Knowledge Representation in XSB, Flora and Silk

Terrance Swift

April 23, 2012
Part 1: Operational Semantics, Complexity and Termination

1. Knowledge Representation, Complexity and Prolog
2. Tabling for Definite Programs
3. Tabling and Negation
4. Termination
5. Defeasible Reasoning
6. Summary
Knowledge Representation: Provisional Definition

Knowledge Representation and Reasoning (KRR) is an area of artificial intelligence whose fundamental goals are

- to be able to represent knowledge about a broad complex domain in a manner that can be understood, created, and maintained by non-programmers (called Subject Matter Experts (SMEs) with the collaboration of knowledge engineers.

- to be able to use this knowledge in a manner similar to that of a trained expert.

This is just my definition, but I didn’t see a better one on the web.
KRR Examples

- One of the best-known examples is the Watson project, sponsored by IBM, which answers Jeopardy questions.
- Another is the Cyc system, which “… is an artificial intelligence project that attempts to assemble a comprehensive ontology and knowledge base of everyday common sense knowledge” (http://en.wikipedia.org/wiki/Cyc)
- Whether a program to play chess would be a KRR example depends on how you interpret the previous definition.
- Another example is the Digital Aristotle project sponsored by Vulcan, Inc (http://www.projecthalo.com), which seeks to train a computer to be able to use scientific knowledge at a level of US college Advanced Placement exams (similar to a baccalaureat degree).
Example: U.S. College Advanced Placement Exams

Regarding mitosis and cytokinesis, one difference between higher plants and animals is that in plants

- (a) the spindles contain cellulose microfibrils in addition to microtubules, whereas animal spindles do not contain microfibrils
- (b) sister chromatids are identical, whereas in animals they differ from one another
- (c) a cell plate begins to form at telophase, whereas in animals a cleavage furrow is initiated at that stage
- (d) chromosomes become attached to the spindle at prophase, whereas in animals chromosomes do not become attached until anaphase
- (e) spindle poles contain centrioles, whereas spindle poles in animals do not
Which of the following offers the best description of neural transmission across a mammalian synaptic gap?

- (a) Neural impulses involve the flow of K⁺ and Na⁺ across the gap.
- (b) Neural impulses travel across the gap as electrical currents.
- (c) Neural impulses cause the release of chemicals that diffuse across the gap.
- (d) Neural impulses travel across the gap in both directions.
- (e) The calcium within the axons and dendrites of nerves adjacent to a synapse acts as the neurotransmitter.
Example: U.S. College Advanced Placement Exams

- These are two sample questions. A KRR system should be able to answer other similar questions, that it hasn’t seen.
- Question answering is also used in the Watson KRR system.
- The AP task requires a KRR system that is less broad but deeper than answering Jeopardy questions
  - AP answering for biology requires representation and reasoning about processes, temporal events, and spatial relations, in addition to general logical and ontological reasoning.
- As with Jeopardy the AP domain requires
  - Weighing alternatives about the “best” answer (though neither requires explicitly probabilistic reasoning).
  - A natural language interface
- Both of these systems employ reasoning that has low computational complexity, rather than highly expressive theorem proving systems.
A very small fragment from a large rule set to support process reasoning (FLORA-2 syntax).

// a transitive relation from a given 1st to a 3rd holds
// if it holds from the 1st to the 2nd and from 2nd to the 3rd.
?r(?a, ?c) :- allow(transitive) and transitive(?r)
  and %silkb#isGround(?a)@builtin and %silkb#isGround(?b)@builtin
  and ?a != ?c and ?r(?a, ?b) and ?r(?b, ?c) .

// a transitive relation from a given 1st to a 3rd holds
// if it holds from the 1st to the 2nd and from 2nd to the 3rd.
?r(?a, ?c) :- allow(transitive) and transitive(?r)
  and %silkb#isGround(?a)@builtin and not %silkb#isGround(?b)@builtin
  and ?r(?a, ?b) and ?r(?b, ?c) and ?a != ?c .

// a transitive relation from a 1st to a given 3rd holds
// if it holds from a 2nd to the 3rd and from the the 1st to the 2nd”’].
?r(?a, ?c) :- allow(transitive) and transitive(?r)
  and not %silkb#isGround(?a)@builtin and %silkb#isGround(?c)@builtin
  and ?r(?b, ?c) and ?r(?a, ?b) and ?a != ?c .
Some Biology Axioms that use the Process Ontology

- `cyc:SitTypeFollowsSitTypeInSitType(cellCycle,mitosis,interphase)`
- `after(ends,begins)(in(cell(cycle)))(mitosis,interphase)`
- `immediate(part(of))(mitosis,phase(M))`
- `exclusively(part(of))(phase(M),cell(cycle));`
Many KRR systems are quite declarative but have high computational complexity.

First-order logic is undecidable.

Description logics are tractable subsets of first-order logic. Description logics based on $\mathcal{ALC}$ or its extension are $PSPACE$-complete, or $det-EXPTIME$-complete (e.g. $\mathcal{ALCQ}$).

Answer-set programming is NP-complete (credulous, a query is true in a model), or co-NP complete (skeptical – a query is true in all models).

Well-founded semantics is $O(size(P) \times atoms(P))$ for a ground program $P$. In practice, usually it acts as $O(size(P))$. 
High-complexity approaches often are not suitable for large-scale applications, e.g., semantic web, enterprise KR. So, rather than starting with high-complexity approaches, we start with a low complexity approach (WFS) and extend it. Extensions include:

- CLP(R), CLP(Q), CLP(FD).
- Low-complexity description logics: $\mathcal{EL}^+$ [BBL05], Description Logic Programs [GHVD03]
- Probability inference (to be discussed in another lecture)
Prolog as a KRR tool

“What more do you need?”

- Horn clauses have both a procedural and a logical semantics
- Allow easy creation of recursive types
- Efficient, robust, and scalable implementation
- Dynamic code allows changeable or self-modifying code.
- Constraints allow numeric or combinatorial reasoning
“Are you crazy?”

- Negation in Prolog is often useful but has a weak semantics
  - Completion semantics cannot be extended to allow positive recursion
  - The Fitting fixed point semantics [Fit85] is still weak (it keeps the truth value of many atoms as undefined).
- Prolog doesn’t terminate when it should, and programs can suffer from poor complexity.
- Prolog isn’t “smart” about its search unlike e.g., Cyc
- Programming with constraints is powerful, but can be difficult.
Tabling as a KRR tool

Claim: *Tabling fundamentally changes Prolog so that it is suitable as a KRR tool*

- You should be skeptical of this claim until it is substantiated.
- Tabling supports the Well-founded Semantics (WFS) which can be used as a basis of defeasible reasoning
- Tabling with *answer subsumption* forms a basis for quantitative and multi-valued logics
- Tabling with call abstraction can give powerful termination properties
- Tabling works well with dynamic code and tables can be incrementally updated
- Tabling extends rather than replaces Prolog semantics, including constraints
- These computational techniques support new KRR languages such as FLORA-2, Silk, and PITA.
reach(1,X), edge(X,Y)

edge(1,Y)

reach(1,X), edge(X,Z), edge(X,Y)
An Example of SLD Evaluation

The previous slide highlights some problems with SLD evaluation

- The SLD tree contains all solutions (completeness) but is infinite (non-termination)
- If you simply check for loops, you can achieve termination, but you don’t address complexity, which is also a problem with SLD.
An Example of SLG (Tabled) Evaluation

0. \texttt{reach(1,Y):- reach(1,Y)}

1. \texttt{reach(1,Y):- reach(1,X),edge(X,Y)}

2. \texttt{reach(1,Y):- edge(1,Y)}

3. \texttt{reach(1,2):-}

4. \texttt{reach(1,Y):- edge(2,Y)}

5. \texttt{reach(1,3):-}

6. \texttt{reach(1,Y):- edge(3,Y)}
Definite Programs: SLD vs. Tabling

- Tabling factors out redundant subcomputations (here \( r(1, Y) \)) to compute the minimal model of a definite program (expressible as the least fixed point of the \( T_P \) monotonic operator).
- Nodes have the form \( \text{fail} \) or

\[
\text{AnswerTemplate} :- \text{Delays} \mid \text{GoalList}
\]

\( \text{Delays} \) is only needed for negation, so we’ll ignore that for now.
- If a tree \( T \) has root \( A :- |A \), then \( A \) is termed the root subgoal of \( T \).
- In this formulation, answers are (non-failure) leaf nodes whose goal list is empty.
- Examples in these slides use Prolog’s top to bottom clause selection strategy and left-to-right literal selection strategy. These orderings are among many that are possible.
Definite Programs: Tabling Operations

From [Swi99], based on [CW96].

1 **New Subgoal:** Let $F_n$ be a forest that contains a non-root node
   \[ N = D \mid \text{Ans} :- G, \text{Goal List} \]
   where $G$ is the selected literal $S$ or $\text{not } S$. Assume $F_n$ contains no tree with root subgoal $S$. Then add the tree $S :- |S$ to $F_n$.

2 **Program Clause Resolution:** Let $F_n$ contain a root node $N = S :- |S$ and $C$ be a program clause $\text{Head} :- \text{Body}$ such that $\text{Head}$ unifies with $S$ with mgu $\theta$. Assume that in $F_n$, $N$ does not have a child $N_{child} = (S :- |\text{Body}) \theta$. Then add $N_{child}$ as a child of $N$.

3 **Positive Return:** Let $F_n$ contain a non-root node $N$ whose selected literal $S$ is positive. Let $\text{Ans}$ be an answer node for $S$ in $F_n$ and $N_{child}$ be the SLG resolvent of $N$ and $\text{Ans}$ on $S$. Assume that in $F_n$, $N$ does not have a child $N_{child}$. Then add $N_{child}$ as a child of $N$.

4 **Completion:** Given a *completely evaluated* set $S$ of subgoals where the tree for no subgoal $S \in S$ is marked completed, mark the trees for all subgoals in $S$ as completed.

---

1 For definite programs, simple resolution is all that is needed.
What does *Completely Evaluated* mean?

- A subgoal is completely evaluated iff it has all of its possible answers.
- A subgoal \( S \) is completely evaluated when all possible operations have been done on its nodes, and the nodes of trees upon which \( S \) depends.
- A ground subgoal is completely evaluated when an answer is derived for it.

*Incremental Completion* is necessary for efficient evaluation of programs.

We also haven’t yet defined \( | \) or *SLG answer resolution* – these will be defined when we introduce non-stratified negation.

- For now, ignore \( | \) and think of *SLG answer resolution* as the resolution of a fact against the leftmost argument of a goal.
But what does \textit{depends} mean, exactly?

\textbf{Definition (Subgoal Dependency Graph)}

Let $\mathcal{F}$ be a forest in a \textit{SLG} evaluation. A tabled subgoal $S_1$ \textit{directly depends on} a tabled subgoal $S_2$ in $\mathcal{F}$ iff neither the tree for $S_1$ nor that for $S_2$ is marked as complete and $S_2$ is the selected literal of some node in the tree for $S_1$.

The \textit{Subgoal Dependency Graph of $\mathcal{F}$}, $\text{SDG}(\mathcal{F})$, is a directed graph $V,E$ in which $V$ is the set of root goals for non-completed trees in $\mathcal{F}$ and $(S_i, S_j) \in E$ iff $S_i$ directly depends on $S_j$. 
Definite Programs: Example

:- table p/2.
p(X,Y) :- p(X,Z), p(Z,Y).
a(1,2).  a(1,3).  a(2,3).  
p(X,Y) :- a(X,Y).
Definite Programs: An SDG

\[ p(1, Z) \]
\[ \rightarrow \]
\[ p(2, Z) \]
\[ \rightarrow \]
\[ p(3, Z) \]
Let’s assume we have an atomic query $Q$. We start with a forest consisting of the single tree $Q :- \lnot Q$. From then on we apply applicable SLG operations until no more operations apply. This is an SLG evaluation.

Theoretically, SLG evaluations may have an infinite number of steps. But we won’t worry here about which infinite ordinal an evaluation stops at: termination means finite termination in these slides.

The ordering of applicable operations is called a *scheduling strategy*.

XSB mostly uses *local scheduling* (cf. [MS08]) which is illustrated by the forest on Slide 22. That’s what we use here.

The definition follows, but you can think of local evaluation as depth-first search for tabling. The evaluation explores SCCs in the search space in a depth-first manner.
Definite Programs: Local Evaluation

How should the operations \textbf{New Answer}, \textbf{Program Clause Resolution}, \textbf{Positive Return}, and \textbf{Completion} – be scheduled?

- \textbf{Completion} should be performed incrementally to save space.
- \textbf{Program Clause Resolution} should be scheduled as in Prolog, as far as possible.
- \textbf{New Subgoal} should be performed as soon as a goal is encountered, to accord with Prolog’s search strategy.
- This leaves the question of how to schedule \textbf{Positive Return} \textit{w.r.t} \textbf{Program Clause Resolution}.

Definition (Locality property)

Let $\mathcal{F}$ be an SLG forest. Resolution of an answer $A$ against a consuming node $N$ occurs \textit{in an independent SCC of} $\mathcal{F}$ if the root subgoal for $N$ is in an independent SCC in $SDG(\mathcal{F})$. An SLG evaluation has the \textit{locality property} if any \textbf{Answer Resolution} operation applied to a state $\mathcal{F}_n$ occurs in an independent SCC of $\mathcal{F}_n$. 

Termination for Definite Datalog Programs

- Let’s consider a special case of (finite!) termination: that of definite datalog programs (i.e. definite programs that don’t contain any function symbols). Recall that the Herbrand Base for datalog programs is finite.

- Let $P$ be a (finite) datalog program in which every predicate is tabled. We want to show that any query $Q$ to $P$ terminates. We can show this by showing that any SLG evaluation is finite.

- Consider the New Subgoal operation. The definition stated: Assume $F_n$ contains no tree with root subgoal $S$. We assume that any two subgoals (atoms) that are variants are identical. Since the Herbrand Base is finite, there can only be a finite number of New Subgoal operations in the evaluation (and the final forest of the evaluation has only a finite number of trees).

- Similarly, the Program Clause Resolution definition stated: Assume that in $F_n$, $N$ does not have a child $N_{child} = (S :- |Body)\theta$. So any clause $S' :- Body$ where $S'$ unifies with $S$ can only be resolved once, and there can only be a finite number of Program Clause Resolution operations.
For the case of Positive Return, since the Herbrand Base is finite, there can only be a finite number of answers. The definition prevents the same answer being used for resolution against the same node more than once via the statement: Assume that in $F_n$, $N$ does not have a child $N_{child} = (S :- |Body)\theta$.

The definition of completion explicitly prevents a tree from being marked as complete more than once. Since there are a finite number of trees, there can only be a finite number of completion operations.

Since there can only be a finite number of each kind of operation the SLG evaluation is finite. Also, note that each tree will only have a finite number of nodes in it: we have a finite forest of finite trees.

Assuming ideal indexing, all operations except completion can be made in constant time. [SS98] provides an algorithm for performing incremental completion so that the sum of all completion operations is $O(size(P))$. 

---

**Termination for Definite Datalog Programs**
Tabling and Negation

A loop through negation occurs when a subgoal $S$ calls a series of subgoals that end up calling $\text{tnot}(S)$. We’ll treat this informally, but programs that contain no “loops through negation” can be formalized as fixed-order stratified programs [SSW00]

For programs that contain no loops through negation, we add the operation

- **Negative Return** Let $\mathcal{F}_n$ contain a leaf node

  \[
  N = \text{Ans} :\neg S, \text{Goal}_\text{List}.
  \]

  whose selected literal $\neg S$ is ground.

  1. **Negative Success**: If $S$ is failed in $\mathcal{F}$ and $N$ has no children, create a child for $N$ of the form: $\text{Ans} :\text{Goal}_\text{List}$.

  2. **Negative Failure**: If $S$ successful in $\mathcal{F}$ and $N$ has no children, create a child for $N$ of the form $\text{fail}$.

However, we will see that well-founded programs can contain loops through negation that become resolved.

\(^2\)The terms successful and failed will be defined later.
Negation: Example

\[ p(b). \]
\[ p(c) :- \text{not } p(a). \]
\[ p(X) :- t(X,Y,Z), \text{not } p(Y), \text{not } p(Z). \]
\[ t(a,b,a). \quad t(a,a,b). \]

The ground instantiation of the above program is:

\[ p(b). \]
\[ p(c) :- \text{not } p(a). \]
\[ p(a) :- t(a,a,a), \text{not } p(a), \text{not } p(a). \]
\[ p(a) :- t(a,a,b), \text{not } p(a), \text{not } p(b). \]
\[ \vdots \]
\[ p(a) :- t(a,b,a), \text{not } p(b), \text{not } p(a). \]
\[ v: \]
\[ p(c) :- t(c,c,c), \text{not } p(c), \text{not } p(c). \]
\[ t(a,b,a). \quad t(a,a,b). \]

This program has a 2-valued model: w.r.t \( p/1 \) it is

\[ \{p(b), p(c)\} = \text{true}; \{p(a)\} = \text{false} \]

but it cannot be evaluated with a fixed literal selection strategy.
The method just shown was pure bottom-up.

- To make it goal-oriented requires a notion of relevance. Assuming (as always) a left-to-right computation rule:
  - In Prolog, relevant literals for a selected clause belong to a failing prefix.
    \[ \text{p}(a) \leftarrow \text{t}(a,b,a), \textbf{not} \ \text{p}(b), \text{not} \ \text{p}(a). \]
  - To get this dynamic stratification an evaluation cannot view only a prefix.
    \[ \text{p}(a) \leftarrow \text{t}(a,a,b), \text{not} \ \text{p}(a), \textbf{not} \ \text{p}(b). \]

Are relevant literals all those in a body for a selected clause?
:- table p/1.
p(b).
p(c) :- not p(a).
p(X) :- t(X,Y,Z), not p(Y), not p(Z).
t(a,b,a).
t(a,a,b).

1. p(c) :- p(c)
2. p(c) :- not p(a)

3. p(a) :- p(a)
4. p(a) :- t(a,a,b), not p(a), not p(b)
5. p(a) :- not p(a), not p(b)

6. p(a) :- t(a,b,a), not p(b), not p(a)
7. p(b) :- not p(b), not p(a)

8. p(b) :- p(b) (completed)

9. p(b) :- 

10. fail
1. \( p(c) :\leftarrow | p(c) \)

2. \( p(c) :\leftarrow | \neg p(a) \)

3. \( p(a) :\leftarrow | p(a) \)

4. \( p(a) :\leftarrow | t(a,b,a), \neg p(b), \neg p(a) \)

5. \( p(a) :\leftarrow | \neg p(a), \neg p(b) \)

6. \( p(a) :\leftarrow | t(a,b,a), \neg p(b), \neg p(a) \)

7. \( p(b) :\leftarrow | \neg p(b), \neg p(a) \)

8. \( p(b) :\leftarrow | p(b) \)

9. \( p(b) :\leftarrow \)

10. fail

11. \( p(c) :\leftarrow \neg p(a) \)

12. \( p(a) :\leftarrow \)

13. fail

\[ :- \text{table } p/1. \]
\[ p(b). \]
\[ p(c) :\leftarrow \neg p(a). \]
\[ p(X) :\leftarrow t(X,Y,Z), \neg p(Y), \neg p(Z). \]
```prolog
:- table p/1.
p(b).
p(c) :- not p(a).
p(X) :- t(X,Y,Z), not p(Y), not p(Z).
t(a,b,a).
t(a,a,b).
```
Negation: Tabled Evaluation of WFS

Let’s review how each node was produced.

1. Initial Forest
2. Program Clause Resolution
3. New Subgoal
4. Program Clause Resolution
5. Program Clause Resolution (t/3 is not tabled)
6. Program Clause Resolution
7. Program Clause Resolution
8. New Subgoal
9. Program Clause Resolution
10. Completion of p(b) + Negative Return
11. Delaying
12. Delaying
13. Negative Return
14. Simplification

Note that the evaluation was not delay-minimal, and completion was not fully incremental. If the delaying operation in 12 had been performed and p(a) completed, no Delaying operation would have been needed in the p(c) tree.

Also note that we can mix evaluation of tabled with non-tabled predicates.
Negation: Tabled Evaluation of WFS

So *that’s* what the $|$ means.

- An answer is unconditional if there are no literals to the left of the $|$; otherwise it is conditional.
- Let $\mathcal{F}$ be a forest. We say that an atom $S$ is *failed in $\mathcal{F}$* if $S$ is completely evaluated in $\mathcal{F}$, and $S$ has no answers. We say that an atom $S$ is *successful in $\mathcal{F}$* if the tree for $S$ has an unconditional answer $S$.
- **SLG Answer Resolution** Let $N$ be a node $H \iff D \mid L_1, \ldots, L_n$, where $n > 0$. Let $Ans = A' \iff D'$ be an answer whose variables have been standardized apart from $N$. $N$ is *SLG resolvable with $Ans$* if $L_i$ and $A'$ are unifiable with an mgu $\theta$. The SLG resolvent of $N$ and $Ans$ on $L_i$ has the form
  \[(A \iff D \mid L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n)\theta\]
  if $D'$ is empty, and
  \[(A \iff D, \overline{D} \mid L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n)\theta\]
  otherwise, where $\overline{D} = L_i$ if $L_i$ is negative, and $\overline{D} = L_i^{L_i}_{A'}$ otherwise.

- Positive delayed literals are added *only* by SLG Answer Resolution, not directly by **Delaying**.
We introduced two new operations which may be applicable to a node in a forest $\mathcal{F}_n$:

- **Delaying**: \(^3\) Let $\mathcal{F}_n$ contain a leaf node

  \[
  N = \text{Ans} : - \text{DelaySet}|\text{not } S, \text{Goal}_\text{List}
  \]

  with selected literal \text{not } S such that $S$ is ground, in $\mathcal{F}_n$, but $S$ is neither successful nor failed in $\mathcal{F}_n$. Then create a child for $N$ of the form

  \[
  \text{Ans} : - \{\text{DelaySet} \cup \text{not } S\}|\text{Goal}_\text{List}
  \]

  if $N$ does not already have such a child.

---

\(^3\)This definition of Delaying assumes a left-to-right literal selection strategy.
Negation: Tabled Evaluation of WFS

**Simplification:** Let $\mathcal{F}_n$ contain a leaf node $N = \text{Ans} :- \text{DelaySet} |$ with $D \in \text{Delayset}$.

- If $D = \text{not Subg}$ and
  - $\text{Subg}$ is failed in $\mathcal{F}_n$ then create a child $\text{Ans} :- \text{DelaySet}' |$ for $N$, where $\text{Delay}_\text{Set}' = \text{Delay}_\text{Set} - \text{Subg}$, if $N$ does not already have such a child.
  - $\text{Subg}$ succeeds in $\mathcal{F}_n$ then create a child $\text{fail}$ for $N$, if $N$ does not already have such a child.

- Otherwise, if $D = L_{\text{Subg}}^{\text{Subg}}$ and
  - $\text{A'}$ succeeds in $\mathcal{F}_n$, then create a child $\text{Ans} :- \text{DelaySet}' |$ for $N$, where $\text{Delay}_\text{Set}' = \text{Delay}_\text{Set} - \text{Subg}$, if $N$ does not already have such a child.
  - $\text{A'}$ is failed in $\mathcal{F}_n$, then create a child $\text{fail}$ for $N$, if $N$ does not already have such a child.
Negation: Tabled Evaluation of WFS

- SLG Answer Resolution adds a positive literal to the delay list rather than the negative delayed literals in order to avoid exponential blowup in the answers returned.

- Delayed literals sometimes are depicted as $L^\text{Subgoal}_{\text{Answer}}$. A delayed literal $L$ may be unified by further resolution of literals, but the Subgoal before the answer resolution is not, nor is the Answer used for answer resolution.

- This is definition of Simplification explains the behavior of XSB and has a variant flavor. For non-ground positive literals
  - A delayed literal fails a clause if its Subgoal fails.
  - A delayed literal is removed if its Answer becomes unconditional.

- Other forms of subsumptive simplification are possible, such as failing the clause if $L\theta$ is failed, or removing the literal if there is a $L\theta$ that is conditional [Swi99]

- In a program $P$, if clauses all have the form
  \[ \text{Head} : \neg L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \]

  All bindings will have been made before any Delaying occurs.
Unfortunately, Delaying and Simplification are not always enough. Consider this program under local evaluation.

```prolog
:- table p/0, s/0, r/0.
p :- p. p :- tnot(s).
s :- tnot(r). s :- p.
r :- tnot(s), r.
```

- A query `?- p` will detect a positive loop using the first clause, then call `tnot(s)`. `s` will call `r` and `r` will call `s`; finally `s` will call `p`.
- At this point, `p`, `s` and `r` are in the same SCC.
- **Delaying** is applied to the literals `tnot(s)` and `tnot(r)`.
- The various computations are re-invoked: conditional answers `p :- tnot(s)`| and `s :- tnot(r)`| are derived. New conditional answers `s :- p`| and `p :- p`| are derived.
- `r` is failed. `s :- tnot(r)`| is simplified to a unconditional answer; the conditional answer `p :- tnot(s)` is removed through simplification.
- The conditional answer `p :- p`| remains, giving `p` a truth value of *undefined* rather than of *false*.
- This computation is informationally sound, but not complete.
Negation: Answer Completion

Let’s take a closer look at what was wrong with p. In SLG terminology, it was non-supported

Definition (Supported Answer)

Let $F$ be a SLG forest, $S$ a subgoal in $F$, and $Answer$ be an atom that occurs in the head of some answer of $S$. Then Template is supported by $S$ in $F$ if and only if:

1. $S$ is not completely evaluated; or
2. there exists an answer node

$$Answer :- \text{Delay}_\text{Set}$$

of $S$ such that for every positive delay literal $D_{Call}^{\text{Call}}$, $Ans$ is supported by $Call$.

Our final SLG operation removes non-supported answers:

**Answer Completion**: Given a set of non-supported answers $UA$, create a failure node as a child for each answer $Ans \in UA$. 
Negation: Residual Programs

- For a normal program $P$, the well-founded model of $P$ is contained in the intersection of all stable models of $P$ [Prz89]
- Thus, the conditional answers of a query to $P$ form a residual program that can be used to compute the stable models of $P$ using the notion of Revised Stable Models [PP05]
Termination for datalog programs is shown by a now-familiar argument for the new operations Negative Return, Delaying and Simplification.

For Answer Completion note that a given node $N$ in a set of non-supported answers is no longer an answer after this operation has been performed, due to the failure node that is created as the child of $N$. So Answer Completion can occur only a finite number of times.
Tabling with Negation: Complexity

- The new operations **Negative Return**, **Delaying**, and **Simplification** all take constant time (assuming ideal indexing). But WFS is $O(\text{atoms}(P) \times \text{size}(P))$. So where does the non-linearity creep in?
- First, note that detecting a set of unsupported answers is a $O(\text{size}(P))$ operation. As each operation reduces the potential unsupported answers by at least one, the worst-case is $\text{atoms}(P)$ applications of a $O(\text{size}(P))$ operation.
- It also is the case that early completion means that exact SCCs may need to be recomputed, again leading to up to $\text{atoms}(P)$ operations of $\text{size}(P)$ in order to support incremental completion.
- Experience shows that XSB programs almost never require answer completion, due to XSB’s reluctance to delay which is based on some non-linear aspects of its completion algorithm. So there is a trade-off.
- However it is a challenge to find programs that don’t scale linearly due to the SLG operations (you can write programs that scale non-linearly if you specify poor indexing, but that doesn’t count). Let me know if you construct any.
More on Termination

We’ve considered termination for datalog programs, but haven’t considered non-datalog programs, which have infinite Herbrand bases.

1 SLG, as so far described, does not terminate for the program

\[
p(X) :- \ p(f(X)).
\]
\[
p(1).
\]
which has no true answers.

2 It also does not terminate on the program

\[
p(s(X)) :- \ p(X).
\]
\[
p(1).
\]
which has an infinite number of (true) answers.
More on Termination

- Note that in the previous slide, case (1) produced a forest with an infinite number of trees; case (2) produced a forest with one tree, but that tree had an infinite number of nodes.
- Let’s consider the case of an infinite number of subgoals/trees, and let’s consider a mechanism called *call subsumption*. 
Call Subsumption

- Recall the **New Subgoal** definition creates a new tree if $\mathcal{F}_n$ contains no tree with root subgoal $S$, where subgoals are taken to be identical if they are variants. This is sometimes called *tabling with call variance*.

- One could use the definition Assume $\mathcal{F}_n$ contains no tree with root subgoal $S'$, such that $S'$ subsumes $S$, which is sometimes called *tabling with call subsumption*.

- Note that any answer to $S$ will be an answer for the subsuming subgoal $S'$. We just need to make sure that an answer for $S'$ used to resolve away a selected literal containing $S$ actually unifies with $S$.

- Call subsumption is stronger than call variance in that it will reuse results that call variance would not, and may result in shorter evaluations.

- In fact, it is easy to prove that no SLG evaluation with call subsumption is ever longer than an SLG evaluation with call variance, and may be shorter.
Call Subsumption

- It might seem that call variance would solve case (1). For the query `?- p(X)`, a new subquery `p(f(X))` is immediately made. `p(f(X))` is subsumed by `p(X)` so a new tree will not be created for it. At this point, there are no other operations applicable for `p(X)` so the query (correctly) fails.

- However, for the query `p(f(1))`, the subqueries `p(f(f(1)))`, `p(f(f(f(1))))`,... will be created, none of which are subsumed by any other query.

- Call subsumption is still important, because of the efficiency property in the previous slide.
Case 1 can be addressed via call abstraction. When a subgoal \( S \) is created, any arguments in \( S \) with depth greater than some chosen number \( d \) are rewritten as variables.

For instance, if \( d = 3 \), \( p(f(f(f(A)))) \) is rewritten as \( p(f(f(X))) \). At this stage, the correct result in case 2 is derived regardless of whether call variance or call subsumption is otherwise used.

For a program with a two-valued WFM, tabling with call abstraction is complete for any program with a finite model.

Call abstraction was introduced in [TS86], and extended to the well-founded and the distribution semantics in [RS12].
Program Classes for Termination

- [RS12] formalizes the class of *bounded-term size* programs, and shows that tabling with call abstraction terminates for such programs.
- Whether a program is bounded-term size is semi-decidable – you can tell when a program is, but not when it isn’t.
- This class appears to be co-extensive with Finitely Ground programs [CCIL08] and to properly include fully decidable termination classes in the literature such as finitely recursive [BBC09], finite domain programs [CCIL08], and argument-reduceable programs [LL09].
Case (2), which has an infinite number of answers is harder to address.

Some problems may benefit from answer abstraction where an answer $A$ is transformed to some other answer $A'$, so that there will be a finite number of transformed answers, even if there are an infinite number of raw answers.

This makes semantic sense for some programs, and can be thought of as a kind of abstract interpretation, but is not a general solution.

As a concrete example, problems of reachability in Petri nets may abstract configurations by marking one of the states in the configuration as a special 'omega-state'. [SW10] shows an example of this using tabling.

For practical KRR problems, it is often best to throw an exception when adding an answer whose depth is greater than some pre-set level.
As an illustration of what you can do with WFS, we consider an approach to defeasible reasoning [WGK+09]

- There are two user-defined predicates: `opposes/2` and `overrides/2`.
- Two atoms $A_1$ and $A_2$ oppose each other if $A_1$ and $A_2$ cannot both be in the same model. `opposes/2` is usually symmetric.
- An atom $A_1$ overrides an atom $A_2$ if the truth of $A_1$ in a model makes $A_2$ false in the model. `overrides/2` is usually transitive.
Courteous Logic Programs

Assume each rule

\[ H :- \text{Body} \]

is rewritten as

\[ H :- \text{Body}, \neg \neg (\text{defeated}(H)) \]

where defeated/1 is implemented as followed (this will look confusing at first :-)

\[- \text{table defeated/1.} \]
\[
\text{defeated}(A):- \text{defeated}_by(A, \_B). \quad \text{defeated}(A):- \text{defeats}(\_B, A). \\
\text{defeated}_by(A,B):- \text{refutes}(B,A). \quad \text{defeats}(A,B):- \text{refutes}(A,B). \\
\text{defeated}_by(A,B):- \text{rebuts}(B,A). \quad \text{defeats}(A,B):- \text{rebuts}(A,B). \\
\text{refutes}(A,B):- \text{conflicts}(A,B), \text{overrides}(A,B). \\
\text{rebuts}(A,B):- \text{conflicts}(A,B). \\
\text{conflicts}(A,B):- \text{opposes}(A,B), B. \]
Courteous Logic Programs

- Suppose you only specify `opposes(A, neg A)`, where `neg A` is the explicit negation of `A`, and `opposes/2` is symmetric.
- Since the predicate `overrides/2` is empty, the code of the last slide can be folded and reduced to:

  ```prolog
  :- table defeated/1.
defeated(A):- opposes(A,B),B. defeated(A):- opposes(B,A),B.
  ```

  which is the same as the semi-normal translation in [ADP95].
- As an example, using this defeasibility theory on the program:

  ```prolog
  p  neg p.
  ```

  both `p` and `neg p` have the truth value undefined.
opposes/2 can of course be used for other purposes. For instance a rule such as

\[
opposes(location(Obj,\text{Loc1},\text{Time}), location(Obj,\text{Loc2},\text{Time})): -
\]

\[\text{location(Loc1), location(Loc2), Loc1} \neq \text{Loc2}.\]

indicates that an object cannot be in two places at once.

By specifying that one atom overrides another, you can have preference logic programming in the style of [CS02].

Generally speaking, Subject Matter Experts (SMEs) and many KEs (Knowledge Engineers) will not want to devise their own argumentation theory, but will want to incorporate one. We’ll see later that this is the approach FLORA-2 and Silk use.
Defeasibility of Rules

- Many defeasibility theories talk about opposition of rules as opposed to opposition of literals (e.g. [BE98]).
- If you want labelled rules, such as
  
  republican_rule: unconstitutional(Law):- imposes_mandate(Law).
  democratic_rule: constitutional(Law):- regulates_commerce(Law)

  You can simply transform the labels into atoms in the body, so the atom-based approach is general.
- In this way, a rule may be *disqualified* if it rebuts itself, thus failing a derivation rather than making it undefined.
Defeasibility in Flora-2

- Flora-2 [YKWZ12] implements exactly this type of defeasibility.
- By “default” rules are strict – they do not have the defeated literal added to their body.
- Flora-2 supports 3 defeasibility theories
  - Original Generalized Courteous Logic: a rule $R$ is defeated if another rule refutes, cancels, or rebuts $R$.
  - New Generalized Courteous Logic: a rule $R$ is defeated if another rule $R'$ refutes, cancels, or rebuts $R$ and $R$ is itself not defeated.
  - Generalized Courteous Logic with Exclusion Constraints adds the ability to have more than two literals in opposition to each other: if $n$ literals oppose each other they may not all be true, but $n - 1$ of them could be true. For instance, a student might take 2 classes from a list, but not more than 2.
Defeasibility in Flora-2

Here is an example from the FLORA-2 manual (using Prolog syntax). The underlying argumentation theory specifies that a rule is disqualified if it is cancelled.

device(fax).  abused(bill,scanner).
  abused(bill,printer).
  abused(mary,fax).

person(bob).  person(bill).  person(mary).

@{id1}  authorized(Persn,Device) :-
  device(Device),  person(Persn).

@{id2(Dev,Persn)}  _cancel(id1,authorized(Persn,Dev)) :-
  abused(Persn,Dev).

@{id3}  _cancel(id2(Device,Persn)) :-
  pardoned(Device,Persn).
So far, we’ve seen that tabling can add some powerful termination properties to Prolog and furthermore can support WFS and defeasibility theories.

But is this enough to substantiate the claim:

*Tabling fundamentally changes Prolog so that it is suitable as a KRR tool?*

Next lecture we’ll examine *FLORA-2* and Silk.
References I

A logic programming system for non-monotonic reasoning.

On finitely recursive programs.

Pushing the $\mathcal{EL}$ envelope.

Preferred answer sets.

[CCIL08] F. Calimeri, S. Cozza, G. Ianni, and N. Leone.
References II


References III

Concurrent and local evaluation of normal programs.

[PP05] L. M. Pereira and A. M. Pinto.
Revised stable models - a semantics for logic programs.

The well-founded semantics coincides with the three-valued stable semantics.
Fundamenta Informaticae, 1989.

Well-definedness and efficient inference for probabilistic logic programming under
the distribution semantics.
Theory and Practice of Logic Prog., 2012.

An abstract machine for tabled execution of fixed-order stratified logic programs.
References IV

The limits of fixed-order computation.  

Tabling with answer subsumption: Implementation, applications and performance.  

A new formulation of tabled resolution with delay.  

OLDT resolution with tabulation.  

Logic programming with defaults and argumentation theories.  