Lecture 26: Type Inference and Unification

- Type Inference
- Matrix Example
- Unification
- Unification Algorithm
- Type Inference and Unification

Type Inference

Type inference refers to the process of determining the appropriate types for expressions based on how they are used. For example, in the expression `f 3`, OCaml knows that `f` must be a function, because it is applied to something (not because its name is `f`!) and that it takes an `int` as input. It knows nothing about the output type. Therefore the type inference mechanism of OCaml would assign `f` the type `int -> 'a`.

```
# fun f -> f 3;;
- : (int -> 'a) -> 'a = <fun>
```

There may be many different occurrences of a symbol in an expression, all leading to different typing constraints, and these constraints must have a common solution, otherwise the expression cannot be typed.

```
# fun f -> f (f 3);;
- : (int -> int) -> int = <fun>
# fun f -> f (f "hello");;
- : (string -> string) -> string = <fun>
# fun f -> f (f 3, f 4);;
Error: This expression has type 'a * 'a
       but an expression was expected of type int
```

In the first example, how does it know that the output type of `f` is `int`? Because the input type of `f` is `int`, and the output of `f` is fed into `f` again, so the output type of `f` has to be the same as the input type of `f`.

Matrix Example

Let's illustrate the inference process with a simple example. Suppose we have matrices of different sizes and shapes. Let's write the type of an m x n matrix as m -> n. I can multiply two matrices if and only if the column dimension of the first is equal to the row dimension of the second. This can be represented by a typing rule

```
A : s -> t   B : t -> u
-----------------------
      AB : s -> u
```

which says that if matrix **A** has row and column dimensions **s** and **t**, respectively, and matrix **B** has row and column dimensions **t** and **u**, respectively, then I can multiply them, and the resulting product has row and column dimensions **s** and **u**, respectively. Similarly, there might be other typing rules that say when I can add two matrices or square a matrix and what the result types will be:

```
A : s -> t   B : s -> t       A : s -> s
-----------------------       -----------
     A + B : s -> t           A^2 : s -> s
```

Now given some matrix expression, say $(AB + CD)^2$, for what dimensions of **A**, **B**, **C**, and **D** will this expression make sense? We can start out with abstract types for each subexpression, then add constraints as necessary so that the typing rules will apply. So we start out with types

```
A : s -> t              AB : a -> b
B : u -> v              CD : c -> d
C : w -> x        AB + CD : e -> f
D : y -> z     (AB + CD)^2 : g -> h
```

where the small letters are distinct type variables. Now in order for **AB** to make sense as determined by the typing rule for multiplication, we had better have **t = u**, so that the multiplication can be performed, and **a = s** and **b = v**, so that the type of the result is the same as the type of the product. Similarly, in order for **CD** to make sense, we must have **x = y**, **c = w**, and **d = z**, etc. Collecting all these constraints, we get

```
t = u, a = s, b = v   for AB
x = y, c = w, d = z   for CD
a = c = e, b = d = f  for AB + CD
e = f = g = h         for (AB + CD)^2
```

Solving these constraints, we get three equivalence classes

```
a = b = c = d = e = f = g = h = s = v = w = z
t = u
x = y
```

which we can represent by one of the members from each, say `a`, `t`, and `x`. Thus we have

```
A : a -> t
B : t -> a
C : a -> x
D : x -> a
```

and this is the most general typing for which this expression makes sense. Any values of `a`, `t`, and `x` can be used here as long as `A`, `B`, `C`, and `D` have the given types.

Unification

Both polymorphic type inference and pattern matching in OCaml are instances of a very general mechanism called ***unification***. Briefly, unification is the process of finding a substitution that makes two given terms equal. Pattern matching in OCaml is done by applying unification to OCaml expressions (e.g. `Some x`), whereas type inference is done by applying unification to type expressions (e.g. `'a -> 'b -> 'a`). It is interesting that both these procedures turn out to be applications of the same general mechanism. There are many other applications of unification in computer science; e.g., the programming language Prolog is based on it.

The essential task of unification is to find a substitution `S` that ***unifies*** two given terms (i.e., makes them equal). Let's write `s S` for the result of applying the substitution `S` to the term `s`. Thus, given `s` and `t`, we want to find `S` such that `s S = t S`. Such a substitution `S` is called a ***unifier*** for `s` and `t`. For example, given the two terms

```
f x (g y)      f (g z) w
```

where `x`, `y`, `z`, and `w` are variables, the substitution

```
S  =  [x <- g z, w <- g y]
```

would be a unifier, since

```
f x (g y) [x <- g z, w <- g y]
= f (g z) w [x <- g z, w <- g y]
= f (g z) (g y).
```

Note that this is a purely syntactic definition; the meaning of expressions is not taken into consideration when computing unifiers.

Unifiers do not necessarily exist. For example, the terms `x` and `f x` cannot be unified, since no substitution for x can make the two terms equal. Even when unifiers exist, they are not necessarily unique. For example, the substitution

```
T  =  [x <- g (f a b), y <- f b a, z <- f a b, w <- g (f b a)]
```

is also a unifier for the two terms above:

```
f x (g y) T  =  f (g z) w) T  =  f (g (f a b)) (g (f b a)).
```

However, when a unifier exists, there is a ***most general unifier (mgu)*** that is unique up to renaming. A unifier `S` for `s` and `t` is an ***mgu*** for `s` and `t` if

- `S` is a unifier for `s` and `t`; and
- any other unifier `T` for `s` and `t` is a ***refinement*** of `S`; that is, `T` can be obtained from `S` by doing further substitutions.

For example, the substitution `S` in the example above is an mgu for `f x (g y)` and `f (g z) w`. The unifier `T` is a refinement of `S`, since `T = SU`, where

```
U  =  [z <- f a b, y <- f b a].
```

Note that

```
f x (g y) S U
= f x (g y) [x <- g z, w <- g y] [z <- f a b, y <- f b a]
= f (g z) (g y) [z <- f a b, y <- f b a]
= f (g (f a b)) (g (f b a))
= f x (g y) T.
```

Unification Algorithm

We need unification for not just for pairs of terms, but more generally, for sets of pairs of terms. We say that a substitution `S` is a unifier for $[(s_1,t_1),...,(s_n,t_n)]$ if $s_i\ S = t_i\ S$ for all `1 <= i <= n`. The unification algorithm consists of two mutually recursive procedures `unify` and `unify_one`, which try to unify a list of pairs and a single pair, respectively. The result of the computation is the most general unifier for the list of pairs or the pair, respectively.

```
type id = string

type term =
  | Var of id
  | Term of id * term list

(* invariant for substitutions: *)
(* no id on a lhs occurs in any term earlier in the list *)
type substitution = (id * term) list

(* check if a variable occurs in a term *)
let rec occurs (x : id) (t : term) : bool =
  match t with
  | Var y -> x = y
  | Term (_, s) -> List.exists (occurs x) s

(* substitute term s for all occurrences of variable x in term t *)
let rec subst (s : term) (x : id) (t : term) : term =
  match t with
  | Var y -> if x = y then s else t
  | Term (f, u) -> Term (f, List.map (subst s x) u)

(* apply a substitution right to left *)
let apply (s : substitution) (t : term) : term =
  List.fold_right (fun (x, u) -> subst u x) s t

(* unify one pair *)
let rec unify_one (s : term) (t : term) : substitution =
  match (s, t) with
  | (Var x, Var y) -> if x = y then [] else [(x, t)]
```

```
  | (Term (f, sc), Term (g, tc)) ->
      if f = g && List.length sc = List.length tc
      then unify (List.combine sc tc)
      else failwith "not unifiable: head symbol conflict"
  | ((Var x, (Term (_, _) as t)) | ((Term (_, _) as t), Var x)) ->
      if occurs x t
      then failwith "not unifiable: circularity"
      else [(x, t)]

(* unify a list of pairs *)
and unify (s : (term * term) list) : substitution =
  match s with
  | [] -> []
  | (x, y) :: t ->
      let t2 = unify t in
      let t1 = unify_one (apply t2 x) (apply t2 y) in
      t1 @ t2
```

Type Inference and Unification

Now we show how type inference in OCaml can be done with unification on type expressions. Keep the matrix example above in mind; we will be doing roughly the same thing, but with different typing rules.

For simplicity, let's take a very small subset of OCaml consisting of

```
variables     x, y, ...
expressions   e  ::=  x | fun x -> e | (e1 e2)
```

This subset has a name: the ***λ-calculus*** (***lambda calculus***). Let's also introduce some *type expressions:*

```
type variables    'a, 'b, ...
type expressions  s  ::=  'a | s -> t
```

Take care that these are two separate classes of expressions; they cannot be mixed.

We will assume for simplicity that all bound variables are distinct; that is, no variable is bound twice in two different subexpressions of the form `fun x -> ...`. We can always rename bound variables if necessary to make this true. This is called ***α-conversion*** (***alpha-conversion***). It's not strictly necessary (in fact, the code we have given you does not assume it), but it does make the typing rules a little simpler. It is always ok to α-convert, since for example the expressions `fun x -> x + 3` and `fun y -> y + 3` are semantically equivalent.

The typing rules are

$$\frac{e_1 : s \to t \quad e_2 : s}{(e_1\ e_2) : t} \qquad \frac{x : s \quad e : t}{\texttt{fun x -> e} : s \to t}$$

The first rule says that the function application $(e_1\ e_2)$ only makes sense if e_1 is a function, i.e. has a type of the form `s -> t`, and the input type of e_1 is the same as the type of its argument e_2. When these premises are satisfied, then the result, represented by the expression $(e_1\ e_2)$, has the same type as the result type of e_1. The second rule says that the expression `fun x -> e` represents a function taking elements of the same type as `x` to elements of the type of `e`.

The rules are slightly more complicated without α-conversion, but not much. Essentially, it is necessary to maintain a *type environment*, and type inferences are done with respect to that environment.

These rules impose constraints as follows. Suppose we want to do type inference on a given expression `e`. We first assign unique type variables `'a`, ...

- one to each variable `x` occurring in `e`, and
- one to each *occurrence* of each subexpression of `e`.

Note that in the former clause, the type variable is associated with the variable, and in the latter, it is associated with the *occurrence* of the subexpression in `e`. Call the type variable assigned to `x` in the former clause `u(x)`, and call the type variable assigned to occurrence of a subexpression `e'` in the latter clause `v(e')`.

Now we take the following constraints:

- $u(x) = v(x)$ for each occurrence of a variable `x`
- $v(e_1) = v(e_2) \to v((e_1\ e_2))$ for each occurrence of a subexpression $(e_1\ e_2)$
- $v(\texttt{fun x -> e}) = v(x) \to v(e)$ for each occurrence of a subexpression `fun x -> e`.

This gives us a list of pairs of type expressions representing type constraints imposed by the typing rules above.

Now given an expression **e** whose polymorphic type we would like to infer, we can walk the abstract syntax tree of **e** and collect these constraints, then perform unification on the constraints to obtain their most general unifier. The resulting substitution applied to the type variable **v(e)** gives the most general polymorphic type of **e**.

The complete code for unification and simple polymorphic type inference can be downloaded from **here**. The following is some sample output. The last term is not typable because unification results in a circularity.

```
? fun x -> x
fun x -> x : 'a -> 'a
? fun x -> fun y -> x
fun x -> fun y -> x : 'a -> 'b -> 'a
? fun x -> fun y -> y
fun x -> fun y -> y : 'a -> 'b -> 'b
? fun f -> fun g -> fun x -> f (g x)
fun f -> fun g -> fun x -> f (g x) : ('e -> 'd) -> ('c -> 'e) -> 'c -> 'd
? fun x -> fun y -> fun z -> x z (y z)
fun x -> fun y -> fun z -> x z (y z) : ('c -> 'e -> 'd) -> ('c -> 'e) -> 'c -> 'd
? fun f -> (fun x -> f x x) (fun y -> f y y)
not unifiable: circularity
?
```