Type Unification

Chris Shaver

October 29, 2012

1 Type Unification Example

The following is a possible definition for a *fold-left* function in a functional language.

$$\textbf{def}\ foldl\ f\ z\ l\ =\ \textbf{if}\ l \equiv []\ \textbf{then}\ z\ \textbf{else}\ foldl\ f\ (f\ z\ (head\ l))\ (tail\ l)$$

decomposing this function into its constituting syntactic parts one gets the following tree-like structure

$$\begin{aligned}
&\textbf{def}\ foldl\ f\ z\ l\ =\ E_1\ \textbf{where}\\
&\quad E_1 = \textbf{if}\ C\ \textbf{then}\ z\ \textbf{else}\ E_2\\
&\quad C = l \equiv []\\
&\quad E_2 = foldl\ f\ E_3\ E_4\\
&\quad E_3 = f\ z\ E_5\\
&\quad E_4 = tail\ l\\
&\quad E_5 = head\ l
\end{aligned}$$

where after the first line the equations denote binding the constituting subexpressions of the function definition to names C and E_k. This is, in essence, the structure of the AST, and it might be noted that in functional languages, where programs are expressions, the AST is structured very much how one would simply parse the expression in a mathematical fashion.

Given this decomposition, new type variables can be given to each subexpression, variable, and function:

$$\begin{aligned}
&foldl : F\\
&\textbf{def}\ foldl\ (f:\mu)\ (z:\zeta)\ (l:\nu)\ =\ E_1\ \textbf{where}\\
&\quad E_1 : \alpha = \textbf{if}\ C\ \textbf{then}\ z\ \textbf{else}\ E_2\\
&\quad C : \beta = l \equiv []\\
&\quad E_2 : \gamma = foldl\ f\ E_3\ E_4\\
&\quad E_3 : \delta = f\ z\ E_5\\
&\quad E_4 : \epsilon = tail\ l\\
&\quad E_5 : \eta = head\ l
\end{aligned}$$

Additionally, suppose it is given that the types of functions *head* and *tail* are $List\ \psi \to \psi$ and $List\ \psi \to List\ \psi$, respectively, where ψ is a free type variable. Using the typing rules for each syntactic production, the following

constraints can be derived from the function definition and each subsequent subexpression:

$foldl : F$	
def $foldl\,(f:\mu)\,(z:\zeta)\,(l:\nu)\ =\ E_1$ **where**	$F = \mu \to \zeta \to \nu \to \alpha$
$E_1 : \alpha =$ **if** C **then** z **else** E_2	$\beta = BOOL,\ \zeta = \alpha,\ \gamma = \alpha$
$C : \beta = l \equiv []$	$\nu = List\,\phi$
$E_2 : \gamma = foldl\ f\ E_3\ E_4$	$F = \mu \to \delta \to \epsilon \to \gamma$
$E_3 : \delta = f\ z\ E_5$	$\mu = \zeta \to \eta \to \delta$
$E_4 : \epsilon = tail\ l$	$\epsilon = List\,\psi,\ \nu = List\,\psi$
$E_5 : \eta = head\ l$	$\eta = \kappa,\ \nu = List\,\kappa$

In the above, the variables ϕ, ψ, and κ are introduced as free variables for the polymorphic value [], the empty list, and the two list functions *head* and *tail*. Having derived all of the necessary type constraint formulae, unification can be performed to get a set of substitutions for these type variables that replace them with type expressions involving constant types and a set free type variables. This substitution is know as a *unifier*. To perform unification, the above list of constraints are unified in order producing as a consequence a sequence of substitutions or *bindings*. At each step, the *bindings* generated in previous steps are used. The algorithm proceeds as follows:

1. Bind: $F \mapsto \mu \to \zeta \to \nu \to \alpha$
2. Bind: $\beta \mapsto BOOL$
3. Bind: $\zeta \mapsto \alpha$
4. Bind: $\gamma \mapsto \alpha$
5. Bind: $\nu \mapsto List\,\phi$
6. $\Rightarrow\ \mu \to \zeta \to \nu \to \alpha = \mu \to \delta \to \epsilon \to \gamma$
   (a) $\mu = \mu\ \Rightarrow$ Check: $\mu = \mu$
   (b) $\delta = \zeta\ \Rightarrow$ Bind: $\delta \mapsto \alpha$
   (c) $\epsilon = \nu\ \Rightarrow$ Bind: $\epsilon \mapsto List\,\phi$
   (d) $\gamma = \alpha\ \Rightarrow$ Check: $\alpha = \alpha$
7. Bind: $\mu \mapsto \zeta \to \eta \to \delta$
8. $\Rightarrow\ List\,\phi = List\,\psi$
   (a) $\phi = \psi\ \Rightarrow$ Bind: $\phi \mapsto \psi$
9. $\Rightarrow\ List\,\phi = List\,\psi$
   (a) $\phi = \psi\ \Rightarrow$ Check: $\psi = \psi$
10. Bind: $\eta \mapsto \kappa$
11. $\Rightarrow\ List\,\phi = List\,\kappa$
    (a) $\phi = \kappa\ \Rightarrow$ Bind: $\psi \mapsto \kappa$

On the sixth step of this process, a small simplification was made for brevity in the listing of the steps. In the equation

$$\mu \to \zeta \to \nu \to \alpha = \mu \to \delta \to \epsilon \to \gamma$$

the function type operator $\to$ is a right-associative binary operator, thus this equation can be written specifically

$$\mu \to (\zeta \to (\nu \to \alpha)) = \mu \to (\delta \to (\epsilon \to \gamma))$$

Although I wrote out all of the steps in parallel, in the actual algorithm the leftmost $\to$ initiates two sub-unifications of the left and right operands.

$$\mu = \mu,\ \zeta \to (\nu \to \alpha) = \delta \to (\epsilon \to \gamma)$$

The unification of the right operands then initiates, itself, two sub-unifications for its left and right operands.

$$\zeta = \delta,\ \nu \to \alpha = \epsilon \to \gamma$$

And likewise, there is the final bifurcation of the right operands to

$$\nu = \epsilon,\ \alpha = \gamma$$

Firstly, that the above process succeeded, not running into a condition of unsatisfiability, indicates that the set of constraints is indeed satisfiable. Accumulating the bindings established during the process, one gets the following set of substitutions:

$$\begin{aligned}
&F \mapsto \mu \to \zeta \to \nu \to \alpha \\
&\beta \mapsto \mathit{BOOL} \\
&\zeta \mapsto \alpha \\
&\gamma \mapsto \alpha \\
&\nu \mapsto \mathit{List}\ \phi \\
&\delta \mapsto \alpha \\
&\epsilon \mapsto \mathit{List}\ \phi \\
&\mu \mapsto \zeta \to \eta \to \delta \\
&\phi \mapsto \psi \\
&\eta \mapsto \kappa \\
&\psi \mapsto \kappa
\end{aligned}$$

Taking these substitutions transitively to their ends in expressions of free variables the following set of substitutions is derived:

$$\begin{aligned}
&F \mapsto (\alpha \to \kappa \to \alpha) \to \alpha \to \mathit{List}\ \kappa \to \alpha \\
&\beta \mapsto \mathit{BOOL} \\
&\zeta,\ \gamma,\ \delta \mapsto \alpha \\
&\nu,\ \epsilon \mapsto \mathit{List}\ \kappa \\
&\mu \mapsto \alpha \to \kappa \to \alpha \\
&\phi,\ \eta,\ \psi \mapsto \kappa
\end{aligned}$$

In these substitutions, the only free variables that remain on the RHS are α and κ. Every other variable can be substituted with expressions involving these two free variables alnil with type constants. Performing

the above substitutions one gets the type of the function *foldl* alnil with the types of every subexpression in terms of κ and α.

$$
\begin{aligned}
&\mathit{foldl} : (\alpha \to \kappa \to \alpha) \to \alpha \to \mathit{List}\,\kappa \to \alpha \\
&\textbf{def}\ \mathit{foldl}\,(f : \alpha \to \kappa \to \alpha)\,(z : \alpha)\,(l : \mathit{List}\,\kappa)\ =\ E_1\ \textbf{where} \\
&\quad E_1 : \alpha = \textbf{if}\ C\ \textbf{then}\ z\ \textbf{else}\ E_2 \\
&\quad C : \mathit{BOOL} = l \equiv [] \\
&\quad E_2 : \alpha = \mathit{foldl}\,f\,E_3\,E_4 \\
&\quad E_3 : \alpha = f\,z\,E_5 \\
&\quad E_4 : \mathit{List}\,\kappa = \mathit{tail}\,l \\
&\quad E_5 : \kappa = \mathit{head}\,l
\end{aligned}
$$

Since *foldl* is ultimately a polymorphic function, it will take on a specific instance when applied to parameters that have specific types. For instance, a possible use of this function would be to count the number of 'a's in a string. Assume that a string is represented as a list of characters. The folding function would be as follows:

$$\textbf{def}\ f\,n\,c\ =\ \textbf{if}\ c \equiv \text{'a'}\ \textbf{then}\ n+1\ \textbf{else}\ n$$

Through explicit definition, or derived through some other process of inference, let the type be

$$f : \mathit{INT} \to \mathit{CHAR} \to \mathit{INT}$$

When *foldl* is partially applied to this function as a first parameter then, the type of *foldl* can be fully specified to

$$\mathit{foldl} : (\mathit{INT} \to \mathit{CHAR} \to \mathit{INT}) \to \mathit{INT} \to \mathit{List}\ \mathit{CHAR} \to \mathit{INT}$$

and the partially applied function would have the type

$$\mathit{foldl}\,f : \mathit{INT} \to \mathit{List}\ \mathit{CHAR} \to \mathit{INT}$$

The desired function can be even defined through partial application as

$$\textbf{def}\ \mathit{count\_a}\ =\ \mathit{foldl}\,f\,0$$

which would have a type of

$$\mathit{count\_a} : \mathit{List}\ \mathit{CHAR} \to \mathit{INT}$$

What can be seen from this process is how polymorphic functions can be composed, forming additional mutual constraints that may further specify types to narrower sets of possibilites. Fortunately, when unification finds the most general solution to the type constraints, composition can only make these solutions more specific. In other words, one never has to backtrack on the substitutions when a polymorphic function is applied in a particular context.