0.1 Some ML code for unification

This code comes from a dialect of Ml known as HOL. All terms are regarded as curried: operator applied to operand. Thus HOL would regards $f(x, y)$ as $f(x,$ applied to y. `rev_itlist` iteratively applies a list of functions to an arguments to obtain a values. Thus `apply_subst` successively applies a list of substitutions to a term. A substitution is a pair of terms. `@` concatenates two lists.

```
let apply_subst l t = rev_itlist (\pair term.subst[pair]term) l t;;

% Find a substitution to unify two terms (lambda-terms not dealt with) %

letrec find_unifying_subst t1 t2 =
 if t1=t2
  then []
 if is_var t1
  then if not(mem t1 (frees t2)) then [t2,t1] else fail
 if is_var t2
  then if not(mem t2 (frees t1)) then [t1,t2] else fail
 if is_comb t1 & is_comb t2
  then
   (let rat1,rnd1 = dest_comb t1
    and rat2,rnd2 = dest_comb t2
    in
    let s = find_unifying_subst rat1 rat2
    in s@find_unifying_subst(apply_subst s rnd1)(apply_subst s rnd2)
   )else fail;;
```

This currying corresponds to a determination—when unifying (for example)—'$f(a, b, f(x))$' with '$f(x, y, w)$'—to detect $x \mapsto a$ and then do that to the third argument of the first occurrence of 'f' so that it becomes '$f(a)$' before we get there. This finesses questions about simultaneous *versus* consecutive execution of substitution.

0.2 Unification: an illustration

In the two axioms.

1. $(\forall xy)(x > y \to Sx > Sy)$
2. $(\forall w)(Sw > 0)$

'S' is the successor function: $S(x) = x + 1$. (Remember that $\mathbb{N}$ is the recursive datatype built up from 0 by means of the successor function.)

Now suppose we want to use PROLOG-style proof with resolution and unification to find a z such that $z > S0$. We turn 1 and 2 into clauses getting $\{\neg(x > y), Sx > Sy\}$ and $\{Sw > 0\}$, and the (negated) goal clause $\{\neg(z > S0)\}$.

The idea now is to refute this negated goal clause. Of course we can't refute it, beco's there are indeed some z of which this clause holds, but we might be able to refute some instances of it, and this is where unification comes in.

$z > S0$ will unify with $Sx > Sy$ generating the bindings $z \mapsto Sx$ and $y \mapsto 0$. We apply these bindings to the two clauses clauses $\{\neg(x > y), Sx > Sy\}$ and $\{\neg(z > S0)\}$, obtaining $\{\neg(x > S0), Sx > S0\}$ and $\{\neg(Sx > S0)\}$. These two resolve to give $\{\neg(x > 0)\}$. Clearly the substitution $x \mapsto Sw$ will enable us to resolve $\{\neg(x > 0)\}$ (which has become $\{\neg(Sw > 0)\}$) with $\{Sw > 0\}$ to resolve to give the empty clause. *En route* we have generated the bindings $z \mapsto Sx$ and $x \mapsto Sw$, which compose to give $z \mapsto SSw$, which tells us that the successor of the successor of any number is bigger than the successor of 0 as desired. Notice that the answer given by this binding ($z \mapsto SSW$) is the most general possible response to "find me something $> S0$". This is because the unification algorithm finds the most general answer.

The idea is this: We are trying to find a witness to $(\exists x)(A(x))$. Assume the negation of this, and try to refute it. In the course of refuting it we generate bindings that tell us what the witnesses are.

Higher-order Unification

Unification in first-order logic is well-behaved. For any two complex terms t_1 and t_2 if there is any unifier at all there is a most general unifier which is unique up to relettering. This doesn't hold for higher-order logic where there are function variables. It's pretty clear what you have to do if you want to unify $f(3)$ and 6: you replace f by something like

`if` $x = 3$ `then` 6 `else don't care`

(which one might perhaps write $(\epsilon f)(f(3) = 6)$).

However what happens if you are trying to unify $f(3)$ and $g(6)$? You want to bind 'f' to

$$\texttt{if } x = 3 \texttt{ then } g(6) \texttt{ else don't care} \tag{A}$$

but then you also want to bind 'g' to

$$\texttt{if } x = 6 \texttt{ then } f(3) \texttt{ else don't care} \tag{B}$$

and you have a vicious loop of substitutions. There are restricted versions that work, and there was even a product called `Q-PROLOG` ('Q' for Queensland) that did something clever. I've long ago forgotten.

I find in my notes various ways of coping with this, one using ϵ terms. One can have an epsilon term which is is a pair of things satisfying (A) and (B):

$$(\epsilon p)(\exists h_1, h_2)(p = \langle h_1, h_2\rangle \wedge h_1(3) = h_2(6))$$

so that we bind 'f' to '$\texttt{fst}(p)$' and 'g' to '$\texttt{snd}(p)$'.

$\star$ **20** $\star$ *For each of the following pairs of terms, give a most general unifier or explain why none exists.*
$f(g(x), z)$ *and* $f(y, h(y))$

$f(g(x), h(g(x)))$ is the most general unifier.

$j(x, y, z)$ *and* $j(f(y, y), f(z, z), f(a, a))$

$j(f(f(f(a, a), f(a, a)), f((a, a), f(a, a))), f(f(a, a), f(a, a)), f(a, a))$ is the most general unification.

$j(x, z, x)$ *and* $j(y, f(y), z)$

Any unification requires that $x = y = z$ and that $z = f(y)$ also. Therefore the terms cannot be unified without allowing $f(f(f(\cdots)))$.

$j(f(x), y, a)$ *and* $j(y, z, z)$

This cannot be unified because it required that $y = z = a$ and also that $y = f(x)$. This will only work if $f(x) = a$ for all x.

$j(g(x), a, y)$ *and* $j(z, x, f(z, z))$

$j(g(a), a, f(g(a), g(a)))$ is the most general unification.