Clearing up close_enough

A supplement to Section 3.3
of *Prolog Programming in Depth*

Michael A. Covington

2010 October 14, corrected October 19

When we do arithmetic in Prolog, we may want to get an answer, or we may want to verify an answer we already have:

```
?- X is 2+3.
X=5

?- 5 is 2+3.
yes
```

These correspond to things like `x=y+z` and `if (x == y+z)...` respectively in other programming languages.

The trouble is, floating-point calculations are not exact. Obviously,

```
?- 1.414 is sqrt(2).
no
```

is going to fail, because 1.414 is not *exactly* $\sqrt{2}$. But it's close.

Suppose we want a predicate like `=` except that it also succeeds if its arguments are two numbers that are close together (say, equal to within 0.001).

Let's name it `close_enough/2`. If all we care about is what happens when both its arguments are instantiated, then this will do:

```
% close_enough -- first version

close_enough(X,Y) :- abs(X-Y) < 0.001.
```

Recall that `<` evaluates arithmetic expressions on both the left and the right. As long as both `X` and `Y` are instantiated to numbers, this works.

But we might also want to use it to deliver an answer. That is:

```
?- ...something that instantiates X..., close_enough(X,Y).
```

where we don't know whether `Y` is already instantiated or not. If it is, we want to succeed if `X` is close enough to its value; if not, we want to put the value of `X` into it.

To deal with uninstantiated arguments, we need two clauses:

```
% close_enough -- second version

close_enough(X,X).

close_enough(X,Y) :- abs(X-Y) < 0.001.
```

Now, if either argument is uninstantiated or if the arguments match exactly, the first clause succeeds. Otherwise, if they are slightly different but still close, the second clause succeeds.

But now we have a different problem. It's not enough for the first clause to succeed when it needs to. We must also ensure that when the first clause succeeds, the second clause will not also succeed and will not try to do impossible operations.

The reason this is a concern is that something after `close_enough` in our query might force backtracking to occur. Here is a contrived example:

```
?- close_enough(2,2), write('got it'), nl, fail.
got it
got it
```

This tells us `close_enough(2,2)` is succeeding twice when we make it backtrack. We only want it to succeed once, to keep from doubling up other (as yet unforeseen) computations that might contain it.

What's worse, look at this:

```
?- close_enough(2,X), write('got it'), nl, fail.
got it
instantiation error
```

After using the first clause, it tries to use the second, but `<` won't work with an uninstantiated variable.

These are awkward examples — I can't give you realistic ones until we are constructing much more complicated Prolog programs. But the key idea is, if there aren't really two different solutions, we don't want a query to succeed more than once.

Fortunately, the cure is simple. (Simpler than shown in the book.) We can do this:

```
% close_enough -- third version

close_enough(X,X).

close_enough(X,Y) :- X \= Y, abs(X-Y) < 0.001.
```

Now the second clause does its computation only if `X` and `Y` are not unifiable. That is, it works under exactly the conditions where the first clause doesn't work. There is no situation in which both clauses will succeed.

There is another way. Notice that what we did, just now, was save a backtrack point to the second clause, and then use `\=` to keep the second clause from succeeding. We can use an operator called "cut" (written "`!`") to discard the backtrack point. This is not something you're supposed to understand in Chapter 3, but this is what it looks like:

```
% close_enough -- fourth version

close_enough(X,X) :- !.

close_enough(X,Y) :- abs(X-Y) < 0.001.
```

What the cut means is, "If you get here, don't backtrack; forget about the other clause."

With this in mind, you should be able to understand the solution on page 65, which was written with a Prolog compiler that doesn't have the `abs` function.