Lecture #24: Hindley-Milner Type-inference

CS 6371: Advanced Programming Languages

April 14, 2020

System F requires the programmer to write typing annotations for all functions ($\lambda x{:}\tau.e$), all polymorphic abstractions ($\Lambda\alpha.e$), and all polymorphic applications ($e[\tau]$). It would be nice to relieve the programmer of this burden by inferring suitable types τ automatically if they exist. This is called *type-inference*. Unfortunately, general type-inference for System F is provably undecidable [1]. Thus, there is no algorithm by which suitable types can be inferred automatically for all System F programs.

As a compromise, OCaml supports a restricted version of System F for which type-inference is decidable. The restriction is that in OCaml all functions must have *shallow types*:

Definition 1. A System F type is *shallow* if it has the form $\forall\alpha_1 \ldots \forall\alpha_n.\tau$ where $n \geq 0$ and τ is simply typed (i.e., it contains no $\forall$ quantifiers).

Informally, this definition says that all quantifiers in types must appear at the outermost level. For example, $\forall\alpha.(\alpha \to \mathit{int})$ is a shallow type, but $(\forall\alpha.\alpha) \to \mathit{int}$ is not because the $\forall$ in this latter type appears inside an arrow type. This restriction is why when OCaml writes a type, it does not need to print $\forall$ symbols; all typing variables are implicitly universally quantified at the outermost level.

The type-inference algorithm for OCaml is based on *Hindley-Milner type-inference* (cf., [2]). What follows is a simplified version of the H-M algorithm that does not support recursive functions. (Polymorphic recursive functions are not difficult to add, but will not be covered in this class for the sake of time.) We will consider a language that includes integers, variables, abstractions, applications, and polymorphic applications of named (previously defined) functions f:

$$e ::= n \mid v \mid \lambda v.e \mid e_1 e_2 \mid f$$

The type-inference algorithm can be organized into four steps:

Step 1: Annotate all abstractions and polymorphic applications in e with *unique* type variables α as follows, forming a new *annotated expression* d:

$$d ::= n \mid v \mid \lambda v{:}\alpha.d \mid d_1 d_2 \mid f[\alpha_1]\cdots[\alpha_i]$$

Here, i is the number of universally quantified type variables in the type of function f (which has shallow type by assumption).

Step 2: Infer a type τ and mapping $\theta : \alpha \rightharpoonup \tau$ that satisfies the judgment $\bot, \Gamma_0 \vdash d : \tau, \theta$ whose semantics are given on the next page. Here, initial typing context Γ_0 is one that maps all named functions f defined so far in the program to their types.

Step 3: Take expression d from Step 1 and substitute all typing variables $\alpha \in \theta^{\leftarrow}$ with the types $\theta(\alpha)$ inferred in Step 2. (For any typing variable $\alpha \notin \theta^{\leftarrow}$, leave it alone in d.)

Step 4: In general there may be some typing variables $\alpha \notin \theta^{\leftarrow}$ that are still in the expression after Step 3. These type variables correspond to types for which no constraints were found during Step 2. They can therefore be universally quantified, so add a polymorphic binding ($\Lambda\alpha. \ldots$) for each one to the beginning of the expression.

The core of the above algorithm is Step 2, which we formally define below. In what follows, notation $\theta(\tau)$ denotes substituting all type variables $\alpha \in \theta^{\leftarrow}$ in τ with the types $\theta(\alpha)$ assigned to them by θ, and notation $\theta(\Gamma) = \{(v, \theta(\Gamma(v))) \mid v \in \Gamma^{\leftarrow}\}$ denotes performing the same substitution throughout the image of Γ.

$$\theta, \Gamma \vdash n : int, \theta \tag{1}$$

$$\theta, \Gamma \vdash v : \Gamma(v), \theta \tag{2}$$

$$\frac{\Gamma(f) = \forall\beta_1 \ldots \forall\beta_i.\tau}{\theta, \Gamma \vdash f[\alpha_1] \cdots [\alpha_i] : \tau[\alpha_1/\beta_1] \cdots [\alpha_i/\beta_i], \theta} \tag{3}$$

$$\frac{\theta, \Gamma[v \mapsto \alpha] \vdash d : \tau, \theta'}{\theta, \Gamma \vdash \lambda v{:}\alpha.d : \alpha \to \tau,\ \theta'} \tag{4}$$

$$\frac{\theta, \Gamma \vdash d_1 : \tau_1, \theta_1 \qquad \theta_1, \theta_1(\Gamma) \vdash d_2 : \tau_2, \theta_2 \qquad \theta_3 = \mathcal{U}(\theta_2(\tau_1), \tau_2 \to \alpha) \qquad \theta' = \theta_2 \sqcup \theta_3}{\theta, \Gamma \vdash d_1 d_2 : \theta'(\alpha), \theta'} \tag{5}$$

In Rule 5, α is a freshly chosen typing variable that appears nowhere in d or θ_2. The function $\mathcal{U} : (\tau \times \tau) \rightharpoonup \theta$ performs *type unification*, and is defined as follows:

Definition 2. The *unification* θ of two types τ_1 and τ_2 is an instantiation of free typing variables in τ_1 and τ_2 such that $\theta(\tau_1) = \theta(\tau_2)$. (Note that the range of θ may contain free type variables.)

Here is a recursive algorithm that performs type unification:

$$\mathcal{U}(\alpha, \alpha) = \bot \tag{6}$$

$$\mathcal{U}(int, int) = \bot \tag{7}$$

$$\mathcal{U}(\alpha, \tau) = \{(\alpha, \tau)\} \text{ if } \alpha \text{ is not free in } \tau \tag{8}$$

$$\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \mathcal{U}(\tau_1, \tau_1') \sqcup \mathcal{U}(\tau_2, \tau_2') \tag{9}$$

$$\mathcal{U} \text{ is undefined otherwise (type-inference rejects)} \tag{10}$$

Note that not all pairs of types can be unified since cases 9 or 10 may fail. (Case 9 may fail because not every pair of functions has an upper bound.) When unification fails, Rule 5 is unprovable, and therefore the type-inferrer rejects the program as having no shallow type.

References

[1] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 176–185, 1994.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 207–212, 1982.