorphic if it is a monotype with no type variables

τ represents type schemes, which are types with some number of quantified type variables.

Free Variables (Monotypes)

$$\begin{aligned} FV(\mathsf{int}) &= \varnothing \\ FV(\mathsf{bool}) &= \varnothing \\ FV(\alpha) &= \{\alpha\} \\ FV(\tau_1 \to \tau_2) &= FV(\tau_1) \cup FV(\tau_2) \end{aligned}$$

Once we introduce variables, we have to again talk about free and bound variables

We will only need to consider free variables of monotypes for HM$^-$ so **there is no issue with variable capture**

Understanding Check

Define substitution $[\tau_1/\alpha]\tau_2$ for (mono)types

Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

Our typing rules will need to keep track of a set of constraints ($\mathcal{C}$) which will be used to determine the actual type of e

Contexts are collections of variables declarations, i.e., mapping of variables to type schemes

Reminder. once we "solve" the contraints and get the "actual" type τ' and generalize (i.e., "polymorphize") it to $\forall \alpha_1 \dots \forall \alpha_k . \tau'$, we'll write

$$\Gamma \vdash e : \forall \alpha_1 \dots \forall \alpha_k . \tau'$$

What is a constraint?

$$\tau_1 = \tau_2$$

In general, a type constraint is a predicate on types. The only kind of constraint we will consider is that of the form

“t_1 is equal to t_2"

Enforcing a constraint like this means unifying τ_1 and τ_2

Type System (High-Level)

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

The idea: For each rule, we need to determine:

- what is the *most general* type τ we could give e?
- what must be true of τ, i.e., what *constraints* τ?

If we don't know what type something should be we create a fresh type variable for it (we'll see some examples)

Type System (Literals and Variables with Monotypes)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \mathsf{int} \dashv \varnothing} \text{ (int)} \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma \dashv \varnothing} \text{ (var)}$$

Literals and variables (with monotypes) have their expected types *without any constraints* (pretty much the same as before. . .)

Type System (Operators)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 + e_2 : \mathsf{int} \dashv \tau_1 = \mathsf{int}, \tau_2 = \mathsf{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (add)}$$

$e_1 + e_2$ is an int if the types of e_1 and e_2 can be *unified* to int

We don't require that they have type *exactly* int but rather that they are *constrained* to have type int

Type System (If-Expressions)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_3 \dashv \tau_1 = \text{bool}, \tau_2 = \tau_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3} \text{ (if)}$$

An if-expression has the same type as it's else-case when:

- the type of its condition can be unified with bool
- the type of its then-case is the same as that of its else-case

Understanding Check

$$\{x : \alpha, y : \beta\} \vdash \text{if } x \text{ then } x \text{ else } y : \tau \dashv \mathcal{C}$$

Determine τ and $\mathcal{C}$ in the above judgment

$$\frac{n \textbf{ is an integer}}{\Gamma \vdash n : \textbf{int} \dashv \varnothing}\ (\textbf{int}) \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma \dashv \varnothing}\ (\textbf{var})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau_3 \dashv \tau_1 = \textbf{bool}, \tau_2 = \tau_3, \mathcal{C}_1, \mathcal{C}_2}\ (\textbf{if})$$

Type System (Functions)

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}} \ (\text{fun})$$

The input type of a function is *some* type α and it's output type is the type of the body

We don't know the input type, so we give it the most general form, i.e., a fresh type variable with no constraints

Type System (Application)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_2 : \alpha \dashv \tau_1 = \tau_2 \to \alpha, \mathcal{C}_1, \mathcal{C}_2} \text{ (app)}$$

The type of an application is some type α, such that the type of the function unifies to a *function type* with output type α, and the input type matches the type of the argument (wordy. . .)

Type System (Variables)

$$\frac{(x : \forall\alpha_1.\forall\alpha_2 \dots \forall\alpha_k.\tau) \in \Gamma \qquad \beta_1, \dots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1] \dots [\beta_k/\alpha_k]\tau} \text{ (var)}$$

If x is declared in Γ, then x can be given the type τ with all free variables replaced by fresh variables

This is where the magic happens with respect to polymorphism:

FRESH VARIABLES CAN BE UNIFIED WITH ANYTHING

A Small Example

Let's consider $\{f : \forall\alpha.\alpha \to \alpha\} \vdash f\ 2 : ? \dashv ?$

Type System (Let-expressions)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv C_2}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau_2 \dashv C_1, C_2} \text{ (let)}$$

The type of a let-expression is the same as the type of its body, relative to the constraints of typing the let-defining value and the body (wordy. . .)

An Aside: Let-Polymorphism (To Come)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv C_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \dashv C_1, C_2} \text{ (let)}$$

This rule does not allow let-defined functions to be polymorphic! (We'll see an example in a moment)

This is why we our system is called HM^- (i.e., Hindley-Milner Light)

There are some interesting culture wars in the world of PL with regards to let-polymorphism. . .

Example (Let-Polymorphism Fails)

```
let f = fun x -> x in
let y = f 2 in
f true
```

Example (More Involved)

```
fun f -> fun x -> f (x + 1)
```

Up Next

We still need to:

- introduce a unification algorithm in order to determine the "final" type given a collection of constraints
- introduce polymorphism for top-level let-expressions
- reintroduce type annotations so that we can annotate if we want to

We **won't**:

- deal with type errors (it's a lot more complicated for unification based algorithms)
- handle let-polymorphism (though it's not awful, if you're interested in trying)