Programming in Tabled Prolog Well-Founded Semantics (very) DRAFT [1]

David S. Warren

Department of Computer Science
SUNY @ Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

June 28, 2012

[1]This is a very early draft made available privately for those who might find it of interest. I reserve all rights to this work. -dsw

Chapter 1

Negation in XSB

1.0.1 Well-Founded Semantics

Another approach to defining the meaning of logic programs with default negation is the Well-Founded Semantics, which uses a 3-valued logic [?]. In a 3-valued logic a proposition may be true or false or *undefined*. By using the truth value undefined judiciously, we can provide a single 3-valued model for each logic program with negation.

We will motivate the definition of the well-founded semantics by first considering what we can infer if we have only partial knowledge of some predicates that are used, but not defined, in an old-fashioned definite logic program.

Consider a situation in which we have some (definite) rules that define some predicates inductively, and use other predicates in their definitions, but don't define them. We will call the used, but not defined, predicates as open predicates. We will assume that we have partial knowledge of the truth values of atoms of the open predicates. As a simple, concrete example, consider our friend *reachability* in a graph:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Here `reach/2` is defined using the `edge/2` predicate, but `edge/2` is not defined. We can say that the `reach` definition is parameterized by the open predicate `edge/2`; a definition of the `edge/2` predicate determines a complete definition of the `reach/2` predicate.

But perhaps we have only partial knowledge of the `edge/2` relation: say we know that there is an edge from `a` to `b`, an edge from `b` to `a`, no edge from `a` to `c` and no edge from `b` to `c`, but we don't know about other possible edges. (We'll assume the graph has only these three nodes.) The question is: What can we conclude about the `reach/2` relation using this incomplete information about the open predicate `edge/2`? Intuitively, we know that `b` is reachable from `a`, `a` is reachable from `b`, `a` is reachable from `a`, `b` is reachable from `b`, `c` is not reachable from `a`, and `c` is not reachable from `b`. But we don't know if `a` is reachable from `c`, if `b` is reachable from `c`, or if `c` is reachable from `c`. Let's be more precise (and general) concerning how we might come to such conclusions.

We assume that the known true `edge` facts are T = {`edge(a,b)`, `edge(b,a)`} and the known false `edge` atoms are F = {`edge(a,c)`, `edge(b,c)`}. How do we conclude what `reach` atoms can be known true and what `reach` atoms can be known false? To find ones that must be true, we can just add the known facts to the (definite) program and find its least model. Anything true in that model must be true even if some (or all) of the unknown open atoms would turn out to be false. For our example, we get the following program:

```
edge(a,b).
edge(b,a).
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Now we see what is implied by this set of rules: here we get {`reach(a,b)`, `reach(b,a)`, `reach(a,a)`, `reach(b,b)`} by taking the least fixed point of this program. These are facts that must be true in any model consistent with the partial knowledge we have of the `edge/2` relation. We have made conservative assumptions and what is still true under these most conservative assumptions must indeed be true. We can say that this program defines the "definitely true" atoms for `reach`.

We have just seen how to conclude what `reach` atoms must be true; now how do we determine what atoms must be false? We similarly create a program using our assumptions of what `edge` atoms are true and false, but this time we assume that all `edge` atoms *not* known to be false are actually true. So the program we get is:

```
edge(a,b).
edge(b,a).
edge(c,b).
edge(c,a).
edge(a,a).
edge(b,b).
edge(c,c).
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

The seven `edge` facts are those not known to be false, i.e. those that might be true. We use this program, taking the least model, to see what `reach` atoms "might be true". We can say this program defines the "possibly true" atoms. In this case, we get: {`reach(a,b)`, `reach(b,a)`, `reach(c,b)`, `reach(c,a)`, `reach(a,a)`, `reach(b,b)`, `reach(c,c)`}. Any `reach` atom *not* in this set will have to be false in any model consistent with our initial `edge` assumptions. So we conclude that any `reach` atom in the complement of this set must be false in any consistent model. So we know that `reach(b,c)` and `reach(a,c)` must be false. We have made liberal assumptions about what might be true and anything still not true under these most liberal assumptions must indeed be false.

To review, we have described a way to use partial or incomplete knowledge of facts that are used in the definitions of inductively defined predicates to infer information about those predicates, information including what instances must definitely be true and what are possibly true (and whose complement must definitely be false.) The idea for determining the definitely true defined instances is to assume that all defining facts not known true are false. Then anything that must be true in this situation is definitely true. To determine the definitely false atoms, we assume all defining atoms not known to be false are indeed true; anything that still isn't possibly true under these assumptions must definitely be false.

We can use this idea to give a semantics to programs with negative literals in their bodies; i.e., inductively defined predicates that use negations in their definitions. For simplicity we will consider only propositional programs here. In order to use a negative literal to help infer a fact, we must know that the literal is true, i.e., the atom that is negated is false. But how can we determine such things? We will use the idea we have just developed of reasoning with partial information about open predicates to approach this problem. We start by initially disconnecting the negative literals in the bodies of clauses from their positive forms and just thinking of them as new propositions. E.g., for the literal `tnot(p)`[1], we introduce a new proposition symbol, say `neg_p`. We then replace all negative literals in the program with their new positive forms to get a purely positive program with open predicates. The resulting program can be seen as similar to the old program but now parameterized by the newly introduced `neg_*` predicates.

So let's apply our approach to reasoning about definitions with partial knowledge about open predicates. We begin by assuming that we know nothing about these new open predicates, i.e., none of their atoms are known true or false. So to determine what defined predicates must definitely be true, we interpret all the `neg_*` predicates as false, and see what is true in the resulting least model. Those defined atoms are now known true. And to find the false atoms, we interpret all the open predicates as true, take the least model to see what could conceivably be true, and then take its complement to find the definitely false atoms.

Let's consider an example; the initial general program with negation is:

```
p :- r, tnot(t).
q :- r, tnot(s), tnot(u).
r :- s.
r.
s :- tnot(q), r.
t.
```

We transform it to its open form:

```
p :- r, neg_t.
```

[1]Recall that tnot is the tabled negation operator in XSB.

```
q :- r, neg_s, neg_u.
r :- s.
r.
s :- neg_q, r.
t.
```

To determine the well-founded model of the original program, we will maintain two programs with the open propositions: one that tells us what atoms must be true (called the *definitely true* program), and a program that tells us what atoms could possibly be true (called the *possibly true* program) and therefore tells us what atoms must be false. Each program will be a conservative approximation and we will iteratively modify the programs to improve their accuracy.

The initial definitely true program and definitely false program are:

```
%  definitely true                  %  possibly true

p :- r, neg_t.                      p :- r, neg_t.
q :- r, neg_s, neg_u.               q :- r, neg_s, neg_u.
r :- s.                             r :- s.
r.                                  r.
s :- neg_q, r.                      s :- neg_q, r.
t.                                  t.
                                    neg_p.  neg_q.  neg_r. neg_s.  neg_t.  neg_u.
```

In the definitely true program we assume that none of the `neg_*` propositions are true; in the possibly true program we assume they are all true. If we take the least model of the definitely true program, we get {`r`, `t`}. These propositions will true regardless of the truth values of the `neg_*` propositions, so we will want them to be true in the well-founded model of the original program. And the least model of the possibly true program (on the defined propositions) is {`p`, `q`, `r`, `s`}. So any defined proposition symbol *not* in this set must be false, and we will want it false in the well-founded model.

Now we can try to use each of these programs to improve the accuracy of the other. There really is a connection between the pair of propositions, say `p` and `neg_p`: if one is true, then the other should be false. So if we deduce that `r` is definitely true in the well-founded model, then we know that `neg_r` is definitely false, i.e., not possibly true. And that allows us to update our possibly true program by deleting the fact for `neg_r`. Similarly, if we know that `u` is definitely false then `neg_u` is definitely true, and we can update our definitely true program by adding `neg_u` to it. Having changed our definitely true and possibly true programs, we can again find their least fixed points and see if we have learned something new that will allow us to further update the programs. We continue to add `neg_*` atoms to the definitely true program and remove `neg_*` atoms from the possibly true program in this way until we learn nothing new. The resulting programs define the well-founded semantics of the original program.

Consider this process for our example; the least model of the true program contains r and t, so these are known true. Since they are known true, `neg_r` and `neg_t` must be false, so we can remove them from the possibly true program, improving our estimate of the possibly true atoms and getting an updated possibly true program:

```
%  definitely true                 % possibly true

p :- r, neg_t.                     p :- r, neg_t.
q :- r, neg_s, neg_u.              q :- r, neg_s, neg_u.
r :- s.                            r :- s.
r.                                 r.
s :- neg_q, r.                     s :- neg_q, r.
t.                                 t.
                                   neg_p.  neg_q.  neg_s.  neg_u.
```

Now looking at the least model of this new possibly true program, we see that neither u nor p is possibly true, i.e., not in the least model of the possibly true program and thus must be false. So we can improve our estimate of the true atoms by adding `neg_u` and `neg_p` to our definitely true program, obtaining:

```
%  definitely true                 % possibly true

p :- r, neg_t.                     p :- r, neg_t.
q :- r, neg_s, neg_u.              q :- r, neg_s, neg_u.
r :- s.                            r :- s.
r.                                 r.
s :- neg_q, r.                     s :- neg_q, r.
t.                                 t.
neg_p. neg_u.                      neg_p.  neg_q.  neg_s.  neg_u.
```

Now looking at the current versions of the two programs: the neg version of every defined atom in the least model of the definitely true program has been removed from the possibly true program; and the neg version of every defined atom not in the least model of the possibly true program has been added to the definitely true program. So we can no longer improve our estimates of the definitely true and possibly true atoms, which leaves us with the well-founded model of the original program. The atoms true in the well-founded model are the defined atoms in the least model of the final definitely true program; the atoms false in the well-founded model are the defined atoms *not* in the least model of the final possibly true program. Thus for this program r and t are true in the well-founded model, p and u are false, and s and q are undefined.

To review this process: the definitely true program starts with no `neg_*` atoms and gains them as their positive counterparts are found not to be possibly true. The possibly true program starts with all the `neg_*` atoms and loses them as their positive counterparts are found to be definitely true. The process continues until no more improvements can be made. Then we read off the well-founded model from the final programs.