Introduction

- So far when we studied typing, we always assumed that the programmer annotated some types.
- Example: We gave types to let bindings and lambda variables in class.
- But annotating types can be cumbersome!
- Anyone who has ever written C++ code can really empathize: `vector<Map<int, string>> ::const_iterator it...`

Type Inference

- Goal of type inference: Automatically deduce the most general type for each expression.
- Two key points:
  1. Automatically inferring types: This means the programmer has to write no types, but still gets all the benefit from static typing.
  2. Inferring the most general type: This means we want to infer polymorphic types whenever possible.

Type System

Here is the type system we used in the lambda language:

\[
\begin{align*}
\Gamma & \vdash i : \text{Int} \\
\Gamma & \vdash s : \text{String} \\
\Gamma & \vdash \text{id} : \Gamma(\text{id}) \\
\Gamma & \vdash S_1 : \tau_1 \\
\Gamma & \vdash S_2 : \tau_2 \\
\Gamma, x : \tau_1 \vdash S_1 : \tau_2 \\
\Gamma, x : \tau_1, y : \tau_2 \vdash (S_1 S_2) : \tau_2 \\
\Gamma & \vdash \text{let id} : \tau = S_1 \text{ in } S_2 : \tau_3 \\
\Gamma & \vdash \text{let id} : \tau = S_1 \text{ in } S_2 : \tau_3 \\
\end{align*}
\]

Type Inference Example 1

- But, do we actually need these type annotations to infer the type of programs?
- Consider the following example:
  \[
  \text{let } f1 = \lambda x. x + 2 \text{ in ..}
  \]
- Here, we know that function \( f1 \) adds two to its argument.
- We also know that plus is only defined on integers.
- Therefore, the type of \( f1 \) must be \( \text{Int} \rightarrow \text{Int} \).

Type Inference Example 2

- Consider the following example:
  \[
  \text{let } f2 = \lambda x. \lambda y. x + y \text{ in ..}
  \]
- Here, we know that function \( f2 \) has two (curried) arguments, \( x \) and \( y \).
- We also know that plus is only defined on integers.
- Therefore, the type of \( f2 \) must be \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \).
Type Inference Example 3

- Consider the following example:
  \[ \text{let } f_2 = \lambda x . \lambda y . x + 1 \text{ in } .. \]
- Here, we know that function \( f_2 \) has two (curried) arguments, \( x \) and \( y \)
- We also know that \( \text{plus} \) is only defined on integers
- But \( f_2 \) will work for any type of \( y \)
- Therefore, the type of \( f_2 \) must be \( \forall \alpha. \text{Int} \to \alpha \to \text{Int} \)

Type Inference Example 4

- Now, consider the following example:
  \[ \text{let } f_2 = \lambda g . (g \ 0) \text{ in } .. \]
- Here, we know that function \( f_2 \) takes a function as argument since it is applied to \( 0 \)
- We also know that the function \( g \) is applied to \( \text{integer} \)
- Therefore, the type of \( g \) must be \( \forall \alpha. \text{Int} \to \alpha \)
- This means that the type of \( f_2 \) is \( \forall \alpha . (\text{Int} \to \alpha) \to \alpha \)

Type Inference Overview

- Goal of the rest of this lecture: Develop an algorithm that can compute the most general type for any expression without any type annotations
- For this, let us look at the type derivation for the following simple function:
  \[ \lambda x : \text{Int}. x + 2 \]
- Here is the type derivation tree for this expression:

\[
\begin{align*}
\Gamma(x) &= \text{Int} \\
\Gamma[x \leftarrow \text{Int}] + x : \text{Int} &\quad \Gamma[x \leftarrow \text{Int}] + 2 : \text{Int} \\
\Gamma[x \leftarrow \text{Int}] + x + 2 : \text{Int} &\quad \Gamma + \lambda x : \text{Int}. x + 2 : \text{Int} \\
\end{align*}
\]

Type Variables

- Big Idea: Replace the concrete type \( \text{Int} \) annotated with a type variable and collect all constraints on this type variable.
- Specifically, pretend that the type of the argument is just some type variable called \( a \)
- And for all rules that have preconditions on \( a \), write these preconditions as constraints

Type Variables Cont.

- Here is the type derivation tree for this expression using type variable \( a \):

\[
\begin{align*}
\Gamma(x) &= a \\
\Gamma[x \leftarrow a] + x : a &\quad a = \text{Int} \\
\Gamma[x \leftarrow a] + 2 : \text{Int} &\quad \Gamma + \lambda x : a. x + 2 : a \to \text{Int} \\
\end{align*}
\]
- Observe that we have one additional precondition on the plus rule: The type variable \( a \) must be equal to \( \text{Int} \) for this rule to apply.
- We now obtain the type: \( a \to \text{Int} \) and the constraint \( a = \text{Int} \)
- Final type: \( \text{Int} \to \text{Int} \)
Generalizing this Example

- This strategy generalizes!
- We will introduce type variables for every type annotation
- We will collect constraints on type variables during type checking
- We will end up with a type containing type variables
- We will solve this type with respect to the collected constraints

Generalizing our typing rules

- Let’s move on to the typing rule for lambda:
  \[ \Gamma \vdash S_1 : \tau_1 \]
  \[ \Gamma \vdash S_2 : \tau_2 \]
  \[ \tau_1 = \text{String}, \tau_2 = \text{String} \]
  \[ \Gamma \vdash S_1 :: S_2 : \text{String} \]
- The lines marked in red are constraints.
- Again, this rule now succeeds as long as \( S_1 \) and \( S_2 \) evaluate to any type, we simply collect constraints on the types \( \tau_1 \) and \( \tau_2 \) to be processed later

The Lambda Case

- Let’s move on to the typing rule for lambda:
  \[ \Gamma \vdash x \leftarrow a \vdash S_1 : \tau \] (a fresh)
  \[ \Gamma \vdash \lambda x. S_1 : a \rightarrow \tau \]
- Here, again we introduce a fresh type variable to capture the (unknown) type of \( x \).
- We also use this type variable in the return type

Application

- Now the only rule missing so far is application:
  \[ \Gamma \vdash S_1 : \tau_1 \]
  \[ \Gamma \vdash S_2 : \tau_2 \]
  \[ \tau_1 = \tau_2 \rightarrow a \] (a fresh)
  \[ \Gamma \vdash (S_1 S_2) : a \]
- Here, we again introduce a fresh type variable \( a \)
- In this rule, this type variable encodes the return type of the application
Example 1

- Let’s use these new rules to derive the typing judgment and constraints on some examples:
  - \( \lambda x.x + 2 \)
- Type derivation:
  
  \[
  \begin{align*}
  \Gamma[x \leftarrow a_1] & \vdash x : a_1 \\
  \Gamma[x \leftarrow a_1] & \vdash 2 : \text{Int} \\
  \Gamma & \vdash x + 2 : \text{Int} \\
  \Gamma & \vdash \lambda x. x + 2 : a_1 \rightarrow \text{Int}
  \end{align*}
  \]
- Final Type: \( a_1 \rightarrow \text{Int} \) under constraints \( a_1 = \text{Int}, \text{Int} = \text{Int} \)

Example 2

- What about the following recursive function? (This function does not terminate, but this is unimportant for this example)
  - let \( f = \lambda x. (f x) \) in \( f \)
- Type derivation:
  
  \[
  \begin{align*}
  \Gamma[f \leftarrow a_1] & \vdash f : a_1 \\
  \Gamma[f \leftarrow a_1] & \vdash [f : a_2] : a_2 \\
  a_2 & \vdash a_1 \rightarrow a_3 \\
  \Gamma[f \leftarrow a_1] & \vdash \lambda x. (f x) : a_1 \\
  \Gamma[f \leftarrow a_1] & \vdash \text{let } f = \lambda x. (f x) \text{ in } f : a_1
  \end{align*}
  \]
- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \)

Example 3

- Let’s look at the following expression
  - "duck" + 7
- Type derivation:
  
  \[
  \begin{align*}
  \Gamma & \vdash "\text{duck}" : \text{String} \\
  \Gamma & \vdash 7 : \text{Int} \\
  \text{String} & = \text{Int}, \text{Int} = \text{Int} \\
  \Gamma & \vdash "\text{duck"} + 7 : \text{Int}
  \end{align*}
  \]
- We derived type \( \text{Int} \) under constraints \( \text{String} = \text{Int}, \text{Int} = \text{Int} \)
- These constraints are unsatisfiable!
- This means that the expression cannot be typed

Type Inference Structure

- Observe that we have split the problem of type inference into two separate problems:
  1. Constraint Inference: In this step, we apply the typing rules to find the type (potentially in terms of type variables) and type constraints
  2. Constraint Solving: In this step, we solve the constraints. Either we find a (potentially polymorphic) final type or the constraints are unsatisfiable, in which case the program cannot be typed
- Observe that step 1 can never get stuck! We now reject all programs that cannot be types in step 2.
Constraint Solving

- So far, we have only informally sketched what we mean by solving type constraints
- Convention: I will write constraints as a list with the type of the program at the bottom
- Example: Consider again the expression \( f = \lambda x.(f\ x) \) in \( f \)

Here, the type of \( f \) written as list of constraints is:

\[
\begin{align*}
a_1 &= a_2 \rightarrow a_3 \\
a_1 &= a_1
\end{align*}
\]

Constraint Solving Cont.

- Then, drop all trivial constraints:

\[
a_2 \rightarrow a_3
\]

with substitution \( \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

- Repeat until we find a contradiction (\( \text{Int} = \text{String} \)) or there are no equalities left.

- In this case, we have found the most general solution.

Constraint Solving

- Definition: A solution to a system of type constraints is a substitution \( \sigma \) mapping type variables to types such that all type constraints are satisfied

- We discovered one solution, \( a_1 \rightarrow a_2 \) for the system

\[
\begin{align*}
a_1 &= a_2 \rightarrow a_3 \\
a_1 &= a_1
\end{align*}
\]

- Substitution: \( \sigma = \{ a_1 \leftarrow a_1, a_2 \leftarrow a_2, a_3 \leftarrow (a_1 \rightarrow a_2) \} \)

- But the following is also a solution: \( \text{Int} \rightarrow \text{Int} \)

- Substitution: \( \sigma = \{ a_1 \leftarrow \text{Int}, a_2 \leftarrow \text{Int}, a_3 \leftarrow (\text{Int} \rightarrow \text{Int}) \} \)

Constraint Solving Cont.

- First Idea: We choose a variable on left-hand side and replace all occurrences of this variable with its right-hand side. In other words, we add the substitution \( x \leftarrow y \) for the equality \( x = y \)

- Consider again the constraint system:

\[
\begin{align*}
a_1 &= a_2 \rightarrow a_3 \\
a_1 &= a_1
\end{align*}
\]

- Here, we pick \( a_1 \). It’s right-hand side is \( a_2 \rightarrow a_3 \). If we replace all occurrences of \( a_1 \), we get:

\[
\begin{align*}
a_2 &= a_1 \\
a_2 &= a_2 \rightarrow a_3 \\
a_2 &= a_3
\end{align*}
\]

and the substitution \( \sigma = \{ a_1 \leftarrow (a_2 \rightarrow a_3), a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

Constraint Solving Example

- Another example:

\[
\begin{align*}
a_1 &= a_2 \rightarrow \text{Int} \\
a_1 &= \text{String} \rightarrow a_3
\end{align*}
\]

- Let’s pick \( a_1 \):

\[
\begin{align*}
a_2 &= \text{Int} \rightarrow a_2 \rightarrow \text{Int} \\
a_2 &= \text{Int} \rightarrow \text{String} \rightarrow a_3
\end{align*}
\]

with \( \sigma = \{ a_1 \leftarrow a_2 \rightarrow \text{Int}, a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

- Remove redundant constraints:

\[
\begin{align*}
a_2 &= \text{Int} \rightarrow \text{String} \rightarrow a_3 \\
a_2 &= \text{Int} \rightarrow a_3 \\
&\text{with } \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \}
\end{align*}
\]

- But now we are stuck, even though the final substitution is

\[
\sigma = \{ a_2 \leftarrow \text{String}, a_1 \leftarrow \text{Int}, \ldots \}
\]
**Constraint Solving Example**

- **Solution:** Add one more rule:
  - Rule: If $X \rightarrow Y = W \rightarrow Z$, then add substitution $X = W$ and $Y = Z$

- **Back to the example:**
  
  $a_2 \rightarrow \text{Int} = \text{String} \rightarrow a_3$
  
  with $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$

- **Add $a_2 \leftarrow \text{Int}$ and $a_3 \leftarrow \text{String}$**

- **New constraint system:**
  
  $\text{String} \rightarrow \text{Int} = \text{String} \rightarrow \text{Int}$
  
  with $\sigma = \{a_2 \leftarrow \text{String}, a_3 \leftarrow \text{Int}\}$

**Simple Unification Algorithm**

- From constraints, pick one equality $a_x = e$ and apply substitution $a_x \leftarrow e$

- If such an equality does not exist, pick an equality of the form $X \rightarrow Y = W \rightarrow Z$ and apply substitutions $X \leftarrow W, Y \leftarrow Z$

- Repeat until we either derive a contradiction or there are not equalities left. This is a most general unifier.

**Conclusion**

- We have seen how we can use our typing rules to generate type constraints.

- We looked at a simple algorithm to solve these constraints.

- But this algorithm is not very efficient.

- Next time: How to perform unification efficiently and type inference in L

- Have a nice spring break!