Lecturer: Dexter Kozen

1 Type Inference

Type inference refers to the process of determining the appropriate types for expressions based on how they are used. For example, ML knows that in the expression (f 3), 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 ML would assign f the type int ->'a.

```
- fn f => (f 3);
val it = fn : (int -> 'a) -> 'a
```

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.

```
- fn f => f(f 3);
val it = fn : (int -> int) -> int
- fn f => f(f "hi");
val it = fn : (string -> string) -> string
- fn f => f(f 3,f 4);
stdIn:19.9-19.19 Error: operator and operand don't agree [literal]
  operator domain: int
  operand:         'Z * 'Z
  in expression:
    f (f 3,f 4)
```

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.

If a program is well-typed, then a type can be inferred. For example, consider the program

let $square = \lambda z.\, z * z$ in
  $\lambda f.\, \lambda x.\, \lambda y.$
    if $(f\ x\ y)$
      then $(f\ (square\ x)\ y)$
      else $(f\ x\ (f\ x\ y))$

We are applying the multiplication operator to z, therefore we must have $z : \mathsf{int}$, thus $\lambda z.\, z * z : \mathsf{int} \to \mathsf{int}$ and $square : \mathsf{int} \to \mathsf{int}$. We know that the type of f must be something of the form $f : \sigma \to \tau \to \mathsf{bool}$ for some σ and τ, since it is applied to two arguments and its return value is used in a conditional test. Since f is applied to the value of $square\ x$ as its first argument, it must be that $\sigma = \mathsf{int}$. Since f is applied to the value of $f\ x\ y$ as its second argument, it must be that $\tau = \mathsf{bool}$. The return value is also bool. Thus the type of the entire program is $(\mathsf{int} \to \mathsf{bool} \to \mathsf{bool}) \to \mathsf{int} \to \mathsf{bool} \to \mathsf{bool}$.

2 Unification

Both type inference and pattern matching in ML 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 ML is done by applying unification to ML expressions, whereas type inference is done by applying unification to type expressions. 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; for example, the programming language PROLOG is based on it.

The essential task of unification is to find a substitution S that unifies two given terms (that is, makes them equal). Let's write $s\ S$ for the result of applying the substitution S to the term s. For example,

$$f(x, h(x, y))\,\{g(y)/x,\ z/y\} \quad = \quad f(g(y), h(g(y), z)),$$

where the substitution operator $\{g(y)/x,\ z/y\}$ applied to a term simultaneously substitutes $g(y)$ for x and z for y. The substitution is simultaneous, not sequential. Sequential substitution would give a different result:

$$f(x, h(x, y))\{g(y)/x\}\{z/y\} \quad = \quad f(g(y), h(g(y), y))\{z/y\} \quad = \quad f(g(z), h(g(z), z)).$$

Thus, given s and t, we want to find S such that $sS = tS$. Such a substitution S is called a *unifier* for s and t. For example, given the terms

$$f(x, g(y)) \qquad\qquad f(g(z), w) \tag{1}$$

the substitution

$$S \quad = \quad \{g(z)/x,\ g(y)/w\} \tag{2}$$

would be a unifier, since

$$f(x, g(y))\{g(z)/x,\ g(y)/w\} \quad = \quad f(g(z), w)\{g(z)/x,\ g(y)/w\} \quad = \quad f(g(z), g(y)).$$

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 unique. For example, the substitution

$$T \quad = \quad \{g(f(a, b))/x,\ f(b, a)/y,\ f(a, b)/z,\ g(f(b, a))/w\}$$

is also a unifier for the two terms (1):

$$f(x, g(y))\ T \quad = \quad f(g(z), w)\ T \quad = \quad f(g(f(a, b)), g(f(b, a))).$$

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

- S is a unifier for s and t,
- 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 $T = S\ U$, where

$$U \quad = \quad \{f(a, b)/z,\ f(b, a)/y\}.$$

Note that

$$\begin{aligned}
f(x, g(y))\ S\ U \quad &= \quad f(x, g(y))\{g(z)/x,\ g(y)/w\}\{f(a, b)/z,\ f(b, a)/y\} \\
&= \quad f(g(z), g(y))\{f(a, b)/z,\ f(b, a)/y\} \\
&= \quad f(g(f(a, b)), g(f(b, a))) \\
&= \quad f(x, g(y))\ T.
\end{aligned}$$

Note that we can compose substitutions, as we did in $S\ U$. This is the substitution that first applies S, then applies U to the result. The composition is also a substitution.

3 Unification Algorithm

The unification algorithm is known as Robinson's algorithm (1965). We need unification for not just for a pair of terms, but more generally, for a set of pairs of terms. We say that a substitution S is a *unifier* for a set $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ if $s_i\ S = t_i\ S$ for all $1 \leq i \leq n$.

The unification algorithm is given in terms of a function Unify() that takes a set of pairs of terms (s, t) and produces their mgu, if it exists. If E is a set of pairs of terms, then $E\{t/x\}$ denotes the result of applying the substitution $\{t/x\}$ to all the terms in E.

- $\text{Unify}(\{(x, t)\} \cup E) \triangleq \{t/x\}\ \text{Unify}(E\{t/x\})$ if $x \notin FV(t)$
- $\text{Unify}(\varnothing) \triangleq I$ (the identity substitution $x \mapsto x$)
- $\text{Unify}(\{(x, x)\} \cup E) \triangleq \text{Unify}(E)$
- $\text{Unify}(\{(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n))\} \cup E) \triangleq \text{Unify}(\{(s_1, t_1), \ldots, (s_n, t_n)\} \cup E)$.

In the first rule, $\{t/x\}$ denotes the substitution that substitutes t for x, and $\{t/x\}\ \text{Unify}(E\{t/x\})$ denotes the composition of $\{t/x\}$ and $\text{Unify}(E\{t/x\})$. Since we write substitutions on the right, we follow the convention that composition is from left to right; thus $S\ T$ means, "do S, then do T".

One circumstance that causes a set of terms not to unify is if it contains a pair (x, t) where $x \neq t$ but x occurs in t; then no substitution can make x and t equal.

4 Type Inference and Unification

Now we show how to do type inference using unification on type expressions. This technique gives the most general type (mgt) of any typable term; any other type of this term is a substitution instance of its most general type. Recall the Curry-style simply typed λ-calculus with syntax

$$e \ ::= \ x \mid e_1\ e_2 \mid \lambda x.\, e \qquad\qquad \tau \ ::= \ \alpha \mid \tau_1 \to \tau_2$$

and typing rules

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1\ e_2) : \tau} \qquad\qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \sigma \to \tau}.$$

For the language of types, the last unification rule translates to

- $\text{Unify}(\{(s \to s', t \to t')\} \cup E) \triangleq \text{Unify}(\{(s, t), (s', t')\} \cup E)$.

The problem is that any type derivation starts with assumptions about the types of the variables in the form of a type environment Γ, but without a type environment or an annotation as in Church style, we do not know what these are. However, we can observe that the form of the subterms impose constraints on the types. We can write down these constraints and then try to solve them.

Suppose we want to infer the type of a given λ-term e. Without loss of generality, suppose we have α-converted e so that no variable is bound more than once and no variable with a binding occurrence λx also occurs free.

Let $e_1, \ldots, e_m$ be an enumeration of all *occurrences* of subterms of e. We first assign a unique type variable α_i to each e_i, $1 \leq i \leq m$, as well as a unique type variable β_x to each variable x. Then we take the following constraints:

- if e_i is an occurrence of a variable x, the constraint $\alpha_i = \beta_x$;
- for a subterm $e_i = e_j\ e_k$, the constraint $\alpha_j = \alpha_k \to \alpha_i$; and
- for a subterm $e_i = \lambda x.\, e_j$, the constraint $\alpha_i = \beta_x \to \alpha_j$.

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

Now we do unification on the constraints and apply the resulting substitution to the type variable α_e. The result is the mgu of e.

4.1 An Example

Here is an example of the algorithm applied to the S combinator $\lambda xyz.\, xz(yz)$. Let us mark the second occurrence of z as z' to distinguish it from the first occurrence, although they are occurrences of the same variable z. Thus $S = \lambda xyz.\, xz(yz')$. Each occurrence of a subterm generates a constraint:

$$
\begin{array}{rcll}
e_1 & = & \lambda x.\, \lambda y.\, \lambda z.\, xz(yz') & \alpha_1 = \beta_x \to \alpha_2 \\
e_2 & = & \lambda y.\, \lambda z.\, xz(yz') & \alpha_2 = \beta_y \to \alpha_3 \\
e_3 & = & \lambda z.\, xz(yz') & \alpha_3 = \beta_z \to \alpha_4 \\
e_4 & = & xz(yz') & \alpha_5 = \alpha_8 \to \alpha_4 \\
e_5 & = & xz & \alpha_6 = \alpha_7 \to \alpha_5 \\
e_6 & = & x & \alpha_6 = \beta_x \\
e_7 & = & z & \alpha_7 = \beta_z \\
e_8 & = & yz' & \alpha_9 = \alpha_{10} \to \alpha_8 \\
e_9 & = & y & \alpha_9 = \beta_y \\
e_{10} & = & z' & \alpha_{10} = \beta_z.
\end{array}
$$

Solving these constraints using Robinson's algorithm yields

$$
\begin{array}{rcl}
\alpha_1 & = & (\alpha_7 \to \alpha_8 \to \alpha_4) \to (\alpha_7 \to \alpha_8) \to \alpha_7 \to \alpha_4 \\
\alpha_2 & = & (\alpha_7 \to \alpha_8) \to \alpha_7 \to \alpha_4 \\
\alpha_3 & = & \alpha_7 \to \alpha_4 \\
\alpha_5 & = & \alpha_8 \to \alpha_4 \\
\alpha_6 & = & \alpha_7 \to \alpha_8 \to \alpha_4 \\
\alpha_9 & = & \alpha_7 \to \alpha_8 \\
\alpha_{10} & = & \alpha_7 \\
\beta_x & = & \alpha_7 \to \alpha_8 \to \alpha_4 \\
\beta_y & = & \alpha_7 \to \alpha_8 \\
\beta_z & = & \alpha_7
\end{array}
$$

so we see that the most general type of e_1 is $(\alpha_7 \to \alpha_8 \to \alpha_4) \to (\alpha_7 \to \alpha_8) \to \alpha_7 \to \alpha_4$.