# Worksheet 8

# Unification

The aim of this worksheet is that

- you understand and become proficient in computing the most general unifier of two expressions.

This will be an essential skill that we need when considering the proof theory of clausal form logic, because each step of a derivation includes a unification process. (Unification also happens to be a technique much used in all branches of AI for determining whether two 'patterns' match. In our case, these patterns are wffs or just atoms.)

## 8.1 Unification

Let's begin with some terminology and notation. A *binding* will be a pair $V \mapsto e$. This says that the *variable, V,* is bound to the *expression, e.* The expression, $e$, will be a term, i.e. it may be a constant, another variable or a more complex expression. (In the last case, where $e$ is a more complex expression, $e$ must not itself contain $V$. For example, $X \mapsto f(X)$ is not a legal binding.)

More generally, we will need a *set of bindings*, also referred to as a *substitution*, e.g. $\{V_1 \mapsto e_1, V_2 \mapsto e_2, \ldots, V_n \mapsto e_n\}$.

We talk of *applying* a set of bindings to an expression. What this means is that variables in the expression are replaced by the values they are bound to (if any). The convention is to use Greek letters such as $\theta, \sigma$ and $\tau$ to denote a substitution/set of bindings, and to write the application of a substitution $\theta$ to an expression $e$ as $e\theta$ rather than use the usual functional notation $\theta(e)$.

| $e$ | $\theta$ | $e\theta$ |
|---|---|---|
| p($X$, f($Y$), b) | $\{X \mapsto Z,\ Y \mapsto W\}$ | p($Z$, f($W$), b) |
| p($X$, f($Y$), b) | $\{Y \mapsto \mathrm{c}\}$ | p($X$, f(c), b) |
| p($X$, f($Y$), b) | $\{X \mapsto \mathrm{g}(Z),\ Y \mapsto \mathrm{d}\}$ | p(g($Z$), f(d), b) |
| p($X$, f($Y$), b) | $\{X \mapsto \mathrm{c},\ Y \mapsto \mathrm{d}\}$ | p(c, f(d), b) |

Given two substitutions $\sigma$ and $\tau$ the *composition* of $\tau$ with $\sigma$, again written backwards as $\sigma\tau$, is the substitution that results from applying $\tau$ to the *expressions* in $\sigma$ and then adding any bindings in $\tau$ whose variables do not occur among the variables of $\sigma$.

| $\sigma$ | $\tau$ | $\sigma\tau$ |
|---|---|---|
| $\{Z \mapsto \mathrm{f}(X, Y)\}$ | $\{X \mapsto \mathrm{b},\ Y \mapsto \mathrm{c},\ W \mapsto \mathrm{d},\ Z \mapsto \mathrm{e}\}$ | $\{Z \mapsto \mathrm{f}(\mathrm{b},\ \mathrm{c}),\ X \mapsto \mathrm{b},\ Y \mapsto \mathrm{c},\ W \mapsto \mathrm{d}\}$ |

Two expressions, $e_1$ and $e_2$, *unify* (or are *unifiable*) if there exists a set of bindings, $\theta$, such that:

$$e_1\theta = e_2\theta$$

i.e. applying $\theta$ to each expression results in the same new expression. $\theta$ is called a *unifier* of $e_1$ and $e_2$.

Roughly speaking, $e_1$ and $e_2$ will unify if they are the same, except possibly that one has a variable where the other has a constant. However, in this latter case, all occurrences of the same variable in an expression must match consistently. (This is very rough: the algorithm given below will make the notion more exact.)

However, we are interested in the *most general unifier* of two expressions.

Consider expressions p($X$, f($Y$), b) and p($X$, f(b), b). Now, the set of bindings $\{X \mapsto \mathrm{c},\ Y \mapsto \mathrm{b}\}$ is a unifier, as we can see:

$$\mathrm{p}(X,\ \mathrm{f}(Y),\ \mathrm{b})\{X \mapsto \mathrm{c},\ Y \mapsto \mathrm{b}\} =$$
$$\mathrm{p}(X,\ \mathrm{f}(\mathrm{b}),\ \mathrm{b})\{X \mapsto \mathrm{c},\ Y \mapsto \mathrm{b}\} = \mathrm{p}(\mathrm{c},\ \mathrm{f}(\mathrm{b}),\ \mathrm{b})$$

The results are equal.

However, $\{Y \mapsto \mathrm{b}\}$ is also a unifier:

$$\mathrm{p}(X,\ \mathrm{f}(Y),\ \mathrm{b})\{Y \mapsto \mathrm{b}\} =$$
$$\mathrm{p}(X,\ \mathrm{f}(\mathrm{b}),\ \mathrm{b})\{Y \mapsto \mathrm{b}\} = \mathrm{p}(X,\ \mathrm{f}(\mathrm{b}),\ \mathrm{b})$$

The results are equal.

$\{Y \mapsto \mathrm{b}\}$ is a *more general* unifier than $\{X \mapsto \mathrm{c},\ Y \mapsto \mathrm{b}\}$ because, while managing to unify the expressions, it doesn't put an unnecessary constraint on the value bound to $X$.

More generally, $\tau$ is a most general unifier or *mgu* of expressions $e_1$ and $e_2$ if it has the property that if $\sigma$ is any unifier of the two expressions, then there exists a substitution $\delta$ such that

$$e_1\sigma = e_2\sigma = e_1\tau\delta = e_2\tau\delta$$

(The mgu is unique up to variable renaming.)

I give two versions of the unification algorithm: a DGB-bowdlerised version that I hope is easy(-ish) to understand, and a more formal version (from Nilsson). Both produce the *most general unifier*: the unifiers they produce do not contain unnecessary bindings. This is important in AI in general.

The algorithms take two expressions and step through their arguments in turn. They return either FAIL or the mgu (a set of bindings, which may be empty if the two lists do unify but are absolutely equal anyway).

```
----------------------------------------------------------------------

function UNIFY(E1, E2)  % DGB version

o  Step through the two atoms, argument by argument:
    -if the arguments are equal,
        great
    -if they are different,
        * if at least one is a variable,
             + create a binding (when forming a binding you should
               also check that the variable doesn't occur in the
               other argument when the other argument is a function
               applied to some arguments)
             + apply the binding to the rest of the atom
             + compose the binding with the set of bindings
               accumulated so far
        * if not, but both are compound terms that use the same
             function symbol,
             + step through the arguments of the function symbols
        * otherwise it all fails
o Continue to step through (unless you failed).
  (Fail if the atoms have different numbers of arguments.)
----------------------------------------------------------------------
```

```
--------------------------------------------------------------------

function UNIFY(E1, E2)  % Formal, Nilsson version

   Locals Head-E1, Tail-E1, Head-E2, Tail-E2, Head-bindings,
          Tail-bindings, New-E1-tail, New-E2-tail.


   If either E1 or E2 is a variable or a constant, then
      begin
      Exchange them if necessary so that E1 is the variable/constant
      If E1 = E2 then
         return {}
      Else
         If E1 is a variable then
            If E1 occurs in E2 then
               return FAIL
            Else
               return {E1 -> E2}
         Else
            If E2 is a variable then
               return {E2 -> E1}
            Else
               return FAIL
      end
      Head-E1 := first element of E1
      Tail-E1 := rest of E1
      Head-E2 := first element of E2
      Tail-E2 := rest of E2
      Head-bindings := UNIFY(Head-E1, Head-E2)
      If Head-bindings = FAIL then
         return FAIL
      Else
         begin
         New-E1-tail := APPLY-BINDINGS(Tail-E1, Head-bindings)
         New-E2-tail := APPLY-BINDINGS(Tail-E2, Head-bindings)
         Tail-bindings := UNIFY(New-E1-tail, New-E2-tail)
         If Tail-bindings = FAIL then
            return FAIL
         Else
            return COMPOSE(Head-bindings, Tail-bindings)
         end
      end

--------------------------------------------------------------------
```

So to unify two expressions, we go along them both, comparing them. If at any point we discover different constants, we fail. If one expression has a variable at one point, we construct a binding of the variable with whatever is at the same point in the other expression. We then replace all other instances of that variable in the expression with its value, and continue until we reach the end.

If we are not attempting to compare constants/variables, then we recursively attempt to unify the head of the expression, and if successful, use the resulting bindings to further instantiate the tails of the expressions. This guarantees that any variables are matched consistently: further occurrences of a particular variable are replaced by their value, and thus we will be either checking constants against constants, or variables against constants, in which case the same considerations will apply. Having done this, we now attempt to unify the instantiated tails of the expressions. If this succeeds, we compose the bindings of the two portions of the expressions and return the results.

The "E1 occurs in E2" check is to catch examples such as unifying

$$\mathrm{p}(\mathrm{a}, X, \mathrm{f}(\mathrm{b}, X))$$

and

$$\mathrm{p}(\mathrm{a}, Y, Y).$$

Without it, we would first match $Y$ with $X$, then replace $X$ through with $Y$. Trying to match the second argument with $Y = \mathrm{f}(\mathrm{b}, Y)$ would succeed if this check were not there, for a variable will match with anything. But this is clearly wrong: if $Y = \mathrm{f}(\mathrm{b}, Y)$ then $Y = \mathrm{f}(\mathrm{b}, \mathrm{f}(\mathrm{b}, Y))$ and $Y = \mathrm{f}(\mathrm{b}, \mathrm{f}(\mathrm{b}, \mathrm{f}(\mathrm{b}, Y)))$, ad infinitum.[1]

Some abstract examples are:

| *E1* | *E2* | *mgu* |
|---|---|---|
| p(a, b, c) | p(a, b, c) | $\{\}$ |
| p(a, b, c) | p(a, b, d) | FAIL |
| p(a, b, c) | p(a, b, $X$) | $\{X \mapsto \mathrm{c}\}$ |
| p(a, b, $X$) | p(a, b, c) | $\{X \mapsto \mathrm{c}\}$ |
| p(a, b, $X$) | p(a, b, $Y$) | $\{X \mapsto Y\}$ |
| p(a, $X$, c) | p(a, b, $Y$) | $\{X \mapsto \mathrm{b}, Y \mapsto \mathrm{c}\}$ |
| p(a, $X$, $Y$) | p(a, b, c) | $\{X \mapsto \mathrm{b}, Y \mapsto \mathrm{c}\}$ |
| p(a, b, c) | p(a, $X$, $X$) | FAIL |
| p(a, $X$, $X$) | p(a, b, b) | $\{X \mapsto \mathrm{b}\}$ |
| p(a, $X$, f(c, $X$)) | p(a, b, f(c, b)) | $\{X \mapsto \mathrm{b}\}$ |

[1] However, this check slows the algorithm down. In some systems, most notably in Prolog, it has not been implemented. Thus, Prolog's unification is faster but might hit the odd infinite loop if the programmer isn't careful.

**Exercise 8.1** *Determine whether the members of the following pairs of expressions unify with each other. If so, give the most general unifier; if not, give a brief explanation.*

| | | |
|---|---|---|
| 1. | *colour(tweety, yellow)* | $colour(X, Y)$ |
| 2. | *colour(tweety, yellow)* | $colour(X, X)$ |
| 3. | *colour(hat(postman_pat), blue)* | $colour(hat(Y), X)$ |
| 4. | $r(f(X), b)$ | $r(Y, Z)$ |
| 5. | $r(f(Y), X)$ | $r(X, f(b))$ |
| 6. | $r(f(Y), Y, X)$ | $r(X, f(a), f(V))$ |
| 7. | $loves(X, Y)$ | $loves(Y, X)$ |

**Exercise 8.2** *(This exercise is not essential. Do it only if you want to.)*

*In the file* `unify`*, there is Prolog code for unifying two terms, which you can look at and play with if you fancy. (Clearly, you could use it to do the exercise above but then you wouldn't master the eminently examinable skill of unification.)*

*(Obviously, Prolog has unification built-in, so writing an explicit unifier in Prolog is a bit of a weird thing to do. However, to improve its performance, Prolog's unifier does not perform the "occurs check". If you needed a unifier that did perform the check then you would need an explicit implementation such as the one in this file, which does have the "occurs check".)*

**Acknowledgements**

The unification algorithm comes from Nilsson, and the exercise comes from Genesereth & Nilsson.