The Hindley-Milner Type Inference Algorithm

The Hindley-Milner algorithm reduces the problem of type inference (or: type assigment, type synthesis) to a unification problem. For the sake of simplicity we define a small sublanguage of ML and describe the algorithm for this sublanguage in all detail. It can then be imagined how to scale up the algorithm to the full language (with pairs, lists, recursion etc.), but this will not be described here. The types and the expressions of the sublanguage are defined by the following BNF syntax.

```
<type> ::= <typeconst> | <typevar> | (<type>-><type>)
<typeconst> ::= int | real | bool | ...
<typevar> ::= '<lower case letter>

<expr> ::= <constant> | <variable> | (<expr><expr>)
         | (fn <variable> => <expr>)
<constant> ::= 0 | 1 | ... | 3.14 | ... | true | false
<variable> ::= <lower case letter>

<typeassignment> ::= <variable>:<type>
```

A *type environment* is a sequence of type assignments to distinct expression variables. We will use symbols like M, x, τ, Γ to denote arbitrary expressions, variables, types and type environments, respectively. Type environments will simply be called *environments*. The empty environment will be denoted by $\varnothing$. Extending Γ by $x{:}\tau$ will be denoted by $\Gamma, x{:}\tau$ and $lookup(x, \Gamma)$ denotes the type (if any) of variable x in environment Γ. Constants are assumed to have a fixed type, like `0:int, false:bool`, which can also be looked up. For any substitution σ of types for type variables we denote the result of applying σ to $\Gamma, \tau, \ldots$ by $\Gamma\sigma, \tau\sigma, \ldots$ Finally, $\Gamma \vdash M : \tau$ expresses that M has type τ in environment Γ.

We formulate the precise result in the following theorem.

0.1. Theorem. *Let M be an expression and Γ an environment assigning types to all free (= not bound by* `fn`*) variables of M. Let τ be a type. Then we can compute a set $E(\Gamma, M, \tau)$ of equations between types such that for every substitution σ of types for type variables:*

1. *If σ is a unifier of $E(\Gamma, M, \tau)$, then $\Gamma\sigma \vdash M : \tau\sigma$.*

2. *If $\Gamma\sigma \vdash M : \tau\sigma$, then σ can be extended to a unifier of $E(\Gamma, M, \tau)$.*

In the latter case we have to extend σ since the algorithm introduces auxiliary type variables.

Thus, in order to solve the typing problem $\Gamma \vdash M : \tau$ we compute $E(\Gamma, M, \tau)$ and take this set as a unification problem. Here the variables are type variables and we have type constants and one infix, right-associative, binary function symbol `->`. If we can solve the unification problem we have a valid typing. If not, we know that no such typing exists. Taking for τ a type vrariable we can infer a type for M in Γ provided there exists one.

As a way of proving this theorem we give the algorithm. Thereafter we discuss a number of instructive examples. The first very simple example is about typing the identity function. The second example, concerning self-application, shows that the generated set of equations may fail to have a unifier, in which case there is no type. Thereafter a number of exercises are given.

Let Γ, M, τ be given. The algorithm uses recursion on the structure of M. Recall that M is either a constant, a variable, an application term or an abstraction term (with `fn <var> => ...`).

- $E(\Gamma, c, \tau)$ consists of the equation $\tau = \theta$, with θ the type of constant c.
- $E(\Gamma, x, \tau)$ consists of the equation $\tau = \mathit{lookup}(x, \Gamma)$.
- In case of an application term, we decompose the problem as follows:

  $$E(\Gamma, MN, \tau) = E(\Gamma, N, \texttt{'a}) \cup E(\Gamma, M, \texttt{'a}{\rightarrow}\tau)$$

  Here `'a` must be a fresh type variable to avoid name conflicts.

- In case of an abstraction term, we assign to `x` a yet unknown type, say a type variable `'a` and obtain $E(\Gamma, \texttt{fn x => } M, \tau)$ by adding $\tau = \texttt{'a->'b}$ to $E(\Gamma, \texttt{x:'a }, M, \texttt{'b})$. Again `'a` and `'b` must be fresh type variables.

0.2. Example. Let Γ be empty, $M \equiv \texttt{fn x => x}$ the identity function and `'t` a type variable. In order to type M we compute $E(\varnothing, M, \texttt{'t})$ assigning to the variable `x` a yet unknown type, say a type variable `'a`. Thereafter we type the body of M in the environment `x:'a`, that is, compute $E($`x:'a,x,'b`$)$ for some new type variable `'b` and add the equation `'t` $=$ `'a->'b`. As the body of M happens to be the variable `x` itself, $E($`x:'a ,x,'b`$)$ consists of the equation `'b` $=$ *lookup*$($`x,x:'a`$)$ $=$ `'a`. In total we have generated the set of equations $E(\varnothing, M,$ `'t`$) = \{$`'b` $=$ `'a`, `'t` $=$ `'a->'b`$\}$, whose most general unifier σ satisfies `'t`σ $=$ `'a`$\to$`'a`. The type `'a`$\to$`'a` is polymorphic, which expresses that `fn x => x` can be used as the identity function for any type.

0.3. Example. Let Γ be the environment `x:'a`, $N \equiv$ `xx` an expression and `'t` again a type variable. In order to type N in environment Γ, we compute $E(\Gamma,$`xx,'t`$)$ decomposing the application `xx` in two terms having (yet unknown) types in environment Γ. If the rightmost `x` can be typed, say, by type variable `'b`, then the leftmost `x` must have type `'b->'t` in environment Γ. This leads to two typing problems, the 'right' one $E($`x:'a ,x,'b`$)$ generating the equation `'b` $=$ `'a` as in the previous example, and the 'left' one $E($`x:'a ,x,'b->'t`$)$ generating the equation `'b->'t` $=$ `'a`. The resulting set of equations $\{$`'b` $=$ `'a`, `'b->'t` $=$ `'a`$\}$ has no unifier (occur-check!), since `'a` $=$ `'a->'t` is circular. We conclude that self-application cannot be typed in ML.

0.4. Exercise. If possible, infer types for the following closed terms. Use ML to check your answers.

- `fn f => fn x => x;`
- `fn f => fn x => f (f x);`
- `fn f => fn x => f x (f x);`
- `fn f => fn x => x (f x);`
- `fn f => fn x => x f x;`
- `fn h => fn t => fn f => fn x => f h (t f x);`
- `fn l => l (fn x => fn y => false) true;`
- `fn x => fn y => fn z => x z (y z);`