Type Inference II: Unification

CAS CS 320: Principles of Programming Languages

November 21, 2024 (Lecture 22)

Practice Problem

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

Determine the type τ and constraints $\mathcal{C}$ such that the above judgment is derivable in HM^-.

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

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

Answer

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

Today

- Finish up our discussion on Hindley-Minler (light) (HM^-)
- Briefly discuss let-polymorphism
- Walk through more examples of type inference
- Learn the unification algorithm used to determine the "actual" type of our expression given a collection of constraints

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
- Determine how a sequence of constraints is unified

Outline

Recap: Type Inference

Hindley-Milner (Light)

Unification

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

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 $\forall\alpha.\alpha \to \alpha$ for `'a . 'a -> 'a`

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.

Outline

Recap: Type Inference

Hindley-Milner (Light)

Unification

Recall: Syntax (Mathematical)

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

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

Recall: Type Variables and Type Schemes

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

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.

Recall: Constraint-Based Inference

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

Our typing rules will need to keep track of a set of EQUALITY 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 $\forall\alpha_1 \ldots \forall\alpha_k.\tau'$, we'll write

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

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 *constrains* 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)

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

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

Type System (Operators)

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

$e_1 + e_2$ 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

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)

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

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

Example

$$\{f : \forall\alpha.\alpha \to \alpha\} \vdash \mathsf{if}\ (f\ \mathsf{true})\ \mathsf{then}\ (f\ 2)\ \mathsf{else}\ 0 : \tau \dashv \mathcal{C}$$

Example (Again)

$$\{f : \alpha \to \alpha\} \vdash \mathsf{if}\ (f\ \mathsf{true})\ \mathsf{then}\ (f\ 2)\ \mathsf{else}\ 0 : \tau \dashv \mathcal{C}$$

Type System (Let-expressions)

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

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

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

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 (Familiar)

```
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)

Outline

Recap: Type Inference

Hindley-Milner (Light)

Unification

High-Level

Unification is the process of solving a system of equations over *symbolic* expressions

It's kind of like solving a system of linear equations, but instead of working over real numbers and addition, we work over uninterpreted operations

The best way to think of it (in my opinion): unification is solving a system of equations over variables and ADT constructors

The Simple Case: Just Variables

If we just have variables, then unification is equivalent to connected components over undirected graphs

Unification Problem

Informal. Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (this can be made formal using ideas from logic and algebra)

A **unification problem** is a collection of equations of the form

$$\begin{aligned} s_1 &\doteq t_1 \\ s_2 &\doteq t_2 \\ &\vdots \\ s_k &\doteq t_k \end{aligned}$$

where $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ are terms.

Type Unification

```
type ty =
  | TInt
  | TBool
  | TFun of ty * ty
  | TVar of string
```

Type unification is the particular unification problem over the `ty` ADT (with type variable acting as variables in the sense from the previous slide)

Unifiers

Given a unification problem $\mathcal{U}$, a **solution** or **unifier** is a sequence of substitutions to some of the variables which appear in $\mathcal{U}$, typically written

$$\mathcal{S} = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}$$

We then write $\mathcal{S}t$ for the term $[t_n/x_n] \ldots [t_1/x_1]t$

A solution has the property that it *satisfies* every equation

$$\begin{aligned} \mathcal{S}t_1 &= \mathcal{S}s_1 \\ \mathcal{S}s_2 &= \mathcal{S}t_2 \\ &\vdots \\ \mathcal{S}s_k &= \mathcal{S}t_k \end{aligned}$$

Example

$$a \doteq d \to e$$
$$c \doteq \mathsf{int} \to d$$
$$\mathsf{int} \to \mathsf{int} \to \mathsf{int} \doteq b \to c$$

Aside: Logic Programming

Unification is fundamental is programming languages like Prolog

Logic programming is a not-very-common paradigm which uses logical specifications and unification to define *computations* (very cool stuff)

An Algorithm

The idea. We process equations one at a time, updating *the collection of equations themselves*. We **FAIL** if we every reach an unsatisfiable equation. There are three success cases:

Let $\mathcal{S}$ be an empty solution. Given a type unification problem $\mathcal{U}$, if $\mathcal{U}$ has the equation:

- int $\doteq$ int or bool $\doteq$ bool, then remove it from $\mathcal{U}$
- $s_1 \to s_2 \doteq t_1 \to t_2$, then remove it and add $s_1 \doteq t_1$ and $s_2 \doteq t_2$ to $\mathcal{U}$
- $\alpha \doteq t$ or $t \doteq \alpha$ where does not appear free in t
  - remove it
  - add $\alpha \mapsto t$ to $\mathcal{S}$
  - perform the substitution $\alpha \mapsto t$ in *every* equation remaining in $\mathcal{U}$
- otherwise, **FAIL**

Repeat until $\mathcal{U}$ is empty

Most General Unifiers

The **most general unifier** of a unification problem is the solution $\mathcal{S}$ such that, for any other solution $\mathcal{S}'$, there is another solution $\mathcal{S}''$ such that

$$\mathcal{S}' = \mathcal{S}\mathcal{S}''$$

In other words, every other unifier is $\mathcal{S}$ with more substitutions

Our algorithm is guaranteed to return the most general unifier

Principle Types

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

The constraints $\mathcal{C}$ define a unification problem

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ can get the "actual" type of e, **it's** $\mathcal{S}\tau$

If $\mathcal{S}$ is the most general unifier than $\mathcal{S}\tau$ is called the **principle type** of e. The principle type has the property that every other type is an *instance* of it

Example (Familiar)

$$
\begin{aligned}
a &\doteq d \to e \\
c &\doteq \mathsf{int} \to d \\
\mathsf{int} \to \mathsf{int} \to \mathsf{int} &\doteq b \to c
\end{aligned}
$$

Example (Unification Failure)

$$\cdot \vdash \lambda x.xx : \tau \dashv \mathcal{C}$$