CSCI 740 - Programming Language Theory

Lecture 13
Type Inference & Polymorphism
Instructor: Hossein Hojjat

September 29, 2017

Motivating Example

- Is $(\lambda f : \mathsf{Int} \to \mathsf{Int}.\ f\ 3)(\lambda x : \mathsf{Int}.\ x + 5)$ well-typed in F_1?

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ \ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Motivating Example

- Is $(\lambda f : \mathsf{Int} \to \mathsf{Int}.\ f\ 3)(\lambda x : \mathsf{Int}.\ x + 5)$ well-typed in F_1?
- We did not need to look at the type labels to determine well-typedness
  - We could have figured the actual types without the labels
  - We could have written the derivation without the labels

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ \ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Type Inference

- Consider the simply-typed λ-calculus with integers

$$
\begin{aligned}
e &::= x \mid \lambda x : \tau.\ e \mid e_1\ \ e_2 \mid n \mid e_1 + e_2 \\
\tau &::= \mathsf{Int} \mid \tau_1 \to \tau_2
\end{aligned}
$$

Type inference

- Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?

Type Inference

- **Problem:** Consider the typing rule for function abstraction

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.\ e) : \tau \to \tau'}$$

- Without type annotations, where do we get τ?
- Use type variables to stand for as-yet-unknown types

$$\tau \ ::= \ \alpha \ \mid \ \mathsf{Int} \ \mid \ \tau \to \tau$$

- Generate equality constraints $\tau = \tau$ among the types and type variables
- Solve the constraints to compute a typing

Type Inference Rules

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \alpha \vdash e : \tau' \qquad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : \alpha \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \boxed{\tau_1 = \tau_2 \to \alpha} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1 \; e_2 : \alpha}$$

Generated Constraint

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \boxed{\tau_1 = \mathsf{Int} \wedge \tau_2 = \mathsf{Int}}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Example

$$\dfrac{\dfrac{\Gamma, x : \alpha_2 \vdash x : \alpha_2 \qquad \Gamma, x : \alpha_2 \vdash 1 : \texttt{Int} \qquad \boxed{\alpha_2 = \texttt{Int}}}{\Gamma, x : \alpha_2 \vdash x + 1 : \texttt{Int}}}{\Gamma \vdash (\lambda x. x + 1) : \alpha_2 \to \texttt{Int}} \qquad \Gamma \vdash 2 : \texttt{Int} \qquad \boxed{\alpha_2 \to \texttt{Int} = \texttt{Int} \to \alpha_1}$$
$$\overline{\qquad\qquad\qquad\qquad \Gamma \vdash (\lambda x. x + 1)\ 2 : \alpha_1 \qquad\qquad\qquad\qquad}$$

- We collect all constraints appearing in the derivation into some set C to be solved
- Here, C contains $\alpha_2 \to \text{Int} = \text{Int} \to \alpha_1$ and $\alpha_2 = \text{Int}$
- Solution: $\alpha_1 = \text{Int} = \alpha_2$
- Thus this program is typeable:
  we can derive a typing by replacing α_1 and α_2 by Int in the proof

Solving Equality Constraints

We can solve the equality constraints using the following rewrite rules

- $C \cup \{\texttt{Int} = \texttt{Int}\} \;\Rightarrow\; C$
- $C \cup \{\alpha = \tau\} \;\Rightarrow\; C[\alpha \mapsto \tau]$ provided $\alpha \notin \mathsf{FV}(\tau)$
- $C \cup \{\tau = \alpha\} \;\Rightarrow\; C[\alpha \mapsto \tau]$ provided $\alpha \notin \mathsf{FV}(\tau)$
- $C \cup \{\alpha_1 \to \alpha_2 = \alpha_1' \to \alpha_2'\} \;\Rightarrow\; C \cup \{\alpha_1 = \alpha_1'\} \cup \{\alpha_2 = \alpha_2'\}$
- $C \cup \{\texttt{Int} = \alpha_1 \to \alpha_2\} \;\Rightarrow\;$ unsatisfiable
- $C \cup \{\alpha_1 \to \alpha_2 = \texttt{Int}\} \;\Rightarrow\;$ unsatisfiable

The condition $\alpha \notin \mathsf{FV}(\tau)$ prevents inferring recursive types

Termination

- We can prove that the constraint solving algorithm terminates
- For each rewriting rule, we either
  - Reduce the size of the constraint set
  - Reduce the number of “arrow” constructors in the constraint set
- As a result, the constraint always gets “smaller” and eventually becomes empty
- A similar argument is made for strong normalization in the simply-typed lambda calculus

Exercise

- Is $(\lambda f.\ f\ 3)(\lambda x.\ x + 5)$ well-typed in F_1?

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \alpha \vdash e : \tau' \qquad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : \alpha \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \boxed{\tau_1 = \tau_2 \to \alpha} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\ \ e_2 : \alpha}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \boxed{\tau_1 = \mathsf{Int} \wedge \tau_2 = \mathsf{Int}}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Polymorphism

Motivating Example

- A type system restricts the class of programs that are considered "legal"
- An expression in the untyped λ-calculus may be reducible to a value but may not be typeable in a particular type system

$$\texttt{let} \quad id = \lambda x.x \; \texttt{in}$$

$$\Big(\cdots (id \;\; \mathsf{true}) \cdots (id \;\; 1) \cdots \Big)$$

- This expression is not typeable in the simple type system we have discussed so far

Polymorphism

- Observation: $\lambda x.x$ returns its argument exactly and places no constraints on the type of x
- The identity function works for any argument type
- We can express this with universal quantification:

$$\lambda x.x : \forall \alpha.\ \alpha \to \alpha$$

- For any type α, the identity function has type $\alpha \to \alpha$
- This is also known as parametric polymorphism

Polymorphism

- You have seen this before

```
public interface List<E>{
  void add(E x);
    E get();
}
    ...

List<String> ls = ...
ls.add("Hello");
String hello = ls.get(0);
```

- How do we formalize this concept?

System F: annotated polymorphism

- Let's extend our system as follows:

$$\tau ::= \alpha \mid \mathsf{Int} \mid \tau \to \tau \mid \forall\alpha.\tau$$
$$e ::= n \mid x \mid \lambda x : \tau.e \mid e \;\; e \mid \Lambda\alpha.e \mid e[\tau]$$

- We add polymorphic types, add explicit type abstraction (generalization)
- Annotated code locations at which a value of polymorphic type is created
- We add type application (instantiation)

Defining Polymorphic Functions

- Polymorphic functions map types to terms
- Normal functions map terms to terms
- Examples:
- $\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha$
- $\Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.x : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$
- $\Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.y : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta$

Instantiation

- When we use a parametric polymorphic type, we apply (or instantiate) it with a particular type
- In System F this is done by hand:
  - $(\Lambda\alpha.\lambda x : \alpha.x)[\tau_1] : \tau_1 \to \tau_1$
  - $(\Lambda\alpha.\lambda x : \alpha.x)[\tau_2] : \tau_2 \to \tau_2$
- This is where the term parametric comes from
- The type $\forall\alpha.\alpha \to \alpha$ is a "function" in the domain of types, and it is passed a parameter at instantiation time

Type Rules

Type Abstraction (Generalization)

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e \ : \ \forall\alpha.\tau}$$

Type Application (Instantiation)

$$\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e[\tau'] : \tau[\alpha \mapsto \tau']}$$

Free Variables of a Type

- Need to perform substitutions on quantified types
- Just like λ-calculus, we need to worry about free variables and capture-free substitution
- Define the free variables of a type
  - $\mathsf{FV}(\alpha) = \{\alpha\}$
  - $\mathsf{FV}(c) = \emptyset$
  - $\mathsf{FV}(\tau \to \tau') = \mathsf{FV}(\tau) \cup \mathsf{FV}(\tau')$
  - $\mathsf{FV}(\forall \alpha . \tau) = \mathsf{FV}(\tau) - \{\alpha\}$

Substitution

Define $\tau[\alpha \to u]$ as

- $\alpha[\alpha \mapsto u] = u$
- $\beta[\alpha \mapsto u] = \beta$ where $\beta \neq \alpha$
- $(\tau \to \tau')[\alpha \mapsto u] = \tau[\alpha \mapsto u] \to \tau'[\alpha \mapsto u]$
- $(\forall\beta.\tau)[\alpha \mapsto u] = \forall\beta.(\tau[\alpha \mapsto u])$ where $\beta \neq \alpha$ and $\beta \notin \mathsf{FV}(u)$

Define $e[\alpha \mapsto u]$ as

- $(\lambda x : \tau.e)[\alpha \mapsto u] = \lambda x : \tau[\alpha \mapsto u].e[\alpha \mapsto u]$
- $(\Lambda\beta.e)[\alpha \mapsto u] = \Lambda\beta.e[\alpha \mapsto u]$ where $\beta \neq \alpha$ and $\beta \notin \mathsf{FV}(u)$
- $(e_1\ e_2)[\alpha \mapsto u] = e_1[\alpha \mapsto u]\ e_2[\alpha \mapsto u]$
- $x[\alpha \mapsto u] = x$
- $n[\alpha \mapsto u] = n$

Inference for Polymorphism

- We would like to have the power of System F, and the ease of use of type inference
- Given an untyped λ-calculus expression, can we discover the annotations necessary for typing the term in System F?
  (if such a typing is possible)
- No: This problem has been shown to be undecidable
- Can we at least perform some type inference for parametric polymorphism?
- Yes! (Hindley and Milner Type System)