Tutorial: logic programming

Markus Krötzsch

Institute AIFB, Universität Karlsruhe
mak@aifb.uni-karlsruhe.de
Outline

1 Basics

2 Propositional logic programs

3 Definite logic programs

4 Normal programs

5 F-Logic

6 Summary
A (logic programming) rule is a formula of the form

$$B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m$$

- $B_i$ and $H_j$ are usually **literals**, i.e. negated or non-negated logical atoms
A (logic programming) rule is a formula of the form

\[ B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m \]

- \(B_i\) and \(H_j\) are usually literals, i.e. negated or non-negated logical atoms
- Disjunctions in the body are not needed:

\[ A \lor B \rightarrow C \quad \Rightarrow \quad A \rightarrow C \]
\[ B \rightarrow C \]
A (logic programming) rule is a formula of the form

\[ B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m \]

\( B_i \) and \( H_j \) are usually literals, i.e. negated or non-negated logical atoms

Disjunctions in the body are not needed:

\[ A \lor B \rightarrow C \] \quad \Rightarrow \quad \begin{align*}
A &\rightarrow C \\
B &\rightarrow C
\end{align*}

Conjunctions in the head are not needed:

\[ A \rightarrow B \land C \] \quad \Rightarrow \quad \begin{align*}
A &\rightarrow B \\
A &\rightarrow C
\end{align*}

\[ \sim \text{ Lloyd-Topor translations} \]
In classical logic, implications are disjunctions:

\[ B \rightarrow H \equiv \neg B \lor H \]

For a rule, we obtain:

\[ B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m \]
In classical logic, implications are disjunctions:

\[ B \rightarrow H \equiv \neg B \lor H \]

For a rule, we obtain:

\[ B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m \]
\[ \equiv \neg (B_1 \land B_2 \land \ldots \land B_n) \lor H_1 \lor H_2 \lor \ldots \lor H_m \]
In classical logic, implications are disjunctions:

\[ B \rightarrow H \equiv \neg B \lor H \]

For a rule, we obtain:

\[ B_1 \land B_2 \land \ldots \land B_n \rightarrow H_1 \lor H_2 \lor \ldots \lor H_m \equiv \neg(B_1 \land B_2 \land \ldots \land B_n) \lor H_1 \lor H_2 \lor \ldots \lor H_m \equiv \neg B_1 \lor \neg B_2 \lor \ldots \lor \neg B_n \lor H_1 \lor H_2 \lor \ldots \lor H_m \]

Disjunctions of literals are called clauses

\( \leadsto \) rules as clauses
Outline

1. Basics
2. Propositional logic programs
3. Definite logic programs
4. Normal programs
5. F-Logic
6. Summary
Propositional logic programs

A propositional logic program . . .

is a finite set of rules which are

- **propositional**: only propositional atoms,
- **definite**: exactly one atom in the head.

General form: $B_1 \land B_2 \land \ldots \land B_n \rightarrow H$

Common syntax: $H :\leftarrow B_1, B_2, \ldots, B_n$

**Fact**: rule with empty body ($\rightarrow$ body vacuously true)
Example

```
a
b
c :- a
e :- b, d
e :- b, c
e :- a, b, e```

Markus Krötzsch (AIFB Karlsruhe)
Semantics of propositional programs

Standard semantics of propositional logic:

An interpretation $\mathcal{I}$ . . .

is a mapping $\cdot^\mathcal{I}$ from formulae to \{t, f\}, such that

- $(A \land B)^\mathcal{I} = t$ iff $A^\mathcal{I} = t$ and $B^\mathcal{I} = t$,
- $(A \lor B)^\mathcal{I} = t$ iff $A^\mathcal{I} = t$ or $B^\mathcal{I} = t$,
- $\neg A^\mathcal{I} = t$ iff $A^\mathcal{I} = f$.

We can view interpretations as sets of (true) atoms
Semantics of propositional programs

Standard semantics of propositional logic:

An interpretation $\mathcal{I}$ . . .

is a mapping $\cdot^\mathcal{I}$ from formulae to $\{t, f\}$, such that

- $(A \land B)^\mathcal{I} = t$ iff $A^\mathcal{I} = t$ and $B^\mathcal{I} = t$,
- $(A \lor B)^\mathcal{I} = t$ iff $A^\mathcal{I} = t$ or $B^\mathcal{I} = t$,
- $\neg A^\mathcal{I} = t$ iff $A^\mathcal{I} = f$.

$\leadsto$ We can view interpretations as sets of (true) atoms

Take a set of formulae $\mathcal{F}$ (e.g. a propositional program).

- $\mathcal{I}$ is a model of $\mathcal{F}$ if $F^\mathcal{I} = t$ for all $F \in \mathcal{F}$.
- A formula $G$ is a consequence of $\mathcal{F}$ if $G^\mathcal{I} = t$ for all models $\mathcal{I}$ of $\mathcal{F}$.
Bottom-up computation

- How can we compute the above semantics?
- Observation: every rule has exactly one consequence
- Idea: compute all consequences bottom-up, step by step
How can we compute the above semantics?
Observation: every rule has exactly one consequence
Idea: compute all consequences bottom-up, step by step

The $T_P$ operator

Consider a program $P$. The function $T_P$ maps interpretations to interpretations:

$$T_P(I) = \{ H \mid P \text{ contains a rule } H : \leftarrow B_1, \ldots, B_n \text{ such that } B_i \in I \}$$
Bottom-up computation

- How can we compute the above semantics?
- Observation: every rule has exactly one consequence
- Idea: compute all consequences bottom-up, step by step

The $T_P$ operator

Consider a program $P$. The function $T_P$ maps interpretations to interpretations:

$$T_P(I) = \{ H \mid P \text{ contains a rule } H :- B_1, \ldots, B_n \text{ such that } B_i \in I \}$$

- Step-wise computation:

$$T_P^0 = \emptyset \quad T_P^{i+1} = T_P(T_P^i)$$
Example: $P =$

\[
\begin{align*}
  a & \quad c :\quad a \\
  b & \quad e :\quad b, d \\
  & \quad e :\quad a, b, e
\end{align*}
\]

$\leadsto T^0_P = \emptyset$
Example: $P =$

\[ \begin{align*}
 a & \quad c :\quad a & \quad e :\quad b, c \\
 b & \quad e :\quad b, d & \quad e :\quad a, b, e \\
\end{align*} \]

$\leadsto T^0_P = \emptyset$, $T^1_P = \{a, b\}$
Example: $P =$

\[
\begin{align*}
& a \quad c :\!-\! a \quad e :\!-\! b, c \\
& b \quad e :\!-\! b, d \quad e :\!-\! a, b, e
\end{align*}
\]

$\leadsto T^0_P = \emptyset$, $T^1_P = \{a, b\}$, $T^2_P = \{a, b, c\}$
Example: $P =$

\[
\begin{align*}
& a \quad c \leftarrow a \\
& b \quad e \leftarrow b, d \\
& & e \leftarrow a, b, e
\end{align*}
\]

$\leadsto T^0_P = \emptyset, T^1_P = \{a, b\}, T^2_P = \{a, b, c\}, T^3_P = \{a, b, c, e\}$
Example: \( P = \)

\[
\begin{align*}
  a & \quad c \quad :\!- \quad a \\
  b & \quad e \quad :\!- \quad b, \, d \\
  & \quad e \quad :\!- \quad a, \, b, \, e
\end{align*}
\]

\( \leadsto T^0_P = \emptyset, \ T^1_P = \{a, \, b\}, \ T^2_P = \{a, \, b, \, c\}, \ T^3_P = \{a, \, b, \, c, \, e\} \)

\( \leadsto T^4_P = \{a, \, b, \, c, \, e\} = T^5_P = T^6_P = \ldots \)
Example: \( P = \)

| \( a \) | \( c \) :\( \leftarrow \) \( a \) | \( e \) :\( \leftarrow \) \( b, c \) |
| \( b \) | \( e \) :\( \leftarrow \) \( b, d \) | \( e \) :\( \leftarrow \) \( a, b, e \) |

\( \leadsto T_0^P = \emptyset \), \( T_1^P = \{a, b\} \), \( T_2^P = \{a, b, c\} \), \( T_3^P = \{a, b, c, e\} \)

\( \leadsto T_4^P = \{a, b, c, e\} = T_5^P = T_6^P = \ldots \)

\( I \) is a fixed point of \( T_P \) if \( T_P(I) = I \)

\( T_P^\infty = \bigcup_{i \geq 0} T_i^P \) is always the least fixed point and the least model
Goal-directed computation

- Computing all consequences is not efficient.
- How can we directly check whether some consequence holds?
Goal-directed computation

- Computing all consequences is not efficient.
- How can we directly check whether some consequence holds?

Reasoning by contradiction

$F$ is a consequence of $P$ iff $\neg F \land P$ has no models (i.e. is unsatisfiable).
Computing all consequences is not efficient.

How can we directly check whether some consequence holds?

**Reasoning by contradiction**

\( F \) is a consequence of \( P \) iff \( \neg F \land P \) has no models (i.e. is unsatisfiable).

**Strategy:**

- **Question:** does \( Q_1 \land Q_2 \land \ldots \land Q_n \) follow from \( P \)?
- **Step 1:** negate query \( \neg (Q_1 \land \ldots \land Q_n) \)
- **Step 2:** check whether \( P \land \neg (Q_1 \land \ldots \land Q_n) \) is satisfiable

\( \rightarrow \) A goal is a formula of the form \( \neg (Q_1 \land \ldots \land Q_n) \)

\( \rightarrow \) Goals can be written as := \( Q_1, \ldots, Q_n \)
Resolution

How can we check unsatisfiability?

\[ \rightarrow \text{Check whether a contradictory statements follows!} \]
Resolution

- How can we check unsatisfiability?
  \[ \sim \text{Check whether a contradictory statements follows!} \]
- What is a contradictory statement?
Resolution

- How can we check unsatisfiability?
  \[ \sim \] Check whether a contradictory statement follows!

- What is a contradictory statement?
  Recall: an implication is a disjunction
  Recall: the empty implication is always false
How can we check unsatisfiability? 
\[\sim \Rightarrow\text{Check whether a contradictory statements follows!}\]

What is a contradictory statement? 
Recall: an implication is a disjunction 
Recall: the empty implication is always false 
\[\sim \Rightarrow\text{the one and only contradictory rule is “} \rightarrow \text{”}\]

How to compute logic consequences of a program?
How can we check unsatisfiability?
~ \rightarrow \text{Check whether a contradictory statements follows!}

What is a contradictory statement?
Recall: an implication is a disjunction
Recall: the empty implication is always false
~ \rightarrow \text{the one and only contradictory rule is “→”}

How to compute logic consequences of a program? ~ Resolution

The resolution rule:

\[
\begin{array}{c}
A \rightarrow B \\
B \rightarrow C
\end{array}
\]

\[
A \rightarrow C
\]
How can we check unsatisfiability?
\( \sim \) Check whether a contradictory statements follows!

What is a contradictory statement?
Recall: an implication is a disjunction
Recall: the empty implication is always false
\( \sim \) the one and only contradictory rule is “\( \rightarrow \)”

How to compute logic consequences of a program? \( \sim \) Resolution

The resolution rule:

\[
\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}
\]

More generally:

\[
\frac{A_1 \land \ldots \land A_n \rightarrow B_i \quad B_1 \land \ldots \land B_i \land \ldots \land B_m \rightarrow C}{A_1 \land \ldots \land A_n \land B_1 \land \ldots \land B_{i-1} \land B_{i+1} \land \ldots \land B_m \rightarrow C}
\]
Fact: only resolutions steps with a goal as a main premise are needed:

\[
A_1 \land \ldots \land A_n \rightarrow B_i \quad B_1 \land \ldots \land B_i \land \ldots \land B_m \rightarrow
\]

\[
A_1 \land \ldots \land A_n \land B_1 \land \ldots \land B_{i-1} \land B_{i+1} \land \ldots \land B_m \rightarrow
\]

\[\leadsto\] linear resolution: apply rules only to the latest goal

Fact: selection of \( B_i \) is “don’t care” nondeterministic \( \leadsto \) any selection rule can be chosen
Fact: only resolutions steps with a goal as a main premise are needed:

\[ A_1 \land \ldots \land A_n \rightarrow B_i \quad B_1 \land \ldots \land B_i \land \ldots \land B_m \rightarrow \]

\[ A_1 \land \ldots \land A_n \land B_1 \land \ldots \land B_{i-1} \land B_{i+1} \land \ldots \land B_m \rightarrow \]

\[ \leadsto \text{linear resolution: apply rules only to the latest goal} \]

Fact: selection of \( B_i \) is “don’t care” nondeterministic \( \leadsto \) any selection rule can be chosen

**SLD-resolution**

*Selection-driven, Linear resolution for Definite clauses (a “folk acronym”).*
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic $\leadsto$ backtracking needed

(1) $a$
(2) $b$
(3) $c \leftarrow a$
(4) $e \leftarrow b, d$
(5) $e \leftarrow b, c$
(6) $e \leftarrow a, b, e$

$\vdash e$
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic

\[ \leadsto \text{backtracking needed} \]

(1) {a} (3) {c := a} (5) {e := b, c}
(2) {b} (4) {e := b, d} (6) {e := a, b, e}

\[ \begin{align*}
\text{:- } & e \\
\text{:- } & b, d
\end{align*} \]
Choice of program rule to resolve upon is “don’t known” nondeterministic
\[ \leadsto \text{backtracking needed} \]

(1) \( a \)  \hspace{1cm} (3) \( c :\leftarrow a \)  \hspace{1cm} (5) \( e :\leftarrow b, c \)

(2) \( b \)  \hspace{1cm} (4) \( e :\leftarrow b, d \)  \hspace{1cm} (6) \( e :\leftarrow a, b, e \)

\[ :- e \]

\[ :- b, d \]

\[ :- d \]
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic
\[ \leadsto \text{backtracking needed} \]

(1) \( a \) \hspace{1cm} (3) \( c :\!\!: a \) \hspace{1cm} (5) \( e :\!\!: b, c \)
(2) \( b \) \hspace{1cm} (4) \( e :\!\!: b, d \) \hspace{1cm} (6) \( e :\!\!: a, b, e \)

\[ \begin{align*}
\text{\( e \)} & \rightarrow (4) \\
\text{\( b, d \)} & \rightarrow (2) \\
\text{\( d \)} & \\
\text{\( b, c \)} & \rightarrow (5) \\
\end{align*} \]
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic
\( \leadsto \) backtracking needed

\[
\begin{align*}
(1) & \ a & & (3) & c & : \ a & & (5) & e & : \ b, c \\
(2) & \ b & & (4) & e & : \ b, d & & (6) & e & : \ a, b, e
\end{align*}
\]
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic

\[ \leadsto \text{backtracking needed} \]

\[
\begin{align*}
(1) & \quad a \\
(2) & \quad b \\
(3) & \quad c \leftarrow a \\
(4) & \quad e \leftarrow b, d \\
(5) & \quad e \leftarrow b, c \\
(6) & \quad e \leftarrow a, b, e
\end{align*}
\]
Choice of program rule to resolve upon is “don’t known” nondeterministic
\[ \leadsto \text{backtracking needed} \]

(1) \( a \) \hspace{1cm} (3) \( c \leftarrow a \) \hspace{1cm} (5) \( e \leftarrow b, c \)

(2) \( b \) \hspace{1cm} (4) \( e \leftarrow b, d \) \hspace{1cm} (6) \( e \leftarrow a, b, e \)
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic $\Rightarrow$ backtracking needed

(1) $a$
(2) $b$
(3) $c \leftarrow a$
(4) $e \leftarrow b, d$
(5) $e \leftarrow b, c$
(6) $e \leftarrow a, b, e$

```
(1) a
(2) b
(3) c :- a
(4) e :- b, d
(5) e :- b, c
(6) e :- a, b, e
```

```
::- e
     /   |
(4)   (5) (6)
::- b, d  :- b, c  :- a, b, e
     |
(2)  |
::- d  :- c
     |
(3)  |
::- a
     |
(1)  □
```
SLD-trees and backtracking

Choice of program rule to resolve upon is “don’t known” nondeterministic \( \leadsto \) backtracking needed

\[
\begin{align*}
(1) \quad & a \\
(2) \quad & b \\
(3) \quad & c \leftarrow a \\
(4) \quad & e \leftarrow b, d \\
(5) \quad & e \leftarrow b, c \\
(6) \quad & e \leftarrow a, b, e
\end{align*}
\]

\[
\begin{array}{c}
\vdash e \\
\downarrow \quad (4) \\
\vdash b, d \\
\downarrow \quad (2) \\
\vdash d \\
\downarrow \quad (3) \\
\vdash a \\
\downarrow \quad (1) \\
\square
\end{array}
\quad
\begin{array}{c}
\vdash e \\
\downarrow \quad (5) \\
\vdash b, c \\
\downarrow \quad (2) \\
\vdash c \\
\downarrow \quad (3) \\
\vdash b, e \\
\downarrow \quad (2) \\
\vdash e
\end{array}
\quad
\begin{array}{c}
\vdash e \\
\downarrow \quad (6) \\
\vdash a, b, e
\end{array}
\]
Choice of program rule to resolve upon is “don’t known” nondeterministic

\[ \leadsto \text{backtracking needed} \]

(1) \( a \) \hspace{1cm} (3) \( c \leftarrow a \) \hspace{1cm} (5) \( e \leftarrow b, c \)

(2) \( b \) \hspace{1cm} (4) \( e \leftarrow b, d \) \hspace{1cm} (6) \( e \leftarrow a, b, e \)
Computation with propositional logic programs can be done in P (which is easy to see from the computation of $T_P$).
Outline

1 Basics
2 Propositional logic programs
3 Definite logic programs
4 Normal programs
5 F-Logic
6 Summary
Definite logic programs

Instead of propositional atoms, we have a first order language:
- predicate symbols with some arity, e.g. $R$
- constants, e.g. $a, b$
- function symbols with some arity, e.g. $f, g$
- variables, e.g. $X, Y$

A term is any well-formed expression built from constants, functions, and variables. Terms without variables are called ground.

An atom is an expression $R(t_1, \ldots, t_n)$ with $R$ an $n$-ary predicate and $t_1, \ldots, t_n$ terms

A definite rule is a rule of the form

$$H :~ B_1, \ldots, B_m$$

with $H, B_1, \ldots, B_m$ atoms, where $m$ might be 0.
What do variables mean?

Variables are universally quantified on rule-level.

- Example: uncle(X, Z) :- brother(X, Y), parent(Y, Z)
- Rules with variables can be read as “macros” for rules obtained by uniformly replacing all variables with ground terms
- More generally: variables can be uniformly replaced by any term $\leadsto$ substitution
Example

\[
\begin{align*}
&\text{number}(0) \\
&\text{number}(s(X)) \leftarrow \text{number}(X) \\
&\text{add}(0, Y, Y) \\
&\text{add}(s(X), Y, Z) \leftarrow \text{add}(X, s(Y), Z) \\
&\text{mul}(0, Y, 0) \\
&\text{mul}(s(X), Y, Z) \leftarrow \text{mul}(X, Y, Z'), \text{add}(Y, Z', Z)
\end{align*}
\]

Example conclusions:
\[
\text{number}(s(s(s(0)))), \text{add}(s(s((0)), s(0), s(s(s(0)))), \ldots
\]
Example

\[
\begin{align*}
\text{number}(0) \\
\text{number}(s(X)) & \leftarrow \text{number}(X) \\
\text{add}(0, Y, Y) \\
\text{add}(s(X), Y, Z) & \leftarrow \text{add}(X, s(Y), Z) \\
\text{mul}(0, Y, 0) \\
\text{mul}(s(X), Y, Z) & \leftarrow \text{mul}(X, Y, Z'), \text{add}(Y, Z', Z)
\end{align*}
\]

Example conclusions:
\[
\text{number}(s(s(s(0)))), \text{add}(s(s((0)), s(0), s(s(s(0))))) , \ldots
\]

\[\leadsto\] Definite programs can encode infinite domains (they can also enforce them, though not in the above example).
Definite logic program as abbreviation for infinite propositional program
Semantics

- Definite logic program as abbreviation for infinite propositional program
- Example:

  \[
  \text{number}(s(X)) \leftarrow \text{number}(X) \\
  \sim \\
  \text{number}(s(0)) \leftarrow \text{number}(0) \\
  \text{number}(s(s(0))) \leftarrow \text{number}(s(0)) \\
  \text{number}(s(s(s(0)))) \leftarrow \text{number}(s(s(0)))
  \ldots
  \]
Semantics

- Definite logic program as abbreviation for infinite propositional program
- Example:

\[
\begin{align*}
\text{number}(s(X)) & :\sim \text{number}(X) \\
\sim & \\
\text{number}(s(0)) & :\sim \text{number}(0) \\
\text{number}(s(s(0))) & :\sim \text{number}(s(0)) \\
\text{number}(s(s(s(0)))) & :\sim \text{number}(s(s(0))) \\
\ldots
\end{align*}
\]

- ground atoms instead of propositional atoms
- interpretations as sets of (true) ground atoms
  \(\sim\) Herbrand interpretation
- \(T_P\) operator defined as before (on the infinite set of ground rules)
Goal-directed computation

How can we practically compute consequences?
How can we practically compute consequences?

$\Rightarrow$ Compute (partially) ground rules “on demand”

$\Rightarrow$ Two literals represent overlapping sets of ground atoms if they can be unified.
Goal-directed computation

- How can we practically compute consequences?
  - $\leadsto$ Compute (partially) ground rules “on demand”
  - $\leadsto$ Two literals represent overlapping sets of ground atoms if they can be unified.

- Example: $R(X, f(a))$ and $R(g(Z), W)$ are unifiable
Goal-directed computation

- How can we practically compute consequences?
  - Compute (partially) ground rules “on demand”
  - Two literals represent overlapping sets of ground atoms if they can be unified.

- Example: $R(X, f(a))$ and $R(g(Z), W)$ are unifiable
  - $R(g(Z), f(a))$ represents all ground instances both agree on
  - substitution $\sigma = \{X \mapsto g(Z), W \mapsto f(a)\}$
Goal-directed computation

- How can we practically compute consequences?
  \( \leadsto \) Compute (partially) ground rules “on demand”
  \( \leadsto \) Two literals represent overlapping sets of ground atoms if they can be unified.

Example: \( R(X, f(a)) \) and \( R(g(Z), W) \) are unifiable
\( \leadsto \) \( R(g(Z), f(a)) \) represents all ground instances both agree on
\( \leadsto \) substitution \( \sigma = \{X \mapsto g(Z), W \mapsto f(a)\} \)

Resolution on non-ground rules:

\[
A \rightarrow B \quad B' \rightarrow C \\
\hline
(A \rightarrow C)\sigma
\]

if \( B\sigma = B'\sigma \)

- propositional SLD-resolution can be lifted
Example: SLD resolution for definite programs

(1) \text{add}(0, Y, Y)
(2) \text{add}(s(X), Y, Z) \leftarrow \text{add}(X, s(Y), Z)

Goal: \leftarrow \text{add}(s(0), s(0), W)
Example: SLD resolution for definite programs

(1) \text{add}(0, Y, Y)

(2) \text{add}(s(X), Y, Z) \Leftarrow \text{add}(X, s(Y), Z)

Goal: \Leftarrow \text{add}(s(0), s(0), W)

\Leftarrow \text{add}(s(0), s(0), W)
Example: SLD resolution for definite programs

(1) \text{add}(0, Y, Y)
(2) \text{add}(s(X), Y, Z) :- \text{add}(X, s(Y), Z)

Goal: :- \text{add}(s(0), s(0), W)

:- \text{add}(s(0), s(0), W) + (2) \quad \sigma_1 = \{ X_1 \mapsto 0, Y_1 \mapsto s(0), Z_1 \mapsto W \}
Example: SLD resolution for definite programs

(1) \( \text{add}(0, Y, Y) \)
(2) \( \text{add}(s(X), Y, Z) :\neg \text{add}(X, s(Y), Z) \)

Goal: \( :\neg \text{add}(s(0), s(0), W) \)

\( :\neg \text{add}(s(0), s(0), W) + (2) \quad \sigma_1 = \{ X_1 \mapsto 0, Y_1 \mapsto s(0), Z_1 \mapsto W \} \)

\( \Downarrow \)

\( :\neg \text{add}(0, s(s(0)), W) \)
Example: SLD resolution for definite programs

(1) \text{add}(0, Y, Y)
(2) \text{add}(s(X), Y, Z) \leftarrow \text{add}(X, s(Y), Z)

Goal: \leftarrow \text{add}(s(0), s(0), W)

\leftarrow \text{add}(s(0), s(0), W) + (2) \quad \sigma_1 = \{X_1 \mapsto 0, Y_1 \mapsto s(0), Z_1 \mapsto W\}

\downarrow

\leftarrow \text{add}(0, s(s(0)), W) + (1) \quad \sigma_2 = \{Y_2 \mapsto s(s(0)), W \mapsto s(s(0))\}
Example: SLD resolution for definite programs

(1) add(0, Y, Y)
(2) add(s(X), Y, Z) :- add(X, s(Y), Z)

Goal: :- add(s(0), s(0), W)

:- add(s(0), s(0), W) + (2) \quad \sigma_1 = \{ X_1 \mapsto 0, Y_1 \mapsto s(0), Z_1 \mapsto W \}

\Downarrow

:- add(0, s(s(0)), W) + (1) \quad \sigma_2 = \{ Y_2 \mapsto s(s(0)), W \mapsto s(s(0)) \}

\Downarrow

\square
Example: SLD resolution for definite programs

(1) \text{add}(0, Y, Y)
(2) \text{add}(\text{s}(X), Y, Z) \leftarrow \text{add}(X, \text{s}(Y), Z)

Goal: \leftarrow \text{add}(\text{s}(0), \text{s}(0), W)

\leftarrow \text{add}(\text{s}(0), \text{s}(0), W) + (2) \quad \sigma_1 = \{X_1 \mapsto 0, Y_1 \mapsto \text{s}(0), Z_1 \mapsto W\}
\downarrow
\leftarrow \text{add}(0, \text{s}(\text{s}(0)), W) + (1) \quad \sigma_2 = \{Y_2 \mapsto \text{s}(\text{s}(0)), W \mapsto \text{s}(\text{s}(0))\}
\downarrow
\square

Result: W\sigma_1\sigma_2 = \text{s}(\text{s}(0))
How hard is this computation?

- Extend the earlier example:
  \[ \text{pow}(X, 0, s(0)) \]
  \[ \text{pow}(X, s(Y), Z) \leftarrow \text{pow}(X, Y, Z'), \text{mul}(X, Z', Z) \]

- Goal: \[ \leftarrow \text{pow}(X, s(s(s(N))), XP), \text{pow}(Y, s(s(s(N))), YP), \]
  \[ \text{pow}(Z, s(s(s(N))), ZP), \text{add}(XP, YP, ZP) \]
How hard is this computation?

- Extend the earlier example:
  \[
  \text{pow}(X, 0, s(0)) \\
  \text{pow}(X, s(Y), Z) \ :- \ \text{pow}(X, Y, Z'), \text{mul}(X, Z', Z)
  \]

- Goal: \(- \ \text{pow}(X, s(s(s(N))), XP), \text{pow}(Y, s(s(s(N))), YP), \text{pow}(Z, s(s(s(N))), ZP), \text{add}(XP, YP, ZP)\)

- In words:
  “Which natural numbers satisfy the equation \(x^n + y^n = z^n\), for \(n > 2\)?”
How hard is this computation?

- Extend the earlier example:
  \[
  \text{pow}(X, 0, s(0))
  \]
  \[
  \text{pow}(X, s(Y), Z) \leftarrow \text{pow}(X, Y, Z'), \text{mul}(X, Z', Z)
  \]

- Goal: \(\leftarrow \text{pow}(X, s(s(s(N))), XP), \text{pow}(Y, s(s(s(N))), YP), \text{pow}(Z, s(s(s(N))), ZP), \text{add}(XP, YP, ZP)\)

- In words:
  "Which natural numbers satisfy the equation \(x^n + y^n = z^n\), for \(n > 2\)?"

- Uh oh . . .
How hard is this computation?

- Extend the earlier example:
  \[
  \text{pow}(X, 0, s(0))
  \]
  \[
  \text{pow}(X, s(Y), Z) \leftarrow \text{pow}(X, Y, Z'), \text{mul}(X, Z', Z)
  \]
- Goal: \(\leftarrow \text{pow}(X, s(s(s(N))))), \text{XP}), \text{pow}(Y, s(s(s(N)))), \text{YP}), \text{pow}(Z, s(s(s(N))))), \text{ZP}), \text{add}(\text{XP}, \text{YP}, \text{ZP})\)
- In words:
  “Which natural numbers satisfy the equation \(x^n + y^n = z^n\), for \(n > 2\)?”
- Uh oh . . .

Definite logic programming is indeed **undecidable**.
How hard is this computation?

- Extend the earlier example:
  
  \[ \text{pow}(X, 0, s(0)) \]
  \[ \text{pow}(X, s(Y), Z) \leftarrow \text{pow}(X, Y, Z'), \text{mul}(X, Z', Z) \]

- Goal: \( \leftarrow \text{pow}(X, s(s(s(N))), XP), \text{pow}(Y, s(s(s(N))), YP), \text{pow}(Z, s(s(s(N))), ZP), \text{add}(XP, YP, ZP) \)

- In words:
  
  "Which natural numbers satisfy the equation \( x^n + y^n = z^n \), for \( n > 2 \)"

- Uh oh . . .

Definite logic programming is indeed **undecidable**.
But it is **semi-decidable**: all positive conclusions eventually appear
Decidable cases

Non-recursive definite programs

- Termination guaranteed for logic programs without recursive rules.
- Computational complexity: \( \text{NExpTime} \) (just like OWL DL)
Non-recursive definite programs

- Termination guaranteed for logic programs without recursive rules.
- Computational complexity: $\text{NExpTime}$ (just like OWL DL)

Datalog

- A datalog program is a definite program without function symbols.
- Datalog is decidable
- Computational complexity: $\text{ExpTime}$ (just like OWL Lite)
  - even with just a single rule and no facts (“sirup”)
- Data complexity: $\text{P}$ useful for deductive databases
Logical programs with negation

- **Normal logic programs**: first-order programs with negation in the body
- Typically non-monotonic negation
- Example:

  
  ```
  bird(tweety)
  bird(tux)
  penguin(tux)
  fly(X) :- bird(X), not penguin(X)
  ```
Negation as failure

- How to compute with non-monotonic negation?
How to compute with non-monotonic negation?

Negation as failure:

- Negated statements are true if their positive version cannot be derived.
- Start a new proof tree when encountering negated statements.
- SLDNF-resolution (SLD + Negation as Failure)
Negation as failure:

- How to compute with non-monotonic negation?

Negation as failure:

negated statements are true if it their positive version cannot be derived

\[ \neg \text{ start a new proof tree when encountering negated statements} \]

\[ \neg \text{ SLDNF-resolution (SLD + Negation as Failure)} \]

- *operational*, not *declarative*

- implemented in Prolog (usually based on incomplete SLD-implementation, making the system unsound when using negation)
What does the following mean?

\[ p :\neg \text{ not } q \]
\[ q :\neg \text{ not } p \]
Semantic issues

- What does the following mean?
  \[ p \leftarrow \neg q \]
  \[ q \leftarrow \neg p \]

- Classical semantics: \( p \lor q \)

- Negation as failure: loop proof search

- Intuition(?): neither \( p \) nor \( q \) “really” hold
What does the following mean?
\[ p :\not q \]
\[ q :\not p \]

Classical semantics: \( p \lor q \)

Negation as failure: loop proof search

Intuition(?): neither \( p \) nor \( q \) “really” hold

Introduction of three-valued semantics: true, false, unknown

Example: Well-founded semantics

Many different non-monotonic semantics, usually agree on more well-behaved programs
Expressive power

How powerful are non-monotonic semantics?

- Recall: in definite programs, we could ask whether there is a counterexample to Fermat’s Last Theorem
How powerful are non-monotonic semantics?

- Recall: in definite programs, we could ask whether there is a counterexample to Fermat’s Last Theorem
- In normal logic programs, we can also ask whether there is no such counterexample

$\leadsto$ Non-monotonic semantics usually strictly more powerful than monotonic variants
Expressive power

How powerful are non-monotonic semantics?

- Recall: in definite programs, we could ask whether there is a counterexample to Fermat’s Last Theorem
- In normal logic programs, we can also ask whether there is no such counterexample
  \[ \sim \] Non-monotonic semantics usually strictly more powerful than monotonic variants
- Well-founded semantics: not semidecidable (captures $\Pi_1^1$-fragment of second order logic)
Outline

1. Basics
2. Propositional logic programs
3. Definite logic programs
4. Normal programs
5. F-Logic
6. Summary
“Object oriented” logic programming

- different definitions of semantics: classical (first-order) and LP (non-monotonic)
- no normative definitions for syntax or semantics
- Implementation in ONtoobroker and Flora: assumes well-founded semantics (and computes unspecified parts of it)
- systems cannot return all answers or all true answers, even in infinite time
F-Logic example syntax

/* Atoms */
iokaste:woman.
laios:man.

/* Molecules */
oedipus:man[father->laios; mother->iokaste].
polyneikes:man[father->oedipus; mother->iokaste;
    sister->>{antigone, ismene};
    brother->>eteokles:man]

/* Rules */

/* Queries */
FORALL X,Y   <-  X:man[sohn->>Y[mother->iokaste]].
F-Logic semantics

- “Object oriented” notation is translated into facts:
  - $a : C \mapsto \text{isa}_a(a,C)$
  - $A :: B \mapsto \text{sub}_a(A,B)$
  - $a[B \rightarrow c] \mapsto \text{att}_a(a,B,c)$
  - $a[B \leftarrow c] \mapsto \text{setatt}_a(a,B,c)$
  - $A[B = C] \mapsto \text{atttype}_a(A,B,C)$
  - $A[B = C] \mapsto \text{setatttype}_a(A,B,C)$

- Properties of auxiliary predicates such as sub_ described by axioms.
- Further syntactic features treated with Lloyd-Topor translations.
F-Logic semantics

“Object oriented” notation is translated into facts:

- $a: C \mapsto \text{isa}(a, C)$
- $A::B \mapsto \text{sub}(A, B)$
- $a[B->c] \mapsto \text{att}(a, B, c)$
- $a[B=>c] \mapsto \text{setatt}(a, B, c)$
- $A[B=>C] \mapsto \text{atttype}(A, B, C)$
- $A[B=>C] \mapsto \text{setatttype}(A, B, C)$

Properties of auxiliary predicates such as $\text{sub}_-$ described by axioms.

Further syntactic features treated with Lloyd-Topor translations

Warning! The above is based on incomplete descriptions in Angele and Lausen (2004), which merely sketch the semantics. I know of no normative public spec of the current implementation in Ontobroker or Flora (which does not say that such documents do not exist).
Outline

1 Basics

2 Propositional logic programs

3 Definite logic programs

4 Normal programs

5 F-Logic

6 Summary
Complexity overview:

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Complexity</th>
<th>Var.?</th>
<th>Func.?</th>
<th>Neg.?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional programs</td>
<td>$P$</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Datalog</td>
<td>$\text{ExpTime-complete}$</td>
<td>$\times$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Definite programs</td>
<td>$\text{r.e.-complete}$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>—</td>
</tr>
<tr>
<td>— without recursion</td>
<td>$\text{NExpTime-complete}$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>—</td>
</tr>
<tr>
<td>Well-founded semantics</td>
<td>$\Pi_1^1$-complete</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

Prolog is based on definite programs

F-Logic is based on well-founded semantics + F-Logic
References