Answer Set Programming for the Semantic Web

Tutorial

Thomas Eiter, Roman Schindlauer (TU Wien)
Giovambattista Ianni (TU Wien, Univ. della Calabria)
Axel Polleres (Univ. Rey Juan Carlos, Madrid)

Supported by IST REWERSE, FWF Project P17212-N04, CICyT project TIC-2003-9001-C02.

Unit 2 – ASP Extensions

G. Ianni

Dipartimento di Matematica - Università della Calabria

European Semantic Web Conference 2006

Unit Outline

Logic Programming Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See http://www.tcs.hut.fi/Research/Logic/wasp/wp3/
- Here, we consider some DLV specific extensions.

Logic Programming Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/`
- Here, we consider some DLV specific extensions.

Logic Programming Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See http://www.tcs.hut.fi/Research/Logic/wasp/wp3/
- Here, we consider some DLV specific extensions.

Logic Programming Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/`
- Here, we consider some DLV specific extensions.

Logic Programming Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/`
- Here, we consider some DLV specific extensions.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints "kill" unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
    - Constraints "kill" unwanted models;
    - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints "kill" unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints "kill" unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints "kill" unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
    - Constraints "kill" unwanted models;
    - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Weak Constraints

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints "kill" unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated constraints.
- Such answer sets are called *optimal or best models of* (P, W).
- Other solvers feature similar constructs.

Syntax and Semantics

- Syntax:

$$:\sim\ \ b_1, \cdots, b_k,\ \texttt{not}\ b_{k+1}, \cdots,\ \texttt{not}\ b_m.\quad [\texttt{Weight} : \texttt{Level}]$$

- In the presence of weights, best models minimize the sum of the weights of violated constraints.
- Semantics: minimizes the violation of constraints with highest priority level first; then with the lower priority levels in descending order.
- `Level` part is syntactic sugar, can be compiled into weights.

Syntax and Semantics

- Syntax:

  $$:\sim \quad \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{k},\ \mathtt{not}\ \mathtt{b}_{\mathtt{k}+1}, \cdots,\ \mathtt{not}\ \mathtt{b}_\mathtt{m}. \quad [\mathtt{Weight} : \mathtt{Level}]$$

- In the presence of weights, best models minimize the sum of the weights of violated constraints.
- Semantics: minimizes the violation of constraints with highest priority level first; then with the lower priority levels in descending order.
- `Level` part is syntactic sugar, can be compiled into weights.

Syntax and Semantics

- Syntax:

  $$\mathtt{:\sim}\ \ b_1, \cdots, b_k,\ \mathtt{not}\ b_{k+1}, \cdots,\ \mathtt{not}\ b_m.\quad [\mathtt{Weight} : \mathtt{Level}]$$

- In the presence of weights, best models minimize the sum of the weights of violated constraints.
- Semantics: minimizes the violation of constraints with highest priority level first; then with the lower priority levels in descending order.
- `Level` part is syntactic sugar, can be compiled into weights.

Syntax and Semantics

- Syntax:

  $:\sim\ \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{k},\ \mathtt{not}\ \mathtt{b}_{\mathtt{k}+1}, \cdots,\ \mathtt{not}\ \mathtt{b}_\mathtt{m}.\quad [\mathtt{Weight} : \mathtt{Level}]$
- In the presence of weights, best models minimize the sum of the weights of violated constraints.
- Semantics: minimizes the violation of constraints with highest priority level first; then with the lower priority levels in descending order.
- `Level` part is syntactic sugar, can be compiled into weights.

Weak Constraints: Examples

```
a v b.
c :- b.
:~ a.
:~ b.
:~ c.
```

Best model: a Cost ([Weight:Level]): <[1:1]> Answer set $\{b, c\}$ is discarded because it violates two weak constraints!

Weak Constraints: Examples

```
a v b.
c :- b.
:~ a.
:~ b.
:~ c.
```

Best model: a   Cost ([Weight:Level]): <[1:1]> Answer set $\{b, c\}$ is discarded because it violates two weak constraints!

Weak Constraints: Examples

```
a v b.
c :- b.
:~ a.
:~ b.
:~ c.
```

Best model: a Cost ([Weight:Level]): <[1:1]> Answer set $\{b, c\}$ is discarded because it violates two weak constraints!

Weak Constraints: Examples /2

```
a v b.
:~ a. [1:]    :~ a. [1:]     :~ b. [2:]
```

Best model: b   Cost ([Weight:Level]): <[2:1]>
Best model: a   Cost ([Weight:Level]): <[2:1]>

```
a v b1 v b2.
:~ a. [:1]    :~ b1. [:2]    :~ b2. [:2]
```

Best model: a   Cost ([Weight:Level]): <[1:1],[0:2]>

Weak Constraints: Examples /2

```
a v b.
:~ a. [1:]    :~ a. [1:]    :~ b. [2:]
```

Best model: b   Cost ([Weight:Level]): <[2:1]>
Best model: a   Cost ([Weight:Level]): <[2:1]>

```
a v b1 v b2.
:~ a. [:1]    :~ b1. [:2]    :~ b2. [:2]
```

Best model: a   Cost ([Weight:Level]): <[1:1],[0:2]>

Weak Constraints: Examples /2

```
a v b.
:~ a. [1:]    :~ a. [1:]     :~ b. [2:]
```

Best model: b   Cost ([Weight:Level]): <[2:1]>
Best model: a   Cost ([Weight:Level]): <[2:1]>

```
a v b1 v b2.
:~ a. [:1]    :~ b1. [:2]    :~ b2. [:2]
```

Best model: a   Cost ([Weight:Level]): <[1:1],[0:2]>

Weak Constraints: Examples /2

```
a v b.
:~ a. [1:]    :~ a. [1:]     :~ b. [2:]
```

Best model: b   Cost ([Weight:Level]): <[2:1]>
Best model: a   Cost ([Weight:Level]): <[2:1]>

```
a v b1 v b2.
:~ a. [:1]    :~ b1. [:2]    :~ b2. [:2]
```

Best model: a   Cost ([Weight:Level]): <[1:1],[0:2]>

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y). [:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).    [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).       [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y). [:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y). [:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).    [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).       [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y). [:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).     [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).        [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

A bigger example - Employee Assignment

- Goal: Divide employees in two project groups p_1 and p_2[1].
  1. Skills of group members should be different.
  2. Persons in the same group should not be married to each other.
  3. Members of a group should possibly know each other.
- Requirement 1) is more important than 2) and 3), which are equally important
- Layers express the relative importance of the requirements.

```
assign(X,p1) v assign(X,p2) :-  employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).    [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).       [:1]
:~ assign(X,P), assign(Y,P), X!=Y, not know(X,Y).[:1]
```

[1] Example assignment.dlv

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Guess-Check-Optimize Methodology

- Extend the "Guess & Check" Methodology
- Use weak constraints to filter out best (optimal) solutions

"Guess-Check-Optimize": Divide P into three main parts:

Guessing Part

$G \subseteq P$: $Answer\_Sets(G \cup F_I)$ represent "solution candidates" for instance I.

Checking Part (optional)

$C \subseteq P$: $Answer\_Sets(G \cup C \cup F_I)$ represent the admissible solutions for I.

Optimization Part (optional)

The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function $f : Answer\_Sets(G \cup C \cup F_I) \rightarrow \mathbb{N}$
Those answer sets minimizing f are selected.

Social Dinner III

Task

Now that we have defined `bottleChosen` *as the solution predicate, is there a way to select only the smallest sets of wines? Try to expand wineCover4.dlv*

Social Dinner III

Task

Now that we have defined `bottleChosen` *as the solution predicate, is there a way to select only the smallest sets of wines? Try to expand wineCover4.dlv*

?

Social Dinner III

Task

Now that we have defined `bottleChosen` *as the solution predicate, is there a way to select only the smallest sets of wines? Try to expand wineCover4.dlv*

```
:~ bottleChosen(X). [1:1]
```

Social Dinner III

Task

Now that we have defined `bottleChosen` *as the solution predicate, is there a way to select only the smallest sets of wines? Try to expand wineCover4.dlv*

```
:~ bottleChosen(X). [1:1]
```

Solution available as wineCover5.dlv

Weak Constraints with Weights

- A single weak constraints in some layer n is more important than *all* weak constraints in lower layers ($n-1$, $n-2$,...) *together*!
- Weak constraints are weighted to make finer distinctions among elements of the same priority:
  `:~ G1.[3.5:1]   :~ G2.[4.6:1]`
- The weights of violated weak constraints are summed up for each layer.
- Example: *High School Time Tabling Problem*
  Structural Requirements > Pedagogical Requirements > Personal Wishes

Weak Constraints with Weights

- A single weak constraints in some layer n is more important than *all* weak constraints in lower layers ($n-1$, $n-2$,...) *together*!
- Weak constraints are weighted to make finer distinctions among elements of the same priority:
  ```
  :~ G1.[3.5:1]   :~ G2.[4.6:1]
  ```
- The weights of violated weak constraints are summed up for each layer.
- Example: *High School Time Tabling Problem*
  Structural Requirements > Pedagogical Requirements > Personal Wishes

Weak Constraints with Weights

- A single weak constraints in some layer n is more important than *all* weak constraints in lower layers ($n-1$, $n-2$,...) *together*!
- Weak constraints are weighted to make finer distinctions among elements of the same priority:
  ```
  :~ G1.[3.5:1]   :~ G2.[4.6:1]
  ```
- The weights of violated weak constraints are summed up for each layer.
- Example: *High School Time Tabling Problem*
  Structural Requirements > Pedagogical Requirements > Personal Wishes

Weak Constraints with Weights

- A single weak constraints in some layer n is more important than *all* weak constraints in lower layers ($n-1$, $n-2$,...) *together*!
- Weak constraints are weighted to make finer distinctions among elements of the same priority:
  `:~ G1.[3.5:1]   :~ G2.[4.6:1]`
- The weights of violated weak constraints are summed up for each layer.
- Example: *High School Time Tabling Problem*
  Structural Requirements > Pedagogical Requirements > Personal Wishes

Traveling Salesperson

Given: Weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph.
Task: Find a minimum-cost cycle (closed path) in G starting at a and going through each node in V exactly once[2].

- G stored by facts over predicates `node(X)` and `arc(X,Y)`.
- Starting node a is specified by the predicate `start` (unary).

Guess:

```
inPath(X,Y,C) v outPath(X,Y,C) :- start(X), arc(X,Y,C).
inPath(X,Y,C) v outPath(X,Y,C) :- reached(X), arc(X,Y,C).
reached(X):- inPath(Y,X,C).
```

Check:

```
:- inPath(X,Y,_), inPath(X,Y1,_), Y <> Y1.
:- inPath(X,Y,_), inPath(X1,Y,_), X <> X1.
:- node(X), not reached(X).
```

Optimize:

```
:~ inPath(X,Y,C). [C:1]
```

[2] Example tsp.dlv

Traveling Salesperson

Given: Weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph.
Task: Find a minimum-cost cycle (closed path) in G starting at a and going through each node in V exactly once[2].

- G stored by facts over predicates `node(X)` and `arc(X,Y)`.
- Starting node a is specified by the predicate `start` (unary).

Guess:

```
inPath(X,Y,C) v outPath(X,Y,C) :- start(X), arc(X,Y,C).
inPath(X,Y,C) v outPath(X,Y,C) :- reached(X), arc(X,Y,C).
reached(X):- inPath(Y,X,C).
```

Check:

```
:- inPath(X,Y,_), inPath(X,Y1,_), Y <> Y1.
:- inPath(X,Y,_), inPath(X1,Y,_), X <> X1.
:- node(X), not reached(X).
```

Optimize:

```
:~ inPath(X,Y,C). [C:1]
```

[2] Example tsp.dlv

Traveling Salesperson

Given: Weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph.
Task: Find a minimum-cost cycle (closed path) in G starting at a and going through each node in V exactly once[2].

- G stored by facts over predicates `node(X)` and `arc(X,Y)`.
- Starting node a is specified by the predicate `start` (unary).

Guess:

```
inPath(X,Y,C) v outPath(X,Y,C) :- start(X), arc(X,Y,C).
inPath(X,Y,C) v outPath(X,Y,C) :- reached(X), arc(X,Y,C).
reached(X):- inPath(Y,X,C).
```

Check:

```
:- inPath(X,Y,_), inPath(X,Y1,_), Y <> Y1.
:- inPath(X,Y,_), inPath(X1,Y,_), X <> X1.
:- node(X), not reached(X).
```

Optimize:

```
:~ inPath(X,Y,C). [C:1]
```

[2] Example tsp.dlv

Traveling Salesperson

Given: Weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph.
Task: Find a minimum-cost cycle (closed path) in G starting at a and going through each node in V exactly once[2].

- G stored by facts over predicates `node(X)` and `arc(X,Y)`.
- Starting node a is specified by the predicate `start` (unary).

Guess:

```
inPath(X,Y,C) v outPath(X,Y,C) :- start(X), arc(X,Y,C).
inPath(X,Y,C) v outPath(X,Y,C) :- reached(X), arc(X,Y,C).
reached(X):- inPath(Y,X,C).
```

Check:

```
:- inPath(X,Y,_), inPath(X,Y1,_), Y <> Y1.
:- inPath(X,Y,_), inPath(X1,Y,_), X <> X1.
:- node(X), not reached(X).
```

Optimize:

```
:~ inPath(X,Y,C). [C:1]
```

[2] Example tsp.dlv

Social Dinner IV

Task

Let each wine bootle have a price encoded by `price(bottle,value)`. *Modify* `wineCover5b.dlv` *and try to choose the best cost selection of bottles.*

?

Solution available at `wineCover5c.dlv`

Social Dinner IV

Task

Let each wine bootle have a price encoded by `price(bottle,value)`. *Modify* `wineCover5b.dlv` *and try to choose the best cost selection of bottles.*

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

Solution available at `wineCover5c.dlv`

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                        bought(Good),
                        price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                          bought(Good),
                          price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                        bought(Good),
                        price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                          bought(Good),
                          price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                          bought(Good),
                          price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.
small :- #max{ X : f(A,X,C), b(C,G) } < 3.
ok_price :- 30 <= #sum{ Price :
                          bought(Good),
                          price(Good,Price) } < 50.
```

- other solvers (e.g. Smodels) offer similar constructs (cardinality atoms, weight constraints).

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

  of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).

- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

  where

  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) }`)

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

  of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).

- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

  where

  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) }`)

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

  of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).

- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

  where

  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) }`)

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

  of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).
- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

  where
  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) }`)

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

  of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).
- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

  where
  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) } }`)

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

of a list *Vars* of variables and a list *Conj* of literals (safety required) (e.g. `{ X : f(A,X,C), b(C,G) }`).

- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

where
  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set
    (e.g. `#max{ X : f(A,X,C), b(C,G) } }`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{aligned}
Agg\_Atom ::= \;& val \;\odot\; f\,\{Vars : Conj\} \\
\mid\;& f\,\{Vars : Conj\} \;\odot\; val \\
\mid\;& val_l \;\odot_l f\,\{Vars : Conj\} \;\odot_r\; val_u
\end{aligned}
$$

where

- val, val_l, val_u are constants or variables,
- $\odot \in \{<, >, \leq, \geq, =\}$,
- $\odot_l, \odot_r \in \{<, \leq\}$, and
- $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{aligned}
Agg\_Atom ::= \quad & val \;\; \circledcirc \;\; f\,\{Vars : Conj\} \\
| \quad & f\,\{Vars : Conj\} \;\; \circledcirc \;\; val \\
| \quad & val_l \;\; \circledcirc_l \, f\,\{Vars : Conj\} \;\; \circledcirc_r \;\; val_u
\end{aligned}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{aligned}
Agg\_Atom ::= \;& val \;\; \circledcirc \;\; f\,\{Vars : Conj\} \\
\mid \;& f\,\{Vars : Conj\} \;\; \circledcirc \;\; val \\
\mid \;& val_l \;\; \circledcirc_l\, f\,\{Vars : Conj\} \;\; \circledcirc_r \;\; val_u
\end{aligned}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{array}{rl}
Agg\_Atom ::= & val \;\; \circledcirc \;\; f\,\{Vars : Conj\} \\
\mid & f\,\{Vars : Conj\} \;\; \circledcirc \;\; val \\
\mid & val_l \;\; \circledcirc_l \; f\,\{Vars : Conj\} \;\; \circledcirc_r \;\; val_u
\end{array}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{aligned}
Agg\_Atom ::= \quad & val \;\; \circledcirc \;\; f\,\{Vars : Conj\} \\
| \quad & f\,\{Vars : Conj\} \;\; \circledcirc \;\; val \\
| \quad & val_l \;\; \circledcirc_l \, f\,\{Vars : Conj\} \;\; \circledcirc_r \;\; val_u
\end{aligned}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{array}{rl}
Agg\_Atom ::= & val \ \circledcirc \ f\,\{Vars : Conj\} \\
| & f\,\{Vars : Conj\} \ \circledcirc \ val \\
| & val_l \ \circledcirc_l \, f\,\{Vars : Conj\} \ \circledcirc_r \ val_u
\end{array}
$$

  where

  - val, val_l, val_u are constants or variables,
  - $\circledcirc \in \{<, >, \leq, \geq, =\}$,
  - $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
  - $f\,\{Vars : Conj\}$ is an aggregate function (e.g. `#max{ X : f(A,X,C), b(C,G) } < 3`)

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Aggregate Atoms – Semantics

- Informally:
  Suppose I is an interpretation.
  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.
- Appealing formal definition of semantics is a bit tricky
- Widely acknowledged proposal: Faber et al. [32].

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]
```

```
totalcost(N) :- #int(N),
      ? .
```

Solution at wineCover6.dlv

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]

totalcost(N) :- #int(N),
      ? .
```

Solution at wineCover6.dlv

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]
```

```
totalcost(N) :- #int(N),
      ? .
```

Solution at wineCover6.dlv

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]
```

```
totalcost(N) :- #int(N),
      ? .
```

Solution at wineCover6.dlv

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]

totalcost(N) :- #int(N),
    #sum{ Y : bottleChosen(X),prize(X,Y) } = N.
```

Solution at wineCover6.dlv

Social Dinner V

Task

Modify wineCover5c.dlv so that the weak constraint

```
:~ bottleChosen(X),prize(X,N). [N:1]
```

can be changed in

```
:~ totalcost(N). [N:1]

totalcost(N) :- #int(N),
    #sum{ Y : bottleChosen(X),prize(X,Y) } = N.
```

Solution at wineCover6.dlv

Frame logic: the idea

The molecular syntax typical of F-logic is quite useful for manipulating triple stores and complex join patterns:

Datalog Syntax

```
wineBottle("Brachetto").  isA("Brachetto","RedWine"),
isA("Brachetto","SweetWine").  prize("Brachetto",10).
```

F-Logic Syntax

```
"Brachetto" :  wineBottle[isA->>{"RedWine","SweetWine"},
prize->10].
```

Frame syntax: the idea

The molecular syntax typical of F-logic is quite useful for manipulating triple stores and complex join patterns:

Datalog Syntax

```
mainEntity(M) :- "foaf:PersonalProfileDocument"(X),
                 "foaf:primaryTopic"(X,M).
```

F-Logic Syntax

```
M : mainEntity :-
        X:"foaf:PersonalProfileDocument"["foaf:primaryTopic"->M].
```

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                              ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                                ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                                 ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                              ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                             ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                             ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                            ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                                ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Informal Syntax and Semantics

F-Logic molecule

```
subject : type[predicate1->object, ...,
        predicate2->>{ object1, ..., objectn },
        ...]
```

It is a syntactic shortcut to

Datalog conjunction of facts

```
type(subject),
predicate1(subject,object), ... , predicate2(subject,object1),
                            ... , predicate2(subject, objectn)
```

- Objects can be nested frames (only atomic frames in rules' heads)
- Subjects and Objects unify with terms of the language. Under higher order extensions (see Unit 5), also Predicates and Types do.
- F-Logic semantic features (inheritance, etc.) are not currently implemented, this is only syntactic sugar.

Frame Spaces

A Frame Space directive tells how frames are mapped to regular atoms

```
@triple.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B,triple) :-
        father(A,Y,triple),
        father(B,Y,triple).
```

```
@.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B) :- father(A,Y),
                father(B,Y).
```

Frame Spaces

A Frame Space directive tells how frames are mapped to regular atoms

```
@triple.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B,triple) :-
        father(A,Y,triple),
        father(B,Y,triple).
```

```
@.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B) :- father(A,Y),
                father(B,Y).
```

Frame Spaces

A Frame Space directive tells how frames are mapped to regular atoms

```
@triple.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B,triple) :-
        father(A,Y,triple),
        father(B,Y,triple).
```

```
@.
A[brother->B] :- A[father->Y],
                 B[father->Y].
```

Maps to:

```
brother(A,B) :- father(A,Y),
                father(B,Y).
```

Social Dinner VII

Task

Take wineCover7a.dlt. It is partially in frame syntax. Put the following rule in frame logic syntax:

```
compliantBottle(X,Z) :- preferredWine(X,Y), isA(Z,Y).
```

Solution at wineCover7b.dlt

Social Dinner VII

Task

Take wineCover7a.dlt. It is partially in frame syntax. Put the following rule in frame logic syntax:

```
X[compliantBottle->Z] :- X[preferredWine->Y], Z[isA->Y].
```

Solution at wineCover7b.dlt

Social Dinner VII

Task

Take wineCover7a.dlt. It is partially in frame syntax. Put the following rule in frame logic syntax:

```
X[compliantBottle->Z] :- X[preferredWine->Y], Z[isA->Y].
```

Solution at wineCover7b.dlt

The idea of templates

Imagine you want to encode all the possible permutations of a given predicate p (assume $maxint = |X : p(X)|$)

First, I guess worlds of permutations

```
permutation(X,N) v -permutation(X,N) :- p(X),#int(N).
```

Then, I cut worlds I don't like

```
:- permutation(X,A),permutation(Z,A), Z <> X.
:- permutation(X,A),permutation(X,B), A <> B.
```

Also, each element must be in the partition

```
covered(X) :- permutation(X,A).
:- p(X), not covered(X).
```

The idea of templates

Imagine you want to encode all the possible permutations of a given predicate p (assume $maxint = |X : p(X)|$)

First, I guess worlds of permutations

```
permutation(X,N) v -permutation(X,N) :- p(X),#int(N).
```

Then, I cut worlds I don't like

```
:- permutation(X,A),permutation(Z,A), Z <> X.
:- permutation(X,A),permutation(X,B), A <> B.
```

Also, each element must be in the partition

```
covered(X) :- permutation(X,A).
:- p(X), not covered(X).
```

The idea of templates

Imagine you want to encode all the possible permutations of a given predicate p (assume $maxint = |X : p(X)|$)

First, I guess worlds of permutations

```
permutation(X,N) v -permutation(X,N) :- p(X),#int(N).
```

Then, I cut worlds I don't like

```
:- permutation(X,A),permutation(Z,A), Z <> X.
:- permutation(X,A),permutation(X,B), A <> B.
```

Also, each element must be in the partition

```
covered(X) :- permutation(X,A).
:- p(X), not covered(X).
```

The idea of templates

Imagine you want to encode all the possible permutations of a given predicate p (assume $maxint = |X : p(X)|$)

First, I guess worlds of permutations

```
permutation(X,N) v -permutation(X,N) :- p(X),#int(N).
```

Then, I cut worlds I don't like

```
:- permutation(X,A),permutation(Z,A), Z <> X.
:- permutation(X,A),permutation(X,B), A <> B.
```

Also, each element must be in the partition

```
covered(X) :- permutation(X,A).
:- p(X), not covered(X).
```

The idea of templates - 2

- Thus, this “small” program encodes a search space of permutations
- But it can be reused and put in a library (let *maxint* big enough here)

```
#template permutation{p(1)}(2)
{
    permutation(X,N) v -permutation(X,N)
      :- p(X),#int(N),
         #count{ Y : p(Y) } = N1,
         N <= N1, N > 0.
    :- permutation(X,A),permutation(Z,A), Z <> X.
    :- permutation(X,A),permutation(X,B), A <> B.
    covered(X) :- permutation(X,A).
    :- p(X), not covered(X).
}
```

The idea of templates - 2

- Thus, this "small" program encodes a search space of permutations
- But it can be reused and put in a library (let *maxint* big enough here)

```
#template permutation{p(1)}(2)
{
    permutation(X,N) v -permutation(X,N)
      :- p(X),#int(N),
         #count{ Y : p(Y) } = N1,
         N <= N1, N > 0.
    :- permutation(X,A),permutation(Z,A), Z <> X.
    :- permutation(X,A),permutation(X,B), A <> B.
    covered(X) :- permutation(X,A).
    :- p(X), not covered(X).
}
```

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                   closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- e(2), p(1) = formal parameter list
- ..}(2), ..}(1) = output predicate arities
- closure, max = output predicate names
- exceeded = local predicate name

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                  closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- `e(2), p(1)` = formal parameter list
- `..}(2), ..}(1)` = output predicate arities
- `closure, max` = output predicate names
- `exceeded` = local predicate name

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                   closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- `e(2), p(1)` = formal parameter list
- `..}(2), ..}(1)` = output predicate arities
- `closure, max` = output predicate names
- `exceeded` = local predicate name

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                   closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- `e(2), p(1)` = formal parameter list
- `..}(2), ..}(1)` = output predicate arities
- `closure, max` = output predicate names
- `exceeded` = local predicate name

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                   closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- `e(2), p(1)` = formal parameter list
- `..}(2), ..}(1)` = output predicate arities
- `closure, max` = output predicate names
- `exceeded` = local predicate name

Syntax and Semantics

Template definition:

```
#template closure{e(2)}(2)
{
   closure(X,Y) :- e(X,Y).
   closure(X,Y) :- e(X,Z),
                  closure(Z,Y).
}
```

```
#template max{p(1)}(1)
{
   exceeded(X) :- p(X),p(Y), Y > X.
   max(X) :- p(X),
             not exceeded(X).
}
```

- `e(2), p(1)` = formal parameter list
- `..}(2), ..}(1)` = output predicate arities
- `closure, max` = output predicate names
- `exceeded` = local predicate name

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- `*` = input terms
- `$` = projection terms
- `S` = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

Syntax and Semantics - 2

Template atoms:

```
clo(X,Y) :- closure{ edge(*,*) }(X,Y).
inPath(X,N) :- permutation{ clo(*,$) }(X,N).
maxAgePerSex(S,A) :- max{ person($,S,*) }(A).
```

- `edge(*,*)`, `clo(*,$)`, `person($,S,*)` = actual parameters
- `closure{ edge(*,*) }(X,Y)` = a template atom
- $*$ = input terms
- \$ = projection terms
- S = group-by (quantification) term
- `(X,Y)`, `(X,N)`, `S..A` = output terms

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- `*` = parameter
- `$` = projection

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- * = parameter
- $ = projection

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- `* = parameter`
- `$ = projection`

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- `*` = parameter
- `$` = projection

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- `* = parameter`
- `$ = projection`

The Hamiltonian Path problem

HP: find a path between nodes of a graph s.t. I cross each node exactly once. (permutation.dlt)

If I want to encode the HP problem with templates, I can do this way:

```
path(X,N) :- permutation{node(*)}(X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

Also, I can use permutation taking input predicates other than unary:

```
path(X,N) :- permutation{edge(*,$)}(X,N).
```

- `*` = parameter
- `$` = projection

Social Dinner VIII

Task

Try to expand wineCover7.dlt: define a template `subset` *for specifying the search space of minimum cardinality subsets of wines.*

```
#template subset{ p(1) }(1)
{
    ?
    ?
}
bottleChosen(X) :- ?
```

Solution at `wineCover8.dlt`

Social Dinner VIII

Task

Try to expand wineCover7.dlt: define a template `subset` *for specifying the search space of minimum cardinality subsets of wines.*

```
#template subset{ p(1) }(1)
{
    ?
    ?
}
bottleChosen(X) :- ?
```

Solution at `wineCover8.dlt`

Social Dinner VIII

Task

Try to expand wineCover7.dlt: define a template `subset` *for specifying the search space of minimum cardinality subsets of wines.*

```
#template subset{ p(1) }(1)
{
    subset(X) v nonsubset(X) :- p(X).
    :~ subset(X). [1:1]
}
bottleChosen(X) :- subset{compliantBottle($,*)}(X).
```

Solution at `wineCover8.dlt`

Social Dinner VIII

Task

Try to expand wineCover7.dlt: define a template `subset` *for specifying the search space of minimum cardinality subsets of wines.*

```
#template subset{ p(1) }(1)
{
    subset(X) v nonsubset(X) :- p(X).
    :~ subset(X). [1:1]
}
bottleChosen(X) :- subset{compliantBottle($,*)}(X).
```

Solution at `wineCover8.dlt`

References

1. **Weak Constraints**: [11]
2. **Aggregates**: [32]
3. **Templates**: [13]
4. **Frame Logic**: [48]
5. **Other extensions**:
   `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/`