FOL RESOLUTION AND UNIFICATION

We will consider an example from the book momentarily. But first we give a few "warmup" examples:

1. Suppose we want to prove Qc from Pc and ForAll x, Px → Qx, along lines similar to what we did in PL resolution. We first need to make sure all the wffs are in (equivalent) CNF form. Qx and Px already are. But what about the universally quantified wff? (FOL, CNF means a conjunction of disjunctions of individual predicate letters (with arguments) or their negations.)

   We rewrite the if-then part as usual, get ForAll x (Qx v –Px). Next we drop the ForAll x, with the understanding that any variables will be regarded as universal (how to deal with existential quantifiers will come soon). So now we proceed as follows:

   ```
   Pc      Qx v –Px            -Qc (the negation of what we want to prove)
    \         /               /
     \      / x ← c          /
       Qc _____________/
                  |
                 nil          And so we have proven Qc.
   ```

   What has happened here? The (universal) variable x has been replaced by the constant symbol c. This is an example of *unification*, a crucial part of FOL resolution.

   In general, two non-predicate expressions (made of variable and/or constants, and possibly function symbols) "unify" if one can make substitutions for the variables in such a way that the two become literally identical. And in that case, one goes for the *most general unifier* that does the trick. Thus for instance:

   x and f(x) cannot unify; nor can f(x) and g(x), nor c and f(x), etc

   f(x) and y unify by substituting f(x) for y; also by substituting, for instance, c for x and f(c) for y; or g(b) for x and f(g(b)) for y, etc. But of these, the first choice is the most general: it makes the fewest commitments and leaves open the most possibilities for further unifications later on.

2. Let Suppose we are given P(g(c)) and want to prove Exists x Px, where g is a function symbol. P(g(c)) is already in CNF. But we need to negate what we want to prove: –Exists x Px. This is equivalent to ForAll x –Px (sometimes this is even taken as the definition of Exists). Then proceeding as above:

   ```
   P(g(c))                -Px
      \                    /  x ← g(c)
       \  _____________   /
                |
               nil
   ```

   And so we have proven –Px, which really means ForAll x –Px, which is equivalent to Exists x Px. [Note: in FOL, wffs are equivalent if they have the same truth-value in all *interpretations*, much as in PL, except that an FOL interpretation is vastly more than a PL interpretation.]

3. Suppose P(c) and we want to prove Exists x P(g(x)). If we proceed as in example 2, we might find ourselves working with the wffs P(c) and -P(g(x)). And now we are stuck. There is no unifier for c and g(x)!

   And this is good, because in fact, Exists x P(g(x)) is not entailed by P(c). How can we be sure of that? Well, we need to show that it is *not the case* that Exists x P(g(x)) holds in all models of P(c) (interpretations where P(c) holds). And one way to do that is to come up with an example of an interpretation where P(c) holds but Exists x P(g(x)) does not. So we can try to find a domain D and a function g* that g can represent, and an element c* of D that c can represent, so that c* can never be a value of g*. [It's easy, and there are many many ways to do it, with numbers, people, whatever.]

4. Suppose we know Exists x Px and also ForAll x (Px → Qx), and want to prove Exists x Qx. The universal wff is easy to convert [Qx v –Px]; what about the existential one? If we simply remove the Exists x in front, we'll have an unquantified variable (called a *free* variable) and it will be used as if it were universal as in example 1 above, not what we want (and not equivalent to the wff it came from). The x in this case really represents a single but unknown item, and we'd like to use a constant symbol in its place. But we dare not use any symbol already in the language since it might also be present in some other wff that we could use later on, and we cannot assume the two wffs are "about" the same thing. So, we introduce a *new* constant symbol into the language, called a *Skolem constant*; for instance s1, say. Then we write P(s1) as a replacement for Exists x Px.

   But there is one thing we need to do before using s1. The variable x appears not only in Exists x Px, but also in the other axiom and in the "goal" wff that we want to prove. The goal will be negated and then replaced by ForAll x –Qx, and then by –Qx. This causes a clash with the x in Qx v –Px that we get from the other axiom; there is no reason that the two xs should refer to the same thing. So we need to "standardize" the variables, by making sure variables that are in the scope of different quantifiers are renamed so as to be different variable symbols. Thus we can, for instance, rewrite the universal wff axiom as ForAll y (Py → Qy) which then transforms into Qy v –Py.

   So now we have P(s1) and Qx v –Px, and the negated "goal" –Exists x Qx, giving us the following:

```
 P(s1)                 Qy v –Py                  -Qx
     \  ____________  /                            /
              | y ← s1                            /
          Q(s1) _______________________ /  x ← s1
                           nil
```

5. Now at last we return to the book's example about animals and people and love, except that we will make it a little more interesting and realistic. (Also, the book has some typos in their treatment.)

   "Everybody who loves all animals is loved by someone." This we represent in FOL as follows:

   ForAll x [(Hx & ForAll y (Ay → Lxy)) → Exists y (Hy & Lyx)]

   When we have finished looking at the example *per se*, we will then include another axiom, to the effect that "No one loves the person Charlie", represented as Hc & -Exists x (Hx & Lxc), and from these two wffs we'll prove Exists x (Ax & -Lcx).

   We follow the book's 6 steps (pp 346-7) for conversion of the main wff above to CNF:

   i. eliminate implications
   ii. move negations in
   iii. standardize variables
   iv. Skolemize existentials
   v. drop universals
   vi. distribute v over &.

In step i. we get

ForAll x [-(Hx & ForAll y (-Ay v Lxy)) v Exists y (Hy & Lyx)]

Step ii. gives

ForAll x [(-Hx v -ForAll y (-Ay v Lxy)) v Exists y (Hy & Lyx)] and then

ForAll x [(-Hx v Exists y -(-Ay v Lxy)) v Exists y (Hy & Lyx)]

then

ForAll x [(-Hx v Exists y (--Ay & -Lxy)) v Exists y (Hy & Lyx)]

and finally

ForAll x [(-Hx v Exists y (Ay & -Lxy)) v Exists y (Hy & Lyx)]

Step iii. replaces y in one Exists y(…) with a *new* variable, say z:

ForAll x [(-Hx v Exists y (Ay & -Lxy)) v Exists z (Hz & Lzx)]

Step iv. replaces existentially quantified variables with Skolem functions (since y and z may depend on the universal choice of x:

ForAll x [(-Hx v (A(f1(x)) & -L(x,f1(x)))) v (H(f2(x)) & L(f2(x),x))]

Step v. gives

[-Hx v (A(f1(x)) & -L(x,f1(x)))] v [H(f2(x)) & L(f2(x),x)]

And step vi. gives

[(-Hx v A(f1(x))) & (-Hx v –L(x,f1(x)))] v [H(f2(x)) & L(f2(x),x)]

At this point, for ease of reading, we introduce abbreviations for the predicate expressions, in the same left-right order:

[(P v Q) & (P v W)] v [R & S]

then

{ R v [(P v Q) & (P v W)] } & { S v [(P v Q) & (P v W)] }

{ (R v (PvQ)) & (R v (PvW)) } & { (S v (PvQ)) & (S v (PvW)) }

(R v P v Q) & (R v P v W) & (S v P v Q) & (S v P v W)

and substituting back we get (whew!)

(H(f2(x)) v –Hx v A(f1(x)))
& (H(f2(x)) v –Hx v –L(x,f1(x)))
& (L(f2(x),x) v -Hx v A(f1(x)))
& (L(f2(x),x) v –Hx v –L(x,f1(x)))

CNF at last!!!!

Note the instances of x above are universal, so the clauses (the parts between the &s) are independently true for all x, and thus we can use different variables for each of them if we like.

Now let's introduce the axiom Hc & -Exists x (Hx & Lxc) and the goal Exists x (Ax & -Lcx). These become (when the goal is negated) Hc & (-Hy v –Lyc), and -Az v Lcz, where I have changed to new variables.

So we end up with seven clauses (disjunctions of literals):

1. H(f2(x)) v –Hx v A(f1(x))
2. H(f2(x)) v –Hx v –L(x,f1(x))
3. L(f2(x),x) v -Hx v A(f1(x))
4. L(f2(x),x) v –Hx v –L(x,f1(x))
5. Hc
6. -Hy v –Lyc
7. -Az v Lcz

We start by unifying 5 with each of 1-4, to get 1'-4', substituting c for x in all. These then are

1'. H(f2(c)) v A(f1(c))
2'. H(f2(c)) v –L(c,f1(c))
3'. L(f2(c),c) v A(f1(c))
4'. L(f2(c),c) v –L(c,f1(c))

Now we try to unify 1' and 7, since one contains A and the other –A. Using z ←ßf1(c) we have
8. H(f2(c)) v L(c,f1(c))

Unify 8 with 2':
9. H(f2(c))

From 9 and 6 with y ← f2(c):
10. –L(f2(c),c)

From 10 and 3':
11. A(f1(c))

Now we unify 4' and 6, with y ← f2(c)
12. –H(f2(c)) v –L(c,f1(c))

From 12 and 9:
13. –L(c,f1(c))

From 13 and 7 with z ← f1(c):
14. -A(f1(c))

From 14 and 11 we get nil.

This was *not* obvious, at all. There were very many choices that could have been made for wffs to unify; I have simply shown a fairly quick path leading to nil, out of the very very messy tree of possible paths, most of which lead nowhere interesting.

Fortunately, there are computer programs that can do the search for us. This is a part of the field known as automated theorem-proving. But it probably still will be a very long time before mathematicians are put out of business! Only very, very few new theorems have been first proven by automated means.

Yet if we adopt a simplified language, restricting wffs to so-called Horn clauses, then the search is often very fast indeed. The PROLOG language is a rather popular one of this sort. A Horn clause is a disjunction of literals, where at most one is positive. Of course, the restriction also reduces the what one can conveniently express. In particular, Horn clauses do not include all the wffs we will need for the Monkey and Bananas Problem, to which we turn next.