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:

  \[
  \begin{array}{llll}
  \text{integer} & i & \Gamma \vdash i : \text{Int} \\
  \text{string} & s & \Gamma \vdash s : \text{String} \\
  \text{identifier} & \text{id} & \Gamma \vdash \text{id} : \Gamma(\text{id}) \\
  \text{variable} & x & \Gamma \vdash x : \tau_1 \\
  \text{function} & \lambda x : \tau_1 . S_1 & \Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2 \\
  \text{expression} & (S_1 S_2) & \Gamma \vdash (S_1 S_2) : \tau_2 \\
  \text{expression} & \text{let id} : \tau \leftarrow \tau_1 \text{ in } S_1 & \Gamma \vdash \text{let id} : \tau \leftarrow \tau_1 \text{ in } S_1 : \tau_2 \\
  \text{expression} & S_1 + S_2 & \Gamma \vdash S_1 + S_2 : \tau_1 \rightarrow \tau_2 \\
  \text{expression} & S_1 :: S_2 & \Gamma \vdash S_1 :: S_2 : \tau_1 \\
  \text{expression} & S_1 : \tau & \Gamma \vdash S_1 : \tau_1 \\
  \text{expression} & S_1 : \tau_1 \rightarrow \tau_2 & \Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \\
  \end{array}
  \]

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:
  \[
  \text{let } f2 = \lambda x . x \lambda y . x + y \text{ in }..
  \]
- 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:
  
  ```
  let f2 = lambda x.lambda y.x+y in ..
  ```

- Here, we know that function f2 has two (curried) arguments, x and y

- We also know that plus is only defined on integers

- But f2 will work for any type of y

- Therefore, the type of f2 must be \( \forall a.\text{Int} \rightarrow a \rightarrow \text{Int} \)

Type Inference Example 4

- Now, consider the following example:
  
  ```
  let f2 = lambda g.(g 0) in ..
  ```

- Here, we know that function f2 takes a function as argument since it is applied to 0.

- We also know that the function g is applied to integer

- Therefore, the type of g must be \( \forall a.\text{Int} \rightarrow a \)

- This means that the type of f2 is \( \forall a.(\text{Int} \rightarrow a) \rightarrow a \)

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}] & \vdash x : \text{Int} \\
  \Gamma[x \leftarrow \text{Int}] & \vdash 2 : \text{Int} \\
  \Gamma & \vdash \lambda x.\text{Int}.x + 2 : \text{Int} \rightarrow \text{Int}
  \end{align*}
  ```

Type Variables

- Big Idea: Replace the concrete type 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] & \vdash x : a \\
  \Gamma[x \leftarrow a] & \vdash 2 : \text{Int} \\
  \Gamma & \vdash \lambda x.a.x + 2 : a \rightarrow \text{Int}
  \end{align*}
  ```

- Observe that we have one additional precondition on the plus rule: The type variable \( a \) must be equal to Int for this rule to apply.

- We now obtain the type: \( a \rightarrow \text{Int} \) and the constraint \( a = \text{Int} \)

- Final type: \( \text{Int} \rightarrow \text{Int} \)

Type Variables in Typing Rules

- In this example, we dealt with not knowing the type of \( x \) in the following way:
  
  - We introduced a type variable \( a \) for the type of \( x \)
  
  - Every time a rule uses the type of \( x \), we use \( a \)

  - Since the plus rule has the precondition that both operands must be of type \( \text{Int} \), we introduce a constraint \( a = \text{Int} \)

  - After we typed the expression, we had a the type \( a \rightarrow \text{Int} \) and the constraint \( a = \text{Int} \)

  - Solving the type with respect to the collected constraint yields: \( \text{Int} \rightarrow \text{Int} \)
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

- The base cases stay unchanged:
  \[
  \begin{align*}
  \Gamma \vdash i : \text{Int} & \quad \Gamma \vdash s : \text{String} & \quad \Gamma \vdash id : \Gamma(id)
  \end{align*}
  \]
- When type checking plus, we now collect constraints on the operands:
  \[
  \begin{align*}
  \Gamma \vdash S_1 : \tau_1 & \quad \Gamma \vdash S_2 : \tau_2 \\
  \tau_1 = \text{Int}, \tau_2 = \text{Int} & \\
  \Gamma \vdash S_1 + S_2 : \text{Int}
  \end{align*}
  \]
- The lines marked in red are constraints.
- Specifically, 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:
  \[
  \begin{align*}
  \Gamma \vdash S_1 : \tau_1 \\
  \Gamma \vdash S_2 : \tau_2 \\
  \tau_1 = \text{String}, \tau_2 = \text{String} & \\
  \Gamma \vdash S_1 :: S_2 : \text{String}
  \end{align*}
  \]
- Here, we again 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:
  \[
  \begin{align*}
  \Gamma \vdash S_1 : \tau_1 \\
  \Gamma \vdash S_2 : \tau_2 \\
  \tau_1 = \tau_2 \rightarrow a & \quad (a \text{ fresh}) & \\
  \Gamma \vdash S_1 S_2 : a
  \end{align*}
  \]
- Here, we again introduce a fresh type variable \(a\).
- In this rule, this type variable encodes the return type of the application.

Generalizing our typing rules

- Let’s move on to the typing rule for concatenation:
  \[
  \begin{align*}
  \Gamma \vdash S_1 : \tau_1 & \quad \Gamma \vdash S_2 : \tau_2 \\
  \tau_1 = \text{String}, \tau_2 = \text{String} & \\
  \Gamma \vdash S_1 :: S_2 : \text{String}
  \end{align*}
  \]
- Here, all we do is introduce a fresh type variable to capture the (unknown) type of \(id\).
- Observe that this case only introduces a type variable, but does not add any constraints.

The Let Case

- Let’s move on to the typing rule for let:
  \[
  \begin{align*}
  \Gamma[id \leftarrow a] \vdash S_1 : a & \quad (a \text{ fresh}) & \\
  \Gamma[id \leftarrow a] \vdash S_2 : \tau & \\
  \Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau
  \end{align*}
  \]
- Observe that this case only introduces a type variable, but does not add any constraints.
Example 1

- Let’s use these new rules to derive the typing judgment and constraints on some examples:

  \[
  \text{lambda } x. x + 2
  \]

- Type derivation:

  \[
  \begin{align*}
  \Gamma(x) & \vdash x : a_1 \quad \Gamma(x) \vdash 2 : Int \\
  \Gamma(x) & \vdash x + 2 : Int \\
  \Gamma & \vdash \lambda x. x + 2 : a_1 \rightarrow Int \\
  \end{align*}
  \]

  Final Type: \( a_1 \rightarrow Int \) under constraints \( a_1 = Int, Int = Int \)

Example 2

- What about the following recursive function? (This function does not terminate, but this is unimportant for this example)

  \[
  \text{let } f = \text{lambda } x. (f x) \text{ in } f
  \]

- Type derivation:

  \[
  \begin{align*}
  \Gamma & \vdash f \rightarrow a_1 \rightarrow a_2 \\
  \Gamma & \vdash a_2 \rightarrow a_3 \\
  \Gamma & \vdash a_1 \rightarrow a_2 \\
  \Gamma & \vdash \lambda x. f x : a_1 \\
  \Gamma & \vdash \text{let } f = \lambda x. (f x) \text{ in } f : a_1
  \end{align*}
  \]

  Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \)

Example 3

- Let’s look at the following expression

  \[
  \text{"duck" + 7}
  \]

- Type derivation:

  \[
  \begin{align*}
  \Gamma & \vdash \text{"duck"} : String \\
  \Gamma & \vdash 7 : Int \\
  \Gamma & \vdash \text{Int, Int = Int} \\
  \end{align*}
  \]

  We derived type \( Int \) under constraints \( \text{String = Int, Int = Int} \)

  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 let \( f = \lambda x. (f \ f) \) in \( f \)
  
  Here, the type of \( f \) written as list of constraints is:
  \[
  a_1 = a_2 \rightarrow a_3
  \]

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:
  \[
  a_1 = a_2 \rightarrow a_3
  \]
  \[
  a_2 \rightarrow a_3
  \]

  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:
  \[
  a_2 \rightarrow a_3 = a_2 \rightarrow a_3
  \]
  \[
  a_2 = a_3
  \]
  and the substitution \( \sigma = \{ a_2 \leftarrow (a_2 \rightarrow a_3), a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

Constraint Solving Example

- Another example:
  \[
  a_1 = a_2 \rightarrow Int
  \]
  \[
  a_1 = String \rightarrow a_3
  \]

  Let’s pick \( a_1 \):
  \[
  a_2 \rightarrow Int = a_2 \rightarrow Int
  \]
  \[
  a_2 \rightarrow Int = String \rightarrow a_3
  \]
  with \( \sigma = \{ a_1 \leftarrow a_2 \rightarrow Int, a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

  Remove redundant constraints:
  \[
  a_2 \rightarrow Int = String \rightarrow a_3
  \]
  \[
  a_2 = String \rightarrow a_3
  \]
  with \( \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

  But now we are stuck, even though the final substitution is
  \[
  \sigma = \{ a_2 \leftarrow String, a_3 \leftarrow Int, \ldots \}
  \]
Constraint Solving Example

- Solution: Add one more rule:
  - Rule: If \( X \to Y = W \to Z \), then add substitution \( X = W \) and \( Y = Z \)
- Back to the example:
  - \( a_2 \to \text{Int} = \text{String} \to a_3 \)

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:
  - \( \text{String} \to \text{Int} = \text{String} \to \text{Int} \)
  - 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 \( X \to Y = W \to Z \) 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