CS345H: Programming Languages

Lecture 12: Type Inference

Thomas Dillig

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:

$$\frac{\text{integer } i}{\Gamma \vdash i : Int} \quad \frac{\text{string } s}{\Gamma \vdash s : String} \quad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : Int \\ \Gamma \vdash S_2 : Int\end{array}}{\Gamma \vdash S_1 + S_2 : Int} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : String \\ \Gamma \vdash S_2 : String\end{array}}{\Gamma \vdash S_1 :: S_2 : String}$$

$$\frac{\begin{array}{l}\Gamma \vdash S_1 : \tau_1 \\ \tau = \tau_1 \\ \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3\end{array}}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

$$\frac{\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2} \quad \frac{\begin{array}{l}\Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash S_2 : \tau_1\end{array}}{\Gamma \vdash (S_1\ S_2) : \tau_2}$$

Type Inference Example 1

- But, do we actually need these type annotations to infer the type of programs?
- Consider the following example: `let f1 = lambda x.x+2 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 $Int \rightarrow Int$

Type Inference Example 2

- 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
- Therefore, the type of `f2` must be $Int \rightarrow Int \rightarrow Int$

Type Inference Example 3

- Consider the following example:
  `let f2 = lambda x.lambda y.x+1 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 $\forall\alpha.Int \to \alpha \to Int$

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 in integer
- Therefore, the type of `g` must be $\forall\alpha.Int \to \alpha$
- This means that the type of `f2` is $\forall\alpha.(Int \to \alpha) \to \alpha$

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:Int.x+2`
- Here is the type derivation tree for this expression:

$$\dfrac{\dfrac{\dfrac{identifer\ x}{\Gamma(x) = Int}}{\Gamma[x \leftarrow Int] \vdash x : Int} \quad \dfrac{integer\ 2}{\Gamma[x \leftarrow Int] \vdash 2 : Int}}{\dfrac{\Gamma[x \leftarrow Int] \vdash x + 2 : Int}{\Gamma \vdash \lambda x{:}Int.x + 2 : Int \to Int}}$$

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:

$$\dfrac{\dfrac{\dfrac{identifer\ x}{\Gamma(x) = a}}{\Gamma[x \leftarrow a] \vdash x : a} \quad a = Int \quad \dfrac{integer\ 2}{\Gamma[x \leftarrow a] \vdash 2 : Int}}{\dfrac{\Gamma[x \leftarrow a] \vdash x + 2 : Int}{\Gamma \vdash \lambda x{:}a.x + 2 : a \to Int}}$$

- 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: $a \to Int$ and the constraint $a = Int$
- Final type: $Int \to Int$

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 x
  - Since the plus rule has the precondition that both operands must be of type Int, we introduced a constraint $a = Int$
  - After we typed the expression, we had a the type $a \to Int$ and the constraint $a = Int$
  - Solving the type with respect to the collected constraint yields: $Int \to Int$

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:

$$\frac{\text{integer } i}{\Gamma \vdash i : Int} \quad \frac{\text{string } s}{\Gamma \vdash s : String} \quad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

- When type checking plus, we now collect constraints on the operands:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau_1 \\ \Gamma \vdash S_2 : \tau_2 \\ \tau_1 = Int, \tau_2 = Int\end{array}}{\Gamma \vdash S_1 + S_2 : Int}$$

- 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 τ_1 and τ_2 to be processed later

Generalizing our typing rules

- Let's move on to the typing rule for concatenation:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau_1 \\ \Gamma \vdash S_2 : \tau_2 \\ \tau_1 = String, \tau_2 = String\end{array}}{\Gamma \vdash S_1 :: S_2 : String}$$

- The lines marked in red are again 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 τ_1 and τ_2 to be processed later

The Let Case

- Let's move on to the typing rule for let:

$$\frac{\begin{array}{c}\Gamma[\text{id} \leftarrow a] \vdash S_1 : a \quad (a \text{ fresh}) \\ \Gamma[\text{id} \leftarrow a] \vdash S_2 : \tau\end{array}}{\Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau}$$

- 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 Lambda Case

- Let's move on to the typing rule for lambda:

$$\frac{\Gamma[x \leftarrow a] \vdash S_1 : \tau \quad (a \text{ fresh})}{\Gamma \vdash \lambda x.S_1 : a \rightarrow \tau}$$

- 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:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau_1 \\ \Gamma \vdash S_2 : \tau_2 \\ \tau_1 = \tau_2 \rightarrow a \quad (a \text{ fresh})\end{array}}{\Gamma \vdash (S_1\ S_2) : a}$$

- 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:

$$\dfrac{\dfrac{\dfrac{\textit{identifer } x}{\Gamma(x) = a_1}}{\Gamma[x \leftarrow a_1] \vdash x : a_1} \quad \dfrac{\textit{integer } 2}{\Gamma[x \leftarrow Int] \vdash 2 : Int} \quad {\color{red} a_1 = Int, Int = Int}}{\dfrac{\Gamma[x \leftarrow a_1] \vdash x + 2 : Int}{\Gamma \vdash \lambda x.x + 2 : a_1 \rightarrow Int}}$$

- Final Type: $a_1 \rightarrow Int$ under constraints $a_1 = Int, Int = Int$

Example 1 Cont

- What does this type mean? $a_1 \rightarrow Int$ under constraints $a_1 = Int, Int = Int$
- We want to solve this type, i.e., substitute everything known from the constraints as much as possible.
- Goal of Solving: Deduce final type with no constraints
- Solving this type yields $Int \rightarrow Int$

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:

$$\dfrac{\dfrac{\dfrac{\Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash f : a_1 \quad \Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash x : a_2 \quad {\color{red} a_1 = a_2 \rightarrow a_3}}{\Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash (f\ x) : a_3}}{\Gamma[f \leftarrow a_1] \vdash \lambda x.(f\ x) : a_1} \quad \Gamma[f \leftarrow a_1] f \vdash : a_1}{\Gamma \vdash \text{ let } f = \lambda x.(f\ x) \text{ in } f : a_1}$$

- Final Type: a_1 under constraint $a_1 = a_2 \rightarrow a_3$

Example 2 Cont

- Recall function: `let f = lambda x.(f x) in f`
- Final Type: a_1 under constraint $a_1 = a_2 \rightarrow a_3$, but what does this final type mean?
- First of all, observe that we can solve this type and these constraints.
- This yields $a_2 \rightarrow a_3$
- Here, since the solution still includes type variables, we found a polymorphic type!
- Here, the type is $\forall \alpha_1.\forall \alpha_2.\alpha_1 \rightarrow \alpha_2$
- We will omit the quantifier from type variables and assume that any type variable is implicitly universally quantified

Example 3

- Let's look at the following expression
  `"duck" + 7`
- Type derivation:

$$\dfrac{\Gamma \vdash \text{"}duck\text{"} : String \quad \Gamma \vdash 7 : Int \quad {\color{red} String = Int, Int = Int}}{\Gamma \vdash \text{"}duck\text{"} + 7 : Int}$$

- We derived type Int under constraints $String = Int, Int = Int$
- 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 x) in f`
- Here, the type of `f` written as list of constraints is:

$$a_1 = a_2 \to a_3$$
$$a_1$$

Constraint Solving

- Definition: A solution to a system of type constraints is a substitution σ mapping type variables to types such that all type constraints are satisfied
- We discovered one solution, $\alpha_1 \to \alpha_2$ for the system

$$a_1 = a_2 \to a_3$$
$$a_1$$

- Substitution: $\sigma = \{a_1 \leftarrow \alpha_1, a_2 \leftarrow \alpha_2, a_3 \leftarrow (\alpha_1 \to \alpha_2)\}$
- But the following is also a solution: $Int \to Int$
- Substitution: $\sigma = \{a_1 \leftarrow Int, a_2 \leftarrow Int, a_3 \leftarrow (Int \to Int)\}$

Constraint Solving

- And $\alpha \to \alpha$ is also a solution for

$$a_1 = a_2 \to a_3$$
$$a_1$$

- Substitution: $\sigma = \{a_1 \leftarrow \alpha, a_2 \leftarrow \alpha, a_3 \leftarrow (\alpha \to \alpha)\}$
- But clearly some solutions are more general than others.
- We want to find the most general solution, also know as the most general unifier.
- This can be done using unification

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:

$$a_1 = a_2 \to a_3$$
$$a_1$$

- Here, we pick a_1. It's right-hand side is $a_2 \to a_3$. If we replace all occurrences of a_1, we get:

$$a_2 \to a_3 = a_2 \to a_3$$
$$a_2 \to a_3$$

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

Constraint Solving Cont.

- Then, drop all trivial constraints:

$$a_2 \to a_3$$

  with substitution $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$
- Repeat until we find a contradiction ($Int = String$) or there are no equalities left.
- In this case, we have found the most general solution.

Constraint Solving Example

- Another example:

$$a_1 = a_2 \to Int$$
$$a_1 = String \to a_3$$

- Let's pick a_1:

$$a_2 \to Int = a_2 \to Int$$
$$a_2 \to Int = String \to a_3$$

  with $\sigma = \{a_1 \leftarrow a_2 \to Int, a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$
- Remove redundant constraints:

$$a_2 \to Int = String \to a_3$$

  with $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$
- But now we are stuck, even though the final substitution is $\sigma = \{a_2 \leftarrow String, a_3 \leftarrow Int, ...\}$

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:

$$a_2 \to Int = String \to a_3$$

with $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$

- Add $s_2 \leftarrow Int$ and $a_3 \leftarrow String$
- New constraint system:

$$String \to Int = String \to Int$$

with $\sigma = \{a_2 \leftarrow String, a_3 \leftarrow Int\}$

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