Lecture 8:Type Inference

The main point of type annotations is to document your code and catch programming errors. However, it can be quite annoying to have to write down every single type in a program. If you've used Java Generics or something similar, you've probably encountered the annoyance of type annotations in real code. The type-checker you wrote earlier also has the same property.

The type inference problem is to take a program with no type annotations, such as this version of factorial:

```
fix f . fun x . if n = 0 then 1 else n * f(n  1)
```

and produce a program with type annotations:

```
fix (f : int →int) . fun (x : int ) . if n = 0 then 1 else n * f(n  1)
```

We know how to type-check the latter.

It is easy to code up with heuristics for inferring types. Howeer, we're going to cover an approach known as the *Hindley-Milner algorithm*, which has two key properties:

- It is *sound*, so it always produces correct types. Therefore, you can trust the output of the algorithm and there is no need to type-check after inference (assuming you implement it correctly!)
- It is *complete*, so it always produces a type, if there is a type to be found. Therefore, if the algorithm fails to find a type, then the program is truly un-typable.

The algorithm has one other important property, which we'll look at later.

An Informal Example Suppose we were given the untyped factorial function and were asked to infer its type in our head. We'll run through a detailed analysis below. For brevity, we'll use the following notation:

- We write $[\![e]\!]$ to mean "the type of the expression e".
- We use Greek letters to denote metavariables that stand for types that are unknown.

We might reason through the types as follows:

	Claim	**Justification**
1	$[\![\texttt{fix f . fun n . } \cdots]\!] = [\![\texttt{fun n . } \cdots]\!]$	`fix` evalutes to the body.
2	$[\![\texttt{f}]\!] = [\![\texttt{fix f . } \cdots]\!]$	`f` is bound to the whole expression during evaluation.
3	$[\![\texttt{fun n . } \cdots]\!] = \alpha \to \beta$	The expression is a function, so it must have an argument and result (α and β).
4	$[\![\texttt{n}]\!] = \alpha$	`n` is the argument of the function that takes argument of type α.
5	$[\![\texttt{if n = 0 then 1 else n * f(n 1)}]\!] = \beta$	The expression is the body of a function that produces a value of type β.
6	$[\![\texttt{n = 0}]\!] = \texttt{bool}$	The expression is used as the condition in an if-expression.
7	$[\![\texttt{n}]\!] = \texttt{int}$	`n` is used in a comparison. (We can only compare integers.)
8	$[\![\texttt{1}]\!] = [\![\texttt{n * f(n 1)}]\!]$	Both branches must have the same type.
9	$[\![\texttt{1}]\!] = [\![\texttt{if n = 0 then 1 else n * f(n 1)}]\!]$	Type of a conditional is the type of either branch.
10	$[\![\texttt{f}]\!] = \gamma \to \delta$	`f` appears in function position in an application.
11	$[\![\gamma]\!] = [\![\texttt{(n 1)}]\!]$	The argument to `f`, which has type γ.
12	$[\![\texttt{n 1}]\!] = \texttt{int}$	Subtraction produces integers.
13	$[\![\texttt{f(n 1)}]\!] = \delta$	The return type of `f` is the type of the application.
14	$[\![\delta = \texttt{int}]\!]$	The expression of type δ is used as an argument of $*$.

At this point, we can solve the constraints to figure out the types of the metavariables α, β, γ, and δ:

- $\alpha = \texttt{int}$ by (4) and (7),
- $\beta = \texttt{int}$ by (5), (9) (given that $[\![1]\!] = \texttt{int}$),

Binary Operators

$op_2 ::= + \mid > \mid \cdots$

Constants

$c ::=$ true	True
$\mid$ false	False
$\mid$ n	Integers

Expressions

$e ::= c$	Constants
$\mid$ x	Identifiers
$\mid$ $op_2(e_1, e_2)$	Bin. Ops.
$\mid$ if e_1 then e_2 else e_3	Conditionals
$\mid$ $e_1\ e_2$	Applications
$\mid$ fun x . e	Functions
$\mid$ fix f . e	Fixpoints

(a) Implicitly-Typed Syntax.

Types

$\tau ::=$ int	Integer type
$\mid$ bool	Boolean type
$\mid$ $\tau_1 \rightarrow \tau_2$	Function types
$\mid$ α	Type metavariables

Binary Operators

$op_2 ::= + \mid > \mid \cdots$

Constants

$c ::=$ true	True
$\mid$ false	False
$\mid$ n	Integers

Expressions

$e ::= c$	Constants
$\mid$ x	Identifiers
$\mid$ $op_2(e_1, e_2)$	Bin. Ops.
$\mid$ if e_1 then e_2 else e_3	Conditionals
$\mid$ $e_1\ e_2$	Applications
$\mid$ fun $(x{:}\tau)$. e	Functions
$\mid$ fix $(f{:}\tau)$. e	Fixpoints

(b) Explicitly-Typed Syntax.

Figure 12.1: Syntaxes for type inference.

- $\gamma =$ int by (11) and (12),
- $\delta =$ int by (14).

Now, we have enough information to reconstruct the type annotations in the program.

1 Introduction

Type inference works with two syntaxes, shown in fig. 12.1. The *implicitly-typed syntax*, on the left-hand side, is the syntax in which the user writes the program. The *explicitly-typed syntax*, on the right-hand side, is the result of type-inference. The difference between these two syntaxes is that the explicitly-typed syntax has type annotations that can be used to type-check the program. In contrast, the implicitly-typed syntax has no type annotations. However, it is important to note that **the implicitly-typed language is typed**. The types aren't written down, but we could still write typing rules for this language.[1] The explicitly-typed syntax just makes types manifest, so that a simple type-checker can type-check the program.

Types in the explicitly-typed language include metavariables, α, β, etc. Metavariables are not types themselves, but are placeholders that stand for types. We need metavariables to describe type inference. However, the result of type inference will produce a program that has no metavariables.

Type inference has several steps:

1. We transform a program in the implicitly-typed syntax to a identical program in the explicitly-typed syntax, using unique metavariables for each type annotation. For example, the program fun x . fun y . x + y would be transformed to fun (x: α) . fun (y: β) . x + y. These metavariables will allow us to refer to identifier's types without getting scope mixed up.

2. We generate a set of constraints by recursively processing the program. Each constraint equates two types to each other. To handle variables and scoping correctly, we use an environment that maps identifiers to metavariables. For example, the expression we have should produce the constraints $\alpha =$ int and $\beta =$ int.

3. We solve the constraints, using a classic algorithm called *unification*. Constraint-solving produces a *substitution* from metavariables to types that have no metavariables within them. For example, solving the constraints above produces the substitution $[\alpha \mapsto$ int $, \beta \mapsto$ int $]$. Several things can go wrong during constraint-solving. For example, we may derive an unsatisfiable constraint, such as int $=$ bool, which indicates that the program has a type error.

[1] We would have to "guess" the type of binding identifiers when building typing derivations.

Constraints
$C \in \tau = \tau$
Constraint Sets
$\mathcal{C} \in \{C_1, \cdots C_n\}$
Type Environments
$\Gamma \in x \rightharpoonup \alpha$

$\boxed{\Gamma \vdash e \Rightarrow \tau \times \mathcal{C}}$

$$\text{T-INT } \Gamma \vdash n \Rightarrow (\mathsf{int}, \emptyset)$$

$$\text{T-BOOL } \Gamma \vdash b \Rightarrow (\mathsf{bool}, \emptyset)$$

$$\text{T-ID } \Gamma \vdash x \to (\Gamma(x), \emptyset)$$

$$\text{T-ADD } \frac{\Gamma \vdash e_1 \Rightarrow (\tau_1, \mathcal{C}_1) \qquad \Gamma \vdash e_2 \Rightarrow (\tau_2, \mathcal{C}_2)}{\Gamma \vdash e_1 + e_2 \Rightarrow (\mathsf{int}, \{\tau_1 = \mathsf{int}, \tau_2 = \mathsf{int}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\text{T-IF } \frac{\Gamma \vdash e_1 \Rightarrow (\tau_1, \mathcal{C}_1) \qquad \Gamma \vdash e_2 \Rightarrow (\tau_2, \mathcal{C}_2) \qquad \Gamma \vdash e_3 \Rightarrow (T_3, \mathcal{C}_3)}{\Gamma \vdash \mathsf{if}\ e_1 \mathsf{then} e_2 \mathsf{else} e_3 : (\tau_2, \{\tau_1 = \mathsf{bool}, \tau_2 = \tau_3\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\text{T-FUN } \frac{\Gamma[x \mapsto \alpha] \vdash e \Rightarrow (\tau, \mathcal{C})}{\Gamma \vdash \mathsf{fun}\ (x{:}\alpha)\ .e \Rightarrow (\alpha \to \tau, \mathcal{C})}$$

$$\text{T-FIX } \frac{\Gamma[x \mapsto \alpha] \vdash e \Rightarrow (\tau, \mathcal{C})}{\Gamma \vdash \mathsf{fix}\ (x{:}\alpha)\ .e \Rightarrow (\alpha, \{\alpha = \tau\}\mathcal{C})}$$

$$\text{T-APP } \frac{\Gamma \vdash e_1 \Rightarrow (\tau, \mathcal{C}_1) \qquad \Gamma \vdash e_2 \Rightarrow (\tau', \mathcal{C}_2) \qquad \alpha \text{ is a fresh metavariable}}{\Gamma \vdash e_1 e_2 \Rightarrow (\alpha, \{\tau' \to \alpha = \tau\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)}$$

(a) Type Checking.

Figure 12.2: Constraint Generation

4. We annotate the program by applying the substitution to the type variables we generated in the first step. In this example, we get the program `fun (x:int ) . fun (y:int ) . x + y`.

5. Finally, we can type-check the resulting program to verify the result of type-inference. However, if the steps above are perfectly correct, there is no need to verify the result.

2 Constraint Generation

Figure 12.2 gives the constraint generation rules for our language. We can read the rule $\Gamma \vdash e \Rightarrow T \times \mathcal{C}$ as *in the environment* Γ*,* e *has type* T *and produces the set of constraints* $\mathcal{C}$. There are two key differences between the type-inference rules and the type-checking rules:

- Unlike the type-checking rules, the type T may not be a complete type and may include metavariables (α, β, etc.).

- Constraint-generation does not catch type-errors. For example, the expression true $+ 10$ will fail to type-check. However, the constraint generation rules will produce the type int and the set of constraints $\{\mathsf{bool} = \mathsf{int}, \mathsf{int} = \mathsf{int}\}$.

However, constraint-generation does catch unbound-identifier errors. If a program has a free variable, then it won't be bound to a metavariable in the environment.

Notice that the types produced by constraint-generation are used to further generate additional constraints. For example, in the expression `fun (x:α) . x + 3`, the bound identifier `x` produces the type α, which is used to produce the constraint $\alpha = \mathsf{int}$ when generating constraints for `x+3`.

3 Solving Type Constraints

After constraint-generation completes, we can solve constraints by *unification*. The UNIFY algorithm, shown in fig. 12.3, takes a single constraint and produces as output a *substitution*, which is a finite map from metavariables to types. A substitution can be applied to a type to replace metavariables with concrete types.

For example, UNIFY$(\alpha \to \mathsf{int} = \mathsf{bool} \to \beta)$ produces the substitution $[\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{bool}]$. If we apply this substitution to $\alpha \to \mathsf{int}$ or to $\mathsf{bool} \to \beta$, we get the type $\mathsf{bool} \to \mathsf{int}$. Notice that we get the same type on either side, which is a key property of the algorithm. However, UNIFY$(\gamma \to \mathsf{int} = \mathsf{bool} \to \gamma)$ should fail, since there is no substitution that can be applied to either side to produce the same type.

UNIFY is a simple recursive algorithm with three cases:

- UNIFY$(\mathsf{int} = \mathsf{int})$ should produce the empty substitution, whereas UNIFY$(\mathsf{int} = \mathsf{bool})$ should fail. In general, unifying two base types should succeed with the empty substitution if the types are the same or fail if the types are different.

Substitutions

$$\pi \in \alpha \to T$$
$$(\pi_1 \cdot \pi_2)(\alpha) = \pi_2(\pi_1(\alpha))$$

Unification

$$\begin{aligned}
\textsc{unify} &\in C \to \pi \\
\textsc{unify}(T = T) &= \cdot \\
\textsc{unify}(\alpha = T) &= [\alpha \mapsto T] \text{ if } \alpha \text{ does not occur in } T \\
\textsc{unify}(T = \alpha) &= [\alpha \mapsto T] \text{ if } \alpha \text{ does not occur in } T \\
\textsc{unify}(S_1 \to S_2 = T_1 \to T_2) &= \textsc{unify}(\pi(S_2), \pi(T_2)) \cdot \pi \\
&\quad \text{where } \pi = \textsc{unify}(S_1, T_1)
\end{aligned}$$

Figure 12.3: Unification

- $\textsc{unify}(\alpha = T)$ and $\textsc{unify}(T = \alpha)$ should produce the substitution $[\alpha \mapsto T]$. However, there is an important caveat discussed below, called the *occurs check*.
- $\textsc{unify}(S_1 \to S_2 = T_1 \to T_2)$ needs to recursively unify the argument and the result types. However, the two substitutions have to be *composed* together.

The Occurs Check Consider the constraint $\alpha = \mathsf{int} \to \alpha$. If unification is done naively, we will produce the substitution $[\alpha \mapsto \mathsf{int} \to \alpha]$. If we apply this substitution to either side, we get the constraint $\mathsf{int} \to \alpha = \mathsf{int} \to (\mathsf{int} \to \alpha)$. We can unify this new constraint to get a larger type on either side. In fact, we can repeatedly apply unification to get larger and larger types. The real problem is that the original constraint is circular and contradictory. There is no substitution π such that $\pi(\alpha) = \pi(\mathsf{int} \to \alpha)$. To avoid this infinite regress, unification needs an *occurs check* to detect cyclic constraints. It is enough to run the occurs check when unifying metavariables with types, as shown in fig. 12.3.

Unifying a set of constraints Given $\textsc{unify}$, it is straightforward to unify a set of constraints. After unifying each constraint in the set, we need to apply the intermediate substitution to the remaining constraints.