Tabled Logic Programming and its Applications

Part 2: Tabling for Non-Monotonic and Quantitative Reasoning

Terrance Swift

1999 Summer School in (Constraint) Logic Programming

Overview

- Introduction: Why is non-monotonic or quantitative logic programming useful?
- Stratified and unstratified negation.
- Programming with negation in XSB.
- Tabled aggregation in XSB.
- Practical Applications

Introduction

- Non-monotonic programming can be useful for representing knowledge succinctly.
  - Consistent beliefs of a rational agent;
  - Diagnostic model of a system;
  - Plan of action of an agent, etc;
  - Determining "best" solutions, such as best parses.
- Useful for representing *inconsistent* information.

Introduction

Non-monotonic programming can be useful for implementing specialized (and powerful) logics.

- Greatest Fixed Points
  - Used by temporal logics such as the Modal-mu calculus and many others.
  - In principle, useful for narrowing in program analysis.
- Recursion through aggregation.

Introduction

Quantitative Logic Programming

- Useful to determine when information is "more" true.
  - Probabilistic Reasoning
  - Fuzzy Reasoning
- Useful to express contradictory information: to *mediate* when information in a group of databases contradicts is inconsistent.

Overview

- Introduction: Why is non-monotonic or quantitative logic programming useful?
- *Stratified and unstratified negation.*
- Programming with negation in XSB.
- Tabled aggregation in XSB.
- Practical Applications

Stratified and Non-Stratified Negation

Consider Russell's paradox:

> *A barber in a town shaves everyone in town who does not shave themselves*

as a logic program

```
:- table shaves/2.
shaves(barber,Person):-
                        in_town(Person),
                        tnot(shaves(Person,Person)).

in_town(barber).
in_town(mayor).
```

In the minimal model of this program,

`shaves(barber,mayor)`

is true but what about

`shaves(barber,barber)`?

Stratified and Non-Stratified Negation

- Two prominant approaches:
  - Stable model semantics
    - 2-valued: atoms can be true or false, but not all programs have a stable model.
  - Well-founded semantics (WFS)
    - 3-valued: atoms can be true, false, or undefined.
    - True and false atoms are those that occur in "stratified" portions of the program. Undefined atoms are those that remain.
  - The well-founded partial model is contained in all stable models.

Stratified and Non-Stratified Negation

Review: T_P operator for definite programs

- An interpretation is a set of atoms. For an atom A,
  - $I \models A$ iff $A \in I$
  - $I \not\models A$ iff $A \notin I$
- For a ground program P
  - $T_P(I) = \{Head | Head \text{ :- } Body \in P$ and $I \models Body\}$
  - $T_P(I)$ is monotonic, i.e. $I_1 \subseteq I_2 \Rightarrow T_P(I_1) \subseteq T_P(I_2)$
  - Monotonic operators have a least fixed point. The fixed-point model of P is the least fixed point of T_P
  - This model is computed by tabling but not by Prolog (which computes the Clark Completion model of P)

Stratified and Non-Stratified Negation

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

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

1. $T_P(\emptyset) = \{edge(a,b), edge(b,c), edge(c,d)\}$
2. $T_P^2(\emptyset) = T_P(\emptyset) \cup \{path(a,b), path(b,c), path(c,d)\}$
3. $T_P^3(\emptyset) = T_P^2(\emptyset) \cup \{path(a,c), path(b,d)\}$
4. $T_P^4(\emptyset) = T_P^3(\emptyset) \cup \{path(a,d)\}$

Stratified and Non-Stratified Negation

- Can the T_P operator be extended to normal programs?
- Immediate problem with monotonicity

Example: In the program

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

$\{q\} \models p$, but $\{q,r\} \not\models p$

Intuition: stratify the program

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

`s` should be true, because `p`, `q`, `r` are all involved in a positive loop.

```
s:- tnot(p(0)).
p(0):- p(1).
p(1):- p(2).
p(2):- p(3).
:
```

`s` should be true, because `p(N)` cannot be proved true for any N.

All you have to do is to bear in mind that loops and infinite chains do not prove anything true. Then non-stratified programs become easy to interpret!

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

$\{p, q, r\}$ is an example of an *unfounded set* (w.r.t. the empty interpretation).

A set is unfounded *w.r.t.* an interpretation

```
s:- tnot(p),tnot(q),tnot(r).
p:- tnot(r),q,tnot(s).
p:- a.
q:- tnot(p),r.
q:- b.
r:- p,tnot(q).
```

$\{p, q, r\}$ is now unfounded w.r.t. the interpretation $\{not(a), not(b)\}$ but not w.r.t the empty interpretation.

Stratified and Non-Stratified Negation

Normal Programs:

- An interpretation is a set of literals. For an atom A,
  - $I \models A$ iff $A \in I$
  - $I \models not(A)$ iff $not(A) \in I$
- An interpretation is partial if $I \subset H_P$, consistent if there is no A such that $I \models A$ and $I \models not(A)$.

Stratified and Non-Stratified Negation

- Unfounded sets
  - For a ground program P, and consistent partial interpretation I, a set U of atoms is *unfounded* if for each $A \in U$, then for all rules A :- $Body \in P$,
    * Some literal $L \in Body$ is inconsistent with I ($L = not(A)$ and $I \models A$, or $L = A$ and $I \models not(A)$); or
    * Some positive literal $L \in Body$ occurs in A.
- Union of two unfounded sets w.r.t I is also unfounded. Call the operator that produces the maximal unfounded set U_I.

Stratified and Non-Stratified Negation

- Extend the T_P operator to factor out the non-monotonic part
  - $$T_P^J(I) = \{Head \mid \begin{array}{l} Head \text{ :- } Body \in P \\ \text{and } I \models Poslits(Body) \\ \text{and } J \models Neglits(Body)\} \end{array}$$
  - monotonic w.r.t. I
  - $T_P^J(I)$ is related to the Gelfond-Lifschitz operator:
$$\Gamma_P(J) = \lambda J.lfp(T_P^J(\emptyset))$$
- WFM is an iterated fixed point
  - $I_0 = \emptyset$
  - $I_{i+1} = T_P^{I_i}(\emptyset) \cup U(I_i)$

Stratified and Non-Stratified Negation

- It can be shown that a lfp of this iterated fixed point exists.
- It can be shown that each level of iteration corresponds to a level of *dynamic* stratification.

```
s:- tnot(p),tnot(q),tnot(r).
p:- tnot(r),q,tnot(s).
p:- a.
q:- tnot(p),r.
q:- b.
r:- p,tnot(q).
```

  - Level 0: $\emptyset$
  - Level 1: $\{not(a), not(b)\}$
  - Level 2: $\{not(p), not(q), not(r)\}$
  - Level 3: $\{s\}$

Overview

- Introduction: Why is non-monotonic or quantitative logic programming useful?
- Stratified and unstratified negation.
- *Programming with negation in XSB.*
- Tabled aggregation in XSB.
- Practical Applications

Programming with Negation in XSB

Intuition: delaying allows us to execute

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

in a manner similar to

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

Simplification also allows us to evaluate delayed literals when their truth-value becomes known.

Programming with Negation in XSB

Consider the program

```
:- table p/0.
p:- tnot(p).
```

and trace

```
XSB Version 2.0 (Gouden Carolus) of June 27, 1999
[i586-pc-linux-gnu; mode: optimal; engine: chat;
                          scheduling: batched]
| ?- [test].
[Compiling ./test]
[test compiled, cpu time used: 0.1510 seconds]
[test loaded]

yes
| ?- p,tnot(p).

yes
| ?-
```

- If an atom A is undefined, both `A` and `tnot(A)` succeed!
- XSB defines success as true or undefined.

Programming with Negation in XSB

Consider the program

```
:- table p/1,r/0,s/0.
p(X):- tnot(s).

s:- tnot(r).
s:- p(X).

r:- tnot(s),r.
```

- Note XSB table declarations, `tnot/1`.
- This program is stratified. The well-founded model is $\{not(r), s, (\forall X.not(p(X)))\}$

Programming with Negation in XSB

Consider the following trace:

```
XSB Version 2.0 (Gouden Carolus) of June 27, 1999
[i586-pc-linux-gnu; mode: optimal; engine: chat;
                                    scheduling: batched]
| ?- [table_examples].
[Compiling ./table_examples]
[table_examples compiled, cpu time used: 0.3500 seconds]
[table_examples loaded]

yes
| ?- p(X).

X = _h105;

no
| ?- p(X).

no
| ?-
```

Programming with Negation in XSB

```
p(X):- tnot(s).

s:- tnot(r).
s:- p(X).

r:- tnot(s),r.
```

- When bold literals are evaluated, all literals are undefined — no proofs, no unfounded sets.

```
2) p(X):- tnot(s).|

1) s:- tnot(r).|
3) s:- p(X).|

4) r:- tnot(s),|r.
```

Programming with Negation in XSB

Alternatively the try the query

```
| ?- p(X),get_residual(p(X),F).

X = _h180
F = [tnot(s)];

no
?-
```

- The *conditional* answer `p(X):- tnot(s)` is returned to the interpreter level before it is simplified away.

Programming with Negation in XSB

- Make sure that a subgoal is completely evaluated before using answers from it.
- This can be ensured by failing at the interpreter level, i.e.
  - ?- (p(X),fail ; p(X)).
- Complete evaluation can also be ensured by choosing local evaluation.

Programming with Negation in XSB

- Given a program P and query Q, the residual program for P and Q is the set of all answers created in the course of completely evaluating Q

```
presents_delay(terry):-
                        tnot(presents_delay(david)),
                        tnot(presents_delay(david)),
                        has_time(terry).

presents_delay(david):-
                        tnot(presents_delay(terry)),
                        tnot(presents_delay(cr)),
                        has_time(david).

presents_delay(cr):-
                        tnot(presents_delay(terry)),
                        tnot(presents_delay(david)),
                        has_time(cr).
```

Programming with Negation in XSB

- Given the program on the last slide, and query **`?- presents_delay(X)`** the residual program is:

```
presents_delay(terry):-
                        tnot(presents_delay(david)),
                        tnot(presents_delay(david))|.

presents_delay(david):-
                        tnot(presents_delay(terry)),
                        tnot(presents_delay(cr))|.

presents_delay(cr):-
                        tnot(presents_delay(terry)),
                        tnot(presents_delay(david))|.
```

Programming with Negation in XSB

- Programming with negation in XSB requires accounting for simplification and delay.
  - Since undefined literals succeed in XSB, its usually best to make sure to completely evaluate queries with negation.
  - Delay and simplification is only an issue if a program is not *left-to-right dynamically stratified*.
  - Technically an answer completion operation is also needed.
- The residual program can be used for further programming.
- `get_residual/2` can be used to inspect conditional answers, i.e. as a `clause/2` predicate

Overview

- Introduction: Why is non-monotonic or quantitative logic programming useful?
- Stratified and unstratified negation.
- Programming with negation in XSB.
- *Tabled aggregation in XSB.*
- Practical Applications

Tabled Aggregation in XSB

Consider the extension of the path/2 predicate that counts the cost of a path between two vertices in a graph.

```
:- table path/3.
path(X,Y,C) :-   path(X,Z,C1),
                 edge(Z,Y,C2),
                 C is C1 + C2.
path(X,Y,C) :-   edge(X,Y,C).
```

How do we compute the *shortest* path?

Tabled Aggregation in XSB

For the following edb:

```
edge(1,2,1).        edge(2,2,1).
edge(2,3,1).        edge(2,4,1).
edge(3,5,2).        edge(4,5,1).
```

We get the following trace:

```
| ?- path(1,5,X).

X = 4;

X = 3;

X = 5;

X = 6;

X = 7;

++Warning: Removing incomplete tables...

yes
| ?-
```

Tabled Aggregation in XSB

```
:- import                filterReduce1/4 from aggregs.

shorter_path(X,Y,C) :-  filterReduce1(sp(X,Y),min,infinity,C).

sp(X,Y,C) :-            shorter_path(X,Z,C1),
                        edge(Z,Y,C2),C is C1 + C2.
sp(X,Y,C) :-            edge(X,Y,C).

min(X,Y,Y):-            \+ number(X),!.
min(X,Y,X):-            \+ number(Y),!.
min(One,Two,Min):-      One > Two -> Min = Two ; Min = One.
```

- `sp(X,Y)` indicates the predicate (`sp(X,Y,C)`). to whose answers an associative function is to be applied
- `min/3` is an associative function.
- `infinity` is the minimum value of `min/3`.

Tabled Aggregation in XSB

Now XSB terminates:

```
?- shorter_path(1,5,C).

C = 4;

C = 3;
```

But how can we avoid returning non-optimal solutions?

Tabled Aggregation in XSB

`filterReduce1/4` is only useful for demonstration purposes.

```
filter_reduce(_Call,_Function,_Identity,_Value):-
          filter_reduce(_Call,_Function,_Identity,_Value),fail.
filter_reduce(Call,Function,Identity,Value):-
          filter_reduce(Call,Function,Identity,Value).
```

```
 :- import filterReduce/4 from aggregs.

 shortest_path(X,Y,C) :- filterReduce(sp(X,Y),min,infinity,C).

 sp(X,Y,C) :- shortest_path(X,Z,C1),
 edge(Z,Y,C2),C is C1 + C2.
 sp(X,Y,C) :- edge(X,Y,C).
```

Now the trace is

```
?- shorter_path(1,5,C).

C = 3;

no
```

Tabled Aggregation in XSB

- `filterReduce/4` can be used to implement Generalized Annotated Programs (GAPs), and through GAPs it can implement fuzzy and probabilistic reasoning (see the exercise below).
- `filterPO(Call,Var,Partial_order)` is another form of tabled aggregation.
  - Ex. `filterPO(p(X,Y),Var,preferred)` succeeds for those answers `p(X1,Y1,Var1)` such that there is no other answer `p(X1,Y1,Var2)` such that
    `preferred(Var2,Var1)` is true.
- `bagReduce/4`, `bagPO/4` are also present in XSB and use HiLog.
  - More elegant but less robust than `filterReduce/4` and `filterPO/4`.
- In version 2.0 of XSB, these predicates do not work properly with simplification and delay

Overview

- Introduction: Why is non-monotonic or quantitative logic programming useful?
- Stratified and unstratified negation
- Programming with negation in XSB
- Tabled aggregation in XSB
- *Practical Applications*

Practical Applications

- Case study of how WFS can be used for psychiatric assesment.
- Relies on the ability to write a meta-interpreter!

```
:- table demo/1.

demo(true).
demo((X,Y)):- !,demo(X),demo(Y).
demo(not(X)):- !,tnot(demo(Y)).
demo(X):- clause(X,Y),demo(Y).
```

- WFS reflects through meta-interpreter.

Practical Applications

- With meta-interpreters, we can implement new non-monotonic constructs.
- Explicit negation: requires a statement to be proved false, rather than assuming a statement false because the statement cannot be proved to be true.
- Useful for representing medical reasoning.
  - Default: A patient does not have a brain tumor because he presents no symptoms, or because the symptoms that he presents can be better accounted for in some other way.
  - Explicit: A patient does not have a brain tumor because an MRI was run, and the brain was seen to be healthy.

Practical Applications

- Programs with explicit and default negation are called *extended logic programs*.
- The well-founded semantics with explicit negation (WFSX) extends WFS to extended logic programs.
- Answer sets extend stable models to extended logic programs.
- Both obey the *coherency principle*: Infer the default negation $not(A)$ of an atom A from its explicit negation $-A$ ($A = - - A$).

Practical Applications

```
a:- b.

b:- c.
-b.

c:- not c.

e:- d.

d.  -d.
```

- The paraconsistent WFSX model of the above program is

  $\{-b, d, -d, e, not\, a, not\, -a, not\, b, not\, -c, not\, d, not\, -d, not\, e, not\, -e\}$

Practical Applications

```
a:- b,not -a.

b:- c,not -b.
-b:- not b.

c:- not c,not -c.

e:- d,not -e.

d:- not -d.  -d:- not d.
```

- The *semi-normal rewrite* of a program enforces coherency, but makes too many literals undefined.

Practical Applications

$\texttt{a}_t\texttt{:- b}_t\texttt{.}$	$\texttt{a}_{tu}\texttt{:- b}_{tu}\texttt{,not -a}_t\texttt{.}$
$\texttt{b}_t\texttt{:- c}_t\texttt{.}$	$\texttt{b}_{tu}\texttt{:- c}_{tu}\texttt{,not -b}_t\texttt{.}$
$\texttt{-b}_t\texttt{.}$	$\texttt{-b}_{tu}\texttt{:- not b}_t\texttt{.}$
$\texttt{c}_t\texttt{:- not c}_{tu}\texttt{.}$	$\texttt{c}_{tu}\texttt{:- not c}_{tu}\texttt{,not -c}_t\texttt{.}$
$\texttt{e}_t\texttt{:- d}_t\texttt{.}$	$\texttt{e}_{tu}\texttt{:- d}_{tu}\texttt{,not -e}_t\texttt{.}$
$\texttt{d}_t\texttt{.}$	$\texttt{d}_{tu}\texttt{:- not -d}_t\texttt{.}$
$\texttt{-d}_t\texttt{.}$	$\texttt{-d}_{tu}\texttt{:- not d}_t\texttt{.}$

- Solution is to thread execution of the program with execution of its semi-normal rewrite.
- The WFSX model, $WFSX(P)$ can be computed by the WFS model of the above program, which has undergone the T-TU transformation.
  - A positive literal L is in $WFSX(P)$ if L_t is in $WFS(P_{T-TU})$.
  - A negative literal L is in $WFSX(P)$ if $not(L_{tu})$ is in $WFS(P_{T-TU})$.

Practical Applications

$\texttt{a}_t\texttt{:- b}_t\texttt{.}$	$\texttt{a}_{tu}\texttt{:- b}_{tu}\texttt{,not -a}_t\texttt{.}$
$\texttt{b}_t\texttt{:- c}_t\texttt{.}$	$\texttt{b}_{tu}\texttt{:- c}_{tu}\texttt{,not -b}_t\texttt{.}$
$\texttt{-b}_t\texttt{.}$	$\texttt{-b}_{tu}\texttt{:- not b}_t\texttt{.}$
$\texttt{c}_t\texttt{:- not c}_{tu}\texttt{.}$	$\texttt{c}_{tu}\texttt{:- not c}_{tu}\texttt{,not -c}_t\texttt{.}$
$\texttt{e}_t\texttt{:- d}_t\texttt{.}$	$\texttt{e}_{tu}\texttt{:- d}_{tu}\texttt{,not -e}_t\texttt{.}$
$\texttt{d}_t\texttt{.}$	$\texttt{d}_{tu}\texttt{:- not -d}_t\texttt{.}$
$\texttt{-d}_t\texttt{.}$	$\texttt{-d}_{tu}\texttt{:- not d}_t\texttt{.}$

1. $\{-b_t, d_t, -d_t, e_t, not(-a_t), not(-c_t), not(-e_t)\ not(-a_{tu}), not(-c_{tu}), not(-e_{tu})\}$
2. $\{not(b_{tu}), not(d_{tu}), not(-d_{tu}), not(e_{tu})\}$
3. $\{not(a_{tu})\}$

Practical Applications

- There exists an embedding from WFSX into WFS (an objective literal is either an atom or its explicit negation):

  Let P be an extended logic program. Then a normal logic program P^{T-TU} is derived from P by replacing each rule R in P with two new rules.

  – In the first rule, all default literals of the form *not* L in R are replaced by literals of the form *not* L_{TU}, while all other literals of the form L are replaced by literals of the form L_T.

  – In the second rule, if O is the objective literal in the head of R, then the literal *not* O' is added to the body of R, where O' is the explicit complement of O. Next, all default literals of the form *not* L in R are replaced by literals of the form *not* L_T, while all other literals of the form L are replaced by literals of the form L_{TU}.

- The transformation is simple and linear.

Practical Applications

- The T-TU transformation is equivalent to the following meta-interpreter.
- This meta-interpreter can be partially evaluated to a more efficient program
- XSB has a WFSX package with a preprocessor that uses the T-TU transformation.

```
demo(A):- demo_t(A).
demo(A):- demo_tu(A).

:- table demo_t/1, demo_tu/1.
demo_t(true).
demo_tu(true).
demo_t(',' (A,B)):-!, demo_t(A),demo_t(B).
demo_tu(',' (A,B)):-!, demo_tu(A),demo_tu(B).
demo_t(not A):-!, tnot(demo_tu(A)).
demo_tu(not A):-!, tnot(demo_t(A)).
demo_t(A):- clause(A,B), demo_t(B).
demo_tu(A):-clause(A,B), demo_tu(B),
     exchange(A,A_opp),demo_tu(not A_opp).

exchange(-A,A):-!.
exchange(A,-A).
```

Practical Applications

- XSB has implemented WFSX for a psychiatric assesment system called *Diagnostica* (Web address `http://www.medicinerules.com`)
- Based on the Diagnostic and Statistical Manual of Mental Disorders, Fourth edition, which Requires non-stratified negation.
- Explicitly false atoms differentiated from default false atoms

Practical Applications

Diagnostic Criteria for Dementia of the Alzheimer's Type

1. The development of multiple cognitive defects manifested by both
   (a) memory impairment (impaired ability to learn new information or to recall previously learned information)
   (b) one (or more) of the following cognitive disturbances
      i. aphasia (language disturbance)
      ii. apraxia (impaired ability to carry out motor activities despite intact motor function)
      iii. agnosia (failure to recognize or identify objects despite intact sensory functions)
      iv. disturbance in executive functioning (i.e., planning, organizing, sequencing, abstracting)
2. The cognitive deficits in criteria A1 and A2 each cause significant impairment in social or occupational functioning and represent a significant decline from a previous level of functioning.
3. The course is characterized by gradual onset and continuing cognitive decline.
4. The cognitive deficits in Criteria A1 and A2 are not due to any of the following
   (a) other central nervous system conditions that cause progressive deficits in memory and cognition (e.g. cerebrovascular disease, Parkinson's disease, Huntington's disease, subdural hematoma, normal-pressure hydrocephalus, brain tumor)
   (b) systemic conditions that are known to cause dementia (g. hypothyroidism, vitamin B_{12} or folic acid deficiency, niacin deficiency, hypercalcemia, neurosyphilis, HIV infection
   (c) substance-induced conditions
5. The deficits do not occur exclusively during the course of a delirium
6. *The disturbance is not better accounted for by another Axis I disorder (e.g., Major Depressive Disorder, Schizophrenia)*

Practical Applications

- Thus, WFSX is being used to develop a commercial product.
- Preference Logic Grammars have been used to clean dirty commercial data (using `filterPO/4`). Preferences can be declared about possibly ambiguous parses.
- Preference Logic and Annotated Programs have been used to mine data about part information for the Defense Logistics Agency
- Default negation is used to implement the modal-mu calculus for model checking.
- Other research applictions: such as updates to logic programs, and abduction over the well-founded semantics.
- Many other applications of tabling in which non-monotonic or quantitative information does not play so large a part.

Conclusions

- Still some rough edges in XSB's implementation of these features.
- This brand of non-monotonic and quantitative logic programming certainly has its uses. Whether or not it becomes more widespread is up to you!

Further Reading

The following is provided for anyone wishing to read further on the topics introduced in these slides.

The stable model semantics was introduced in [5], while WFS was introduced in [15]. [10] gives a nice account of how the two approaches are related. The fixed-point defintion in this talk was based on [15]. However, other fixed point definitions for the WFS are quite valuable for understanding how it is computed, including [14] and [9] which introduces dynamic stratification. SLG was introduced in [2] and this paper defines ideas like delay, simplification, complete evaluation, and so on. The student wishing to learn SLG may prefer [13] which reformulates SLG so that it is easier for computer scientists to understand. The best reference of how to program using XSB is, of course, the XSB manual [11], and chapter 5 of the manual gives a tutorial introduction. Probably the best summary of the well-founded semantics with explicit negation is in [1] (their analogue for stable models, answer sets, was introduced in [6]; The presentation of the WFSX meta-interpreter given here follows that of [12]. Of other formalisms, Generalized Annotated Logic Programs were introcuded in [8], and an XSB meta-interpreter for GAPs described in [12]. Preference logic was introduced in [7] and preference logic grammars in [3]. Their implementation and a case study of their use was introduced in [4]. Happy reading!

1 Exercises

The associated handout "Using Tabling in XSB: A Tutorial Introduction" has a number of excercises that address the implementation of non-monotonic reasoning in XSB. Of particular importance are Exercises 5.3.6, 5.3.7 and 5.3.8 which help tie together knowledge of SLG and its implementation in XSB.

Further exercises:

Exercise 1.1 *The vanilla WFS meta-interpreter presented above requires tabling to terminate, but is inefficient since it copies into tables not only calls to predicates, but conjunctions of body literals and negative literals. Rewrite the meta-interpreter to be more efficient by arranging that only calls to predicates are tabled, without affecting its termination properties.*

Does this meta-interpreter have different complexity properties than the one presented in the slides?

Exercise 1.2 *Based on the meta-interpreter presented in the slides, write a predicate to determine the truth value of ground atoms in WFSX given values of atoms in the WFS model of a T-TU transformed program. Specifically, write a predicate* `wfsx_call(?Tpred,?Val)` *such that if* `Tpred` *is a ground call to a tabled predicate,* `wfsx_call(?Tpred,?Val)` *calls* `Tpred` *and unifies* `Val` *with the truth value of* `Tpred` *under the well-founded semantics with explicit negation.*

Exercise 1.3 *Tabled aggregation can be used to implement various forms of* Annotated Programs *in which atoms are associated with an explicit annotation, which may correspond to a "truth-value". So-called* Generalized *Annotated Programs [8] have the syntax of Horn Clauses except they are comprised of annotated atoms instead of atoms. These annotated atoms have the form*

$$Atom : Annotation$$

where A is an atom and Annotation the "truth value" of the atom.

The domain of annotations in GAPs is a semi-lattice with a bottom element. That is, a set in which an (associative) join operator is defined along with a bottom element that is an identity on the join element. Given an interpretation I (set of annotated atoms) of a GAP, and a ground annotated atom $A : \mu$, then

$$I \models A : \mu \textit{ iff } (A : \nu \in I \textit{ and } \mu \leq_{\mathcal{T}} \nu)$$

The analogy of a T_P operator for GAPs is

Definition 1.1 *[8] Let P be a GAP over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$, I be an R-interpretation, and A a ground atom in H_P. Then $R_P(I)(A) = lub\{\mu | A : \mu \leftarrow Body$ is a ground instance of a clause in P, and $I \models Body\}$.*

The model of a GAP is defined as the least fixed point of R_P (which is monotonic over $\mathcal{T}$).

Considering the fixed point definition, it is not surprising that GAPs can be implemented using tabled aggregation. For instance, `filterReduce/4` *can be used by setting the associative function to be the join operator over* $\mathcal{T}$*, and the identity to be* $\perp$ *over* $\mathcal{T}$*.*

A GAP meta-interpreter is provided in ??? which uses a close analog of `filterReduce/4` *called* `filterLub(Type,Atom,Annotation)` *in which* `Type` *is the name of the lattice,* `Call` *specifies the atom to call, and* `Annotation` *holds the truth-value. The file provided has the* `shortest_path/3` *predicate written as a GAP over the lattice* `min`*. The join over min is simply the preicate* `min/3`*, and the bottom element the atom* `infinity`*.*

1. *What is the result of the query* `shortest_path(1,5):[min,X]`*? Does* `shortest_path(1,5):[min,4]` *succeed? Why?*

2. *The* Belnap *lattice is a four-valued lattice containing* $\top$*, true, false, and* $\perp$ *such that* $\top > true$*,* $\top > false$*,* $true > \perp$*,* $false > \perp$*. Extend the GAP meta-interpreter so that it can properly handle the belnap lattice and execute programs like:*

   ```
   p:[belnap,true] <-  r:[belnap,true].
   p:[belnap,false] <-  r:[belnap,false].
   r:[belnap,top] <- true.
   ```

   What is the result of the query `p:[belnap,X]`*?*

3. *Extend the meta-interpreter to properly handle the powerset lattice. For the program*

   ```
   q:[set,[a]] <-  true.
   q:[set,[b]] <-  true.
   ```

   What is the result of the query `q:[set,X]`*? Can you think of uses for annotating an atom with a set? What if the sets represent time points in which an atom is true? Or sources (e.g. databases) in which an atom was true?*

References

[1] J. Alferes, C. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[2] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43(1):20–74, January 1996.

[3] C. Crowner, K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic grammars. *Computer Languages*, 1999. To apear.

[4] B. Cui, T. Swift, and D. S. Warren. A case study in using preference logic grammars for knowledge representation. Available at http://www.cs.sunysb.edu/~tswift, 1998.

[5] M. Gelfond and V. Lifshitz. The stable model semantics for logic programming. In *JICSLP*, pages 1070–1080. MIT Press, 1988.

[6] M. Gelfond and V. Lifshitz. Logic programs with classical negation. In *ICLP*, pages 579–597. MIT Press, 1990.

[7] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic programming. In *ICLP*, pages 731–746. MIT Press, 1995.

[8] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Logic Programming*, 12(4):335–368, 1992.

[9] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM PODS*, pages 11–21. ACM Press, 1989.

[10] T.C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[11] K. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. *The XSB Programmer's Manual: version 2.0*, 1999.

[12] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*. To appear.

[13] T. Swift. A new formulation of tabled resolution with delay. In *EPIA 99*, 1999. Available at http://www.cs.sunysb.edu/~tswift.

[14] A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of 8th PODS*, pages 1–10. ACM, 1989.

[15] A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.