Second-Order Procedures

1 Motivation

At this point in time, we've seen a whole bunch of examples involving queries with multiple answers. For example, consider the following procedure definition:

```
1 foo(3).
2 foo(2).
3 foo(1).
```

With the above definition in mind, we can get multiple answers from the same query, like so:

```
?- foo(X).
X = 3 ;
X = 2 ;
X = 1.
```

As humans reading the above output, it's intuitive to think of these as a set of solutions. However, Prolog takes a decidedly different viewpoint. Instead, Prolog views these as three separate worlds, each yielding one solution. Most importantly, these worlds do not collide; for example, the world wherein `X = 1` is distinct from the world where `X = 2`.

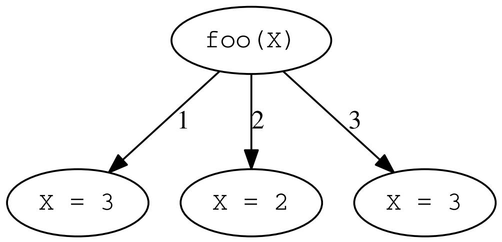
The distinction here is subtle, and up until this point it hasn't really made much of a difference However, a key query will reveal a limitation with Prolog's model, even for such a simple example as the one above. This query is shown below, in English:

What are all the answers to `foo(X)`, in ascending order according to `X`?

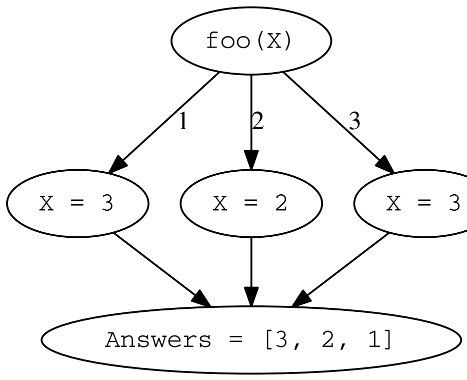
The above query may seem trivial at first: just take the answers we had before (namely, `X = 3; X = 2; X = 1`), and put them in ascending order (yielding `X = 1; X = 2; X = 3`). Sorting (with a library-provided `sort/2` procedure) is already available, so this should be easy.

However, in order to sort something, we need a list of the things to sort. The output of `foo(X)` is not a Prolog list, despite the fact that we can *think* of it as a list. As such, we don't actually have the necessary input list available. This might sound like a trivial problem (I mean, the list is *right there on the screen*), but there is, in fact, no way to get this list using only the Prolog we've seen so far.

There is a fundamental reason why we can't get such a list: each separate answer exists in a separate world, and so far none of the code we've seen crosses world boundaries. To better understand this problem, consider the graph below, which shows how different worlds "split off" from the original `foo(X)` query. Each edge is labeled with the line number of the corresponding code in the `foo` definition that leads to a split.



As shown above, each world is completely distinct. Additionally, while worlds can split off into further worlds, there has (so far) been no way for two worlds to rejoin later. We've forced the exploration of worlds to strictly follow a tree, as we did with solving SAT instances with semantic tableau. This tree-based approach breaks down when asking queries over multiple solutions, since this requires us to look at each of the leaves in the tree. Since these leaves live in distinct worlds, any analysis over leaves has been limited to what we can do as humans reading the output. Crucially, we cannot run Prolog code over all these leaves, since that would require us to join together these leaves somewhere to collect all the results in one place. With this sort of joining in mind, we'd need a graph that looks more like the following:



Once we have something like `Answers` above, we'd be able to perform the all-important `sort`, as with `sort(Answers, SortedAnswers)`. However, as of yet, we haven't seen anything that can perform this sort of join.

2 Enter Second-Order Procedures

This sort of join operation is precisely what *second-order procedures* do, which are built-in operations in Prolog. These operations are named *second-order* from the fact that they operate over multiple paths in the sort of tree structure above (performing a join), as opposed to working over just a single path. Equivalently, one can think of second-order procedures as working over sets of solutions instead of a single solution.

Without further ado, we will translate the original English query to Prolog using a second-order procedure, specifically `findall/3`. This translation is shown below:

```
?- findall(X, foo(X), Answers), sort(Answers, SortedAnswers).
Answers = [3, 2, 1],
SortedAnswers = [1, 2, 3].
```

As the name `findall` suggests, the `findall` second-order procedure finds all answers for which the given query is true. The above Prolog query can be read as follows: find all `X` (the first parameter) such that `foo(X)` is true (the second parameter), and put all the various satisfying values of `X` into the list `Answers`, ordered by when they were found. With this in mind, we get the key `Answers` list in the same order as we expected in the graph before.

2.1 Capturing Multiple Variables

While the above usage of `findall` was sufficient for getting all the answers from the `foo` procedure, we may want to capture more than one variable at a time. For example, consider the `bar` procedure below, which takes two parameters:

```
bar(a, b).
bar(c, d).
bar(e, f).
```

To capture both parameters, we'll need to provide a *template* to `findall` which holds the sort of pattern we want to capture for each satisfying answer. This is perhaps most easily shown with code, so an example is below:

```
?- findall(pair(A, B), bar(A, B), Answers).
Answers = [pair(a, b), pair(c, d), pair(e, f)].
```

As shown, the *template* goes into the first parameter of the call to `findall`. This particular template states that for each satisfying solution, a structure named `pair` will be collected (an arbitrary name), which holds variables `A` and `B`. As for how variables `A` and `B` get values, this can be seen in the second parameter specifying the query `bar(A, B)`. This can be read as calling `bar(A, B)`, and for each satisfying solution, a corresponding `pair(A, B)` will be collected. This `pair` effectively saves the result of the query to `bar`, and these saved answers are collected as a list in `Answers`.

Note that this use of a template isn't a special case; a variable alone is another valid template. With that in mind, the use of `X` in the prior example involving `findall` and `foo` was also using a template, it's just that the template was merely `X`.

It is not necessary for a template to capture all variables involved; you only need to capture the portions you care about. For example, let's say we were interested in only the first parameter to `bar/2`. With this in mind, we can write a `findall` template that captures the first parameter, shown below:

```
?- findall(X, bar(X, _), Answers).
Answers = [a, c, e].
```

As shown, since `X` occupied the position of the first parameter in the call to `bar/2`, `Answers` only considers the first parameter in its answers.

2.2 Enter `bagof/3` and Existential Quantification

The sort of join that is being performed can have certain nuances, particularly when multiple variables are involved. As such, there exist other second-order procedures outside of `findall/3`, which all behave in slightly different ways. The first of these we will look at is that of `bagof/3`.

The `bagof/3` procedure takes the same three parameters as `findall/3`, namely a template, a query, and a list of answers. However, `bagof/3` behaves somewhat differently with these components. The first difference we will look at is what happens for queries which produce no answers and fail. The `findall/3` procedure will produce an empty list of answers in such a case, whereas the `bagof/3` procedure itself will fail. This is illustrated in the queries below involving the built-in `fail` procedure, which always fails:

```
?- findall(X, fail, Answers).
Answers = [].

?- bagof(X, fail, Answers).
false.
```

To see further examples between `findall/3` and `bagof/3`, we'll introduce a new toy procedure `baz/3`, defined below:

```
baz(a, a, c).
baz(a, b, c).
baz(a, c, c).
baz(b, c, d).
```

Now, let's issue some queries involving `findall/3`, `bagof/3`, and `baz/3`. First, let's say we're interested in the last parameter to `baz/3`. We can collect this last parameter with `findall/3`, shown below:

```
?- findall(X, baz(_, _, X), Answers).
Answers = [c, c, c, d].
```

As shown, the answer `c` appears three times, because there are three separate answers with `c`. Other than that, the above output should be unsurprising. However, the same query with `bagof/3` yields something entirely different:

```
?- bagof(X, baz(_, _, X), Answers).
Answers = [c] ;
Answers = [c] ;
Answers = [c] ;
Answers = [d].
```

With the semicolon (`;`), we had to ask `bagof/3` for additional solutions. That is, `bagof/3` didn't actually put all the solutions together.

The reason this happened is because `bagof/3` treats variables in the query different from variables in `findall/3`. Specifically, `findall/3` treats all variables in the query as being *existentially quantified*, whereas `bagof/3` does **not** treat such variables as being existentially quantified. (Oh, is that all!)

As for what "existential quantification" means in this context, this has to do with what variables are given values when. With first-order procedures (everything we've seen before this handout), variables getting values was determined solely by unification. Once a variable received a value, this value was set in stone; variables are immutable. However, when second-order procedures come into play, this suddently gets tricky since we're dealing with queries which operate over multiple leaves in the sort of graph shown before. Phrased another way, what happens to variables in between different solutions to the same query? There are two general ways to view a variable in this context:

1. In between different solutions to the query, variables effectively "reset". That is, a variable can hold a particular value only while processing a *single* solution; when we go to compute the next solution, the variable is refreshed, as if it never recieved a value. We refer to such variables as being existentially quantified.

2. Variables maintain values across solutions as if they were part of the computation to form a single solution. That is, once a variable gets a value, it keeps this value. This describes the normal execution rules of Prolog. Such variables, in contrast to existentially quantified variables, are *universally quantified*: they apply over **all** solutions, as opposed to just a single solution.

Here is where the difference between `findall/3` and `bagof/3` lies. With `findall/3`, variables are always existentially quantified. To illustrate this, let's again consider the query involving `findall/3` above:

```
?- findall(X, baz(_, _, X), Answers).
```

Recall that each use of underscore (\) refers to a different variable, so we can rewrite this query to the following:

```
?- findall(X, baz(A, B, X), Answers).
Answers = [c, c, c, d].
```

With `findall/3`, the variables `A`, `B`, and `X` are all existentially quantified, meaning they can vary between the computation of different solutions to `baz`. As such, there is no requirement that the `A` for one solution be the same as an `A` from another solution. The same cannot be said for the same query using `bagof/3`, shown below with explicit `A` and `B` variables instead of underscore (\):

```
?- bagof(X, baz(A, B, X), Answers).
A = B, B = a,
Answers = [c] ;
A = a,
B = b,
Answers = [c] ;
A = a,
B = c,
Answers = [c] ;
A = b,
B = c,
Answers = [d].
```

As shown above, `A` and `B` remain fixed for particular `X` answers. This is why `bagof/3` gives multiple solutions: it can end up leading to unifications of variables in the query which survive past the `bagof/3` call. This is not true of `findall/3`, which will not maintain variable bindings beyond the call.

With `bagof/3`, we can explicitly request that a variable be existentially quantified by using the carat (`^`) operator. Such usage is illustrated below, which runs the same query as above but asks that variables `A` and `B` be existentially quantified:

```
?- bagof(X, A^B^baz(A, B, X), Answers).
Answers = [c, c, c, d].
```

The above query ends up giving us the same result as the `findall/3` query, as now variables `A` and `B` are existentially quantified. The only difference here is that `findall/3` performed this existential quantification automatically, whereas here with `bagof/3` we explicitly had to specify it.

The real power in `bagof/3` is that we can selectively existentially quantify variables. To illustrate this, consdier the query below:

```
?- bagof(X, A^baz(A, B, X), Answers).
B = a,
Answers = [c] ;
B = b,
Answers = [c] ;
B = c,
Answers = [c, d].
```

The above query specifies that `A` must be existentially quantified, but not `B`. This ends up merging the last two answers together, as:

- The first parameter is allowed to vary between different solutions, so the last two `baz/3` facts are compatible despite the fact that one starts with `a` and the other starts with `b`.
- The second parameter must be the same between the different solutions, which in this case is ok since the last two `baz/3` facts have `c` as the second parameter
- As for the third parameter, this is implicitly existentially quantified because `X` appears in the template (which is simply `X` in this example), allowing for different answers to appear here (namely `c` and `d`). If the variables in the template weren't implicitly existentially quantified, `bagof/3` wouldn't be very useful, as it would force all answers to be the same.

As another example, consider the modified query below, which existentially quantifies `B` instead of `A`:

```
?- bagof(X, B^baz(A, B, X), Answers).
A = a,
Answers = [c, c, c] ;
A = b,
Answers = [d].
```

The above query ends up merging together the first three facts into one set of answers (`[c, c, c]`), as the existential quantification of `B` allows the second parameter of `baz/3` to vary between the answers. This overcomes the fact that the first three facts have incompatible second parameters (namely, `a`, `b`, and `c`). Since `A` is not existentially quantified, the first parameter must be the same for all these answers, and is set to `a` for the first batch of answers. For the second batch of answers, there is only the last `baz/3` fact in play left, namely `baz(b, c, d)`. The answer of `A = b, Answers = [d]` corresponds to this fact, where the second parameter (`c`) is ignored, again because of the existential quantification of `B`.

2.3 Getting Unique Answers with `setof/3`

The previous queries involving `findall/3` and `bagof/3` occasionally involved duplicate answers. For example, consider the query below, which duplicates the answer `c` three times:

```
?- findall(X, baz(_, _, X), Answers).
Answers = [c, c, c, d].
```

If duplicate answers are not desired, then we can use the `setof/3` second-order procedure instead of the `findall/3` or `bagof/3` procedures. The `setof/3` procedure works *exactly* like `bagof/3` (including the bits about optional existential quantification of variables), **except** the list of answers is passed through `sort/2` before being returned. This has the effect of both sorting the answers, and removing duplicate answers (yes, `sort/2` removes duplicates in addition to sorting, whether you want it to or not).

With all this in mind, we can rewrite the above query using `setof/3`, shown below:

```
?- setof(X, A^B^baz(A, B, X), Answers).
Answers = [c, d].
```

As shown, no more duplicate answers are present, so there is only one `c` instead of three.

Going *way* back to the original problem in the motivation, `setof/3` can be used to quite succintly solve it, like so:

```
?- setof(X, foo(X), Answers).
Answers = [1, 2, 3].
```

This saves the need of passing `Answers` through `sort/2`, since `setof/3` internally handles the sorting for us.

3 Use Case: A Metainterpreter which Randomly Orders Nondeterministic Calls

We end this discussion with a use case harking back to an important discussion egarding making the problem space explorable for test case generation. An example test case generator is shown below, which suffers from the sort of problems we've discussed with infinitely large search spaces:

```
exp(integer(1)).
exp(plus(E1, E2)) :-
    exp(E1),
    exp(E2).
```

As shown, `exp/1` is infinitely recursive when used as a test case generator, leading to issues where the rightmost operand in `plus` (namely `E2`) will be "spammed" at the cost of the leftmost operand (`E1`).

During class, one of the proposed solutions was to somehow change Prolog so that it executes nondeterministic calls in a random order, instead of always trying them in the same order as presented in the file. This is a **great** idea, as it both solves the original problem, and leads to a way of performing randomized test case generation. However, at the time, we didn't have enough Prolog knowledge under our belts to be able to do this.

Now that we have metainterpreter skills and second-order procedures, we can tackle this problem. Since we're only going to change how calls work, the rules handling `true`, conjunction, and disjunction are all standard:

```
interpret(true) :- !.
interpret((A, B)) :-
    !,
    interpret(A),
    interpret(B).
interpret((A; B)) :-
    !,
    (interpret(A); interpret(B)).
```

In constrast, the rule for calls is far more interesting. Intuitively, the call rule needs to do the following, in order (written somewhat specifically to how we will approach this problem):

1. Determine what different alternatives are available for calling, yielding a list of alternatives.
2. Put this list in a random order.
3. Nondeterministically select an alternative from this list, and move foreward executing the chosen alternative. Since the choice is nondeterministic, all alternatives will be explored. Since these choices were put in a random order by the previous step, this effectively performs exploration of alternatives in a random order.

With the first step above, this sounds a lot like what `clause/2` does, but now the alternatives are put into a list as opposed to being nondeterministically chosen. Here we can use a second-order procedure to collect all the possibilities returned by `clause/2` into a single list. We start doing this below:

```
interpret(Call) :-
    findall(pair(Call, Body), clause(Call, Body), Pairs),
    ...
```

The above snippet will collect `Call`/`Body` pairs into `Pairs`, where each `Call`/`Body` pair is derived from a call to `clause/2`. The collection of the different bodies to execute (`Body`) should be straightforward: these are alternative code bodies to execute moving forward. However, the collection of `Call` is pretty subtle, especially considering we already seemingly have access to `Call`. The reason why `Call` is included here has to do with existential variable quantification. Each different alternative is permitted to unify with `Call` differently, resulting in a different variant of `Call` depending on which body is chosen. By putting `Call` directly in the template, we not only save which `Call` variant applies to the given body, but we also allow for each `clause/2` solution to unify with `Call` differently. This is because the variables in `Call` will be existentially quantified, due to its presence in the template (with `pair(Call, Body)`).

An example using `foo(X)` is shown below, which illustrates this problem. Let's say we just collected the body, like so:

```
?- Call = foo(X), findall(Body, clause(Call, Body), Bodies).
Call = foo(X),
Bodies = [true, true, true].
```

As shown, while the unification of the clause head (the `foo(X)` part) *does* occur internally in `clause/2`, the result is effectively discarded as nothing in the template saves the call (`Call`). In constrast, this information is retained in the following modified query, which matches up with what the previous code snippet with `findall/3` does:

```
?- Call = foo(X), findall(pair(Call, Body), clause(Call, Body), Pairs).
Call = foo(X),
Pairs = [pair(foo(3), true), pair(foo(2), true), pair(foo(1), true)].
```

As shown above, each element in `Pairs` now holds what the call itself unifies to in its given answer (the first parameter to `pair`), along with the remaining code to execute (the second parameter to `pair`). Note that the original call is unchanged, thanks to existential quantification (that is, `Call = foo(X)`, as in the original query).

For the second step wherein the list of alternatives is put in random order, we can conveniently use the `random_permutation/2` procedure, available in SWI-PL's libraries. The `random_permutation/2` procedure will randomly reorder an input list (the first parameter), resulting in an output reordered list (the second parameter). This reordering is done deterministically; only a single random reordering is chosen, even though many may be available in general.

For the third step, we can nondeterministically select with SWI-PL's `member/2` procedure, and finally move forward with execution via a recursive call to the `interpret/1` procedure we're defining.

The complete code implementing calls is shown below:

```
1  interpret(Call) :-
2      findall(pair(Call, Body), clause(Call, Body), Bodies),
3      random_permutation(Bodies, RandomBodies),
4      member(pair(Call, Body), RandomBodies),
5      interpret(Body).
```

Of special note in the above code snippet is line 4, which does the following succintly:

- Nondeterministically select a pair from `RandomBodies`.
- Unify the call in the chosen pair with the preexisting variable `Call`, effectively performing unification with the clause head.
- Unify the body in the chosen pair with `Body`, getting the remainder to interpret. Note that while variable `Body` was previously syntactically present in line 2, since `Body` was in the template to `findall/3` in that case, it was existentially quantified. As such, before line 4, `Body` did not have any known value. This is unlike `Call`, which was a parameter to `interpret/1` on line 1.

The complete code for this metainterpreter is shown below for convenience:

```
interpret(true) :- !.
interpret((A, B)) :-
    !,
    interpret(A),
    interpret(B).
interpret((A; B)) :-
    !,
    (interpret(A); interpret(B)).
interpret(Call) :-
    findall(pair(Call, Body), clause(Call, Body), Bodies),
    random_permutation(Bodies, RandomBodies),
    member(pair(Call, Body), RandomBodies),
    interpret(Body).
```