Computational Logic

Fundamentals of Definite Programs:
Syntax and Semantics
Towards Logic Programming

- Conclusion: resolution is a complete and effective deduction mechanism using:
  - Horn clauses (related to “Definite programs”),
  - Linear, Input strategy
  - Breadth-first exploration of the tree (or an equivalent approach)
    (possibly ordered clauses, but not required – see Selection rule later)

- Very close to what is generally referred to as SLD-resolution (see later)

- This allows to some extent realizing Green’s dream (within the theoretical limits of
  the formal method), and efficiently!
Given these results, why not use logic as a general purpose *programming language*? [Kowalski 74]

A “logic program” would have two interpretations:

- **Declarative** (“LOGIC”): the logical reading (facts, statements, knowledge)
- **Procedural** (“CONTROL”): what resolution does with the program

**ALGORITHM = LOGIC + CONTROL**

Specify these components separately

Often, worrying about control is not needed at all (thanks to resolution)

Control can be effectively provided through the ordering of the literals in the clauses
Towards Logic Programming: Another (more compact) Clausal Form

- All formulas are transformed into a set of *Clauses*.
  - A clause has the form:
    \[
    \text{conc}_1, \ldots, \text{conc}_m \leftarrow \text{cond}_1, \ldots, \text{cond}_n
    \]
    where
    \[
    \begin{align*}
    \{ \text{conc}_1, \ldots, \text{conc}_m \} & \text{ "or" } \\
    \{ \text{cond}_1, \ldots, \text{cond}_n \} & \text{ "and" }
    \end{align*}
    \]
    are literals, and are the *conclusions* and *conditions* of a rule:
    
    \[
    \begin{array}{ll}
    \{ \text{conc}_1, \ldots, \text{conc}_m \} & \text{ "conclusions" } \\
    \{ \text{cond}_1, \ldots, \text{cond}_n \} & \text{ "conditions" }
    \end{array}
    \]
  - All variables are implicitly universally quantified: (if \( X_1, \ldots, X_k \) are the variables)
    \[
    \forall X_1, \ldots, X_k \quad \text{conc}_1 \lor \cdots \lor \text{conc}_m \leftarrow \text{cond}_1 \land \cdots \land \text{cond}_n
    \]

- More compact than the traditional clausal form:
  - no connectives, just commas
  - no need to repeat negations: all negated atoms on one side, non-negated ones on the other

- A *Horn Clause* then has the form:
  where \( n \) can be zero and possibly \( \text{conc}_1 \) empty.
  \[
  \text{conc}_1 \leftarrow \text{cond}_1, \ldots, \text{cond}_n
  \]
Some Logic Programming Terminology – “Syntax” of Logic Programs

• **Definite Program**: a set of positive Horn clauses \( \text{head} \leftarrow \text{goal}_1, \ldots, \text{goal}_n \)

• The single *conclusion* is called the *head*.

• The conditions are called “goals” or “procedure calls”.

• \( \text{goal}_1, \ldots, \text{goal}_n \) \((n \geq 0)\) is called the “body”.

• if \( n = 0 \) the clause is called a “fact” (and the arrow is normally deleted)

• Otherwise it is called a “rule”

• **Query** (question): a negative Horn clause (a “headless” clause)

• A procedure is a set of rules and facts in which the heads have the same predicate symbol and arity.

• Terms in a goal are also called “arguments”.
• Examples:
  grandfather(X,Y) ← father (X,Z), mother(Z,Y).
  grandfather(X,Y) ←.
  grandfather(X,Y).
  ← grandfather(X,Y).
LOGIC: Declarative “Reading” (Informal Semantics)

- A rule (has head and body)

  \[ \text{head} \leftarrow \text{goal}_1, \ldots, \text{goal}_n. \]

  which contains variables \( X_1, \ldots, X_k \) can be read as
  for all \( X_1, \ldots, X_k \):
  “head” is true if “goal_1” and ... and “goal_n” are true

- A fact n=0 (has only head)

  \[ \text{head}. \]

  for all \( X_1, \ldots, X_k \): “head” is true (always)

- A query (the headless clause)

  \[ \leftarrow \text{goal}_1, \ldots, \text{goal}_n \]

  can be read as:
  for which \( X_1, \ldots, X_k \) are “goal_1” and ... and “goal_n” true?
• Given a first-order language $L$, with a non-empty set of variables, constants, function symbols, relation symbols, connectives, quantifiers, etc. and given a syntactic object $A$,

$$\text{ground}(A) = \{ A\theta | \exists \theta \in \text{Subst}, \text{var}(A\theta) = \emptyset \}$$

i.e. the set of all “ground instances” of $A$.

• Given $L$, $U_L$ (Herbrand universe) is the set of all ground terms of $L$.

• $B_L$ (Herbrand Base) is the set of all ground atoms of $L$.

• Similarly, for the language $L_P$ associated with a given program $P$ we define $U_P$, and $B_P$.

• Example:

$P = \{ \ p(f(X)) \leftarrow p(X). \ p(a). \ q(a). \ q(b). \ \}$

$U_P = \{ a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots \}$

$B_P = \{ p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), \ldots \}$
Herbrand Interpretations and Models

• A Herbrand Interpretation is a subset of $B_L$, i.e. the set of all Herbrand interpretations $I_L = \wp(B_L)$.

(Note that $I_L$ forms a complete lattice under $\subseteq$ – important for fixpoint operations to be introduced later).

• Example: $P = \{ p(f(X)) \leftarrow p(X). \; p(a). \; q(a). \; q(b). \} \\
U_P = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\} \\
B_P = \{p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), \ldots\} \\
I_P = \text{all subsets of } B_P$

• A Herbrand Model is a Herbrand interpretation which contains all logical consequences of the program.

• The Minimal Herbrand Model $H_P$ is the smallest Herbrand interpretation which contains all logical consequences of the program. (It is unique.)

• Example: $H_P = \{q(a), q(b), p(a), p(f(a)), p(f(f(a))), \ldots\}$
• *Declarative semantics of a logic program* $P$: the set of ground facts which are logical consequences of the program (i.e., $H_P$). (Also called the “least model” semantics of $P$).

• *Intended meaning of a logic program* $P$: the set $M$ of ground facts that the user expects to be logical consequences of the program.

• A logic program is *correct* if $H_P \subseteq M$.

• A logic program is *complete* if $M \subseteq H_P$.

• Example:
  
  father(john,peter).
  father(john,mary).
  mother(mary,mike).
  
  grandfather(X,Y) ← father(X,Z), father(Z,Y).
  
  with the usual intended meaning is *correct* but *incomplete*. 
We now turn to the operational semantics of logic programs, given by a concrete operational procedure: *Linear (Input) Resolution*.

- Complementary literals:
  - in two different clauses
  - on different sides of $\leftarrow$
  - unifiable with unifier $\theta$

father(john, mary) $\leftarrow$
grandfather(X, Y) $\leftarrow$ father(X, Z), mother(Z, Y)

$\theta = \{X/john, Z/mary\}$
Resolution step (linear, input, ...):

- given a clause and a resolvent, we can build a new resolvent which follows from them by:
  - renaming apart the clause ("standardization apart" step)
  - putting all the conclusions to the left of the ←
  - putting all the conditions to the right of the ←
  - if there are complementary literals (unifying literals at different sides of the arrow in the two clauses), eliminating them and applying $\theta$ to the new resolvent

LD-Resolution: linear (and input) resolution, applied to definite programs
Note that then all resolvents are negative Horn clauses (like the query).
Example

- from
  \[\text{father}(\text{john}, \text{peter}) \leftarrow\]
  \[\text{mother}(\text{mary}, \text{david}) \leftarrow\]
  we can infer
  \[\text{father}(\text{john}, \text{peter}), \text{mother}(\text{mary}, \text{david}) \leftarrow\]

- from
  \[\text{father}(\text{john}, \text{mary}) \leftarrow\]
  \[\text{grandfather}(X, Y) \leftarrow \text{father}(X, Z), \text{mother}(Z, Y)\]
  we can infer
  \[\text{grandfather}(\text{john}, Y') \leftarrow \text{mother}(\text{mary}, Y')\]
CONTROL: A proof using LD-Resolution

- Prove “grandfather(john,david) ↛” using the set of axioms:
  1. father(john,peter) ↛
  2. father(john,mary) ↛
  3. father(peter,mike) ↛
  4. mother(mary,david) ↛
  5. grandfather(L,M) ↛ father (L,N), father(N,M)
  6. grandfather(X,Y) ↛ father (X,Z), mother(Z,Y)

- We introduce the predicate to prove (negated!)
  7. ↛ grandfather(john,david)

- We start resolution: e.g. 6 and 7
  8. ↛ father(john,Z₁), mother(Z₁,david) \( X^1/john, Y^1/david \)

- using 2 and 8
  9. ↛ mother(mary,david) \( Z^1/mary \)

- using 4 and 9
  ↛
CONTROL: Rules and SLD-Resolution

- Two control-related issues are still left open in LD-resolution. Given a current resolvent $R$ and a set of clauses $K$:
  - given a clause $C$ in $K$, several of the literals in $R$ may unify the non-negated a complementary literal in $C$
  - given a literal $L$ in $R$, it may unify with complementary literals in several clauses in $K$

- A *Computation* (or *Selection* rule) is a function which, given a resolvent (and possibly the proof tree up to that point) returns (selects) a literal from it. This is the goal that will be used next in the resolution process.

- A *Search* rule is a function which, given a literal and a set of clauses (and possibly the proof tree up to that point), returns a clause from the set. This is the clause that will be used next in the resolution process.
• SLD-resolution: Linear resolution for Definite programs with Selection rule.

• An SLD-resolution method is given by the combination of a computation (or selection) rule and a search rule.

• Independence of the computation rule: Completeness does not depend on the choice of the computation rule.

• Example: a “left-to-right” rule (as in ordered resolution) does not impair completeness – this coincides with the completeness result for ordered resolution.

• Fundamental result:
  “Declarative” semantics ($H_P$) $\equiv$ “operational” semantics (SLD-resolution)
  I.e., all the facts in $H_P$ can be deduced using SLD-resolution.
Given a rule

\[ \text{head} \leftarrow \text{goal}_1, \ldots, \text{goal}_n. \]

it can be seen as a description of the goals the solver (resolution method) has to execute in order to solve “head”

- Possible, given computation and search rules.
- In general, “In order to solve ‘head’, solve ‘goal_1’ and ... and solve ‘goal_n’”
- If ordered resolution is used (left-to-right computation rule), then read “In order to solve ‘head’, first solve ‘goal_1’ and then ‘goal_2’ and then ... and finally solve ‘goal_n’”

Thus the “control” part corresponding to the computation rule is often associated with the order of the goals in the body of a clause

Another part (corresponding to the search rule) is often associated with the order of clauses
• Example – read “procedurally”:
  
father(john,peter).
father(john,mary).
father(peter,mike).
father(X,Y) ← mother(Z,Y), married(X,Z).
Towards a Fixpoint Semantics for LP – Fixpoint Basics

- A fixpoint for an operator $T : X \rightarrow X$ is an element of $x \in X$ such that $x = T(x)$.
- If $X$ is a poset, $T$ is monotonic if $\forall x, y \in X, x \leq y \Rightarrow T(x) \leq T(y)$.
- If $X$ is a complete lattice and $T$ is monotonic the set of fixpoints of $T$ is also a complete lattice [Tarski].
- The least element of the lattice is the least fixpoint of $T$, denoted $lfp(T)$.
- Powers of a monotonic operator (successive applications):
  - $T \uparrow 0(x) = x$
  - $T \uparrow n(x) = T(T \uparrow (n-1)(x))(n$ is a successor ordinal)
  - $T \uparrow \omega(x) = \bigsqcup\{T \uparrow n(x)\mid n < \omega\}$

  We abbreviate $T \uparrow \alpha(\bot)$ as $T \uparrow \alpha$.

- There is some $\omega$ such that $T \uparrow \omega = lfp T$. The sequence $T \uparrow 0, T \uparrow 1, ..., lfp T$ is the Kleene sequence for $T$.
- In a finite lattice the Kleene sequence for a monotonic operator $T$ is finite.
A subset $Y$ of a poset $X$ is an (ascending) chain iff $\forall y, y' \in Y, y \leq y' \lor y' \leq y$

A complete lattice $X$ is *ascending chain finite* (or *Noetherian*) if all ascending chains are finite

In an ascending chain finite lattice the Kleene sequence for a monotonic operator $T$ is finite
Lattice Structures

**finite**

```
finite
```

```
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (b) at (1,0) {b};
  \node (c) at (2,0) {c};
  \node (d) at (1,1) {d};
  \node (e) at (1,2) {e};
  \draw (a) -- (b);
  \draw (a) -- (c);
  \draw (b) -- (d);
  \draw (b) -- (e);
  \draw (c) -- (d);
  \draw (c) -- (e);
\end{tikzpicture}
```

**finite_depth**

```
finite_depth
```

```
\begin{tikzpicture}
  \node (a) at (0,0) {\ldots inf \ldots};
  \node (b) at (1,0) {\ldots inf \ldots};
  \node (c) at (2,0) {\ldots inf \ldots};
  \node (d) at (1,1) {\ldots inf \ldots};
  \node (e) at (1,2) {\ldots inf \ldots};
  \draw (a) -- (b);
  \draw (a) -- (c);
  \draw (b) -- (d);
  \draw (b) -- (e);
  \draw (c) -- (d);
  \draw (c) -- (e);
\end{tikzpicture}
```

**ascending chain finite**

```
ascending chain finite
```

```
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (1,0) {2};
  \node (3) at (2,0) {3};
  \node (4) at (3,0) {4};
  \node (top) at (0,1) {\top};
  \draw (top) -- (1);
  \draw (top) -- (2);
  \draw (top) -- (3);
  \draw (top) -- (4);
\end{tikzpicture}
```
The *Immediate consequence operator* $T_P$ is a mapping: $T_P : I_P \rightarrow I_P$ defined by:

$$T_P(I) = \{ A \in B_P | \exists C \in ground(P), C = A \leftarrow L_1, \ldots, L_n \text{ and } L_1, \ldots L_n \in I \}$$

(in particular, if $(A \leftarrow) \in P$, then every element of $ground(A)$ is in $T_P(I)$, $\forall I$).

- $T_P$ is monotonic, so it has a least fixpoint $I^*$ so that $T_P(I^*) = I^*$, which can be obtained by applying $T_P$ iteratively starting from the bottom element of the lattice (the empty interpretation).

- (Characterization Theorem) [Van Emden and Kowalski] A program $P$ has a Herbrand model $H_P$ such that:
  - $H_P$ is the least Herbrand Model of $P$.
  - $H_P$ is the least fixpoint of $T_P$ ($lfp\, T_P$).
  - $H_P = T_P \uparrow \omega$.

I.e., *least model semantics* $(H_P) \equiv *fixpoint semantics* (lfp\, T_P)$

- Because it gives us some intuition on how to build $H_P$, the least fixpoint semantics can in some cases (e.g., finite models) also be an operational semantics (e.g., in *deductive databases*).
A Fixpoint Semantics for Logic Programs: Example

• Example:

\[
P = \{ \ p(f(X)) \leftarrow p(X). \\
p(a). \\
q(a). \\
q(b). \ \}
\]

\[
U_P = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\}
\]

\[
B_P = \{p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), \ldots\}
\]

\[
I_P = \text{all subsets of } B
\]

\[
H_P = \{q(a), q(b), p(a), p(f(a)), p(f(f(a))), \ldots\}
\]

\[
T_P \uparrow 0 = \{p(a), q(a), q(b)\}
\]

\[
T_P \uparrow 1 = \{p(a), q(a), q(b), p(f(a))\}
\]

\[
T_P \uparrow 2 = \{p(a), q(a), q(b), p(f(a)), p(f(f(a)))\}
\]

\[
\ldots
\]

\[
T_P \uparrow \omega = H_P
\]