

Logic Programming

Syntax and semantics

Logic Programming

A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have desired meaning.

Sterling and Shapiro: The Art of Prolog, Page 1.

LP Syntax

- *goal (or query)*:
 - ▶ *empty goal* \top (top) or \perp (bottom), or
 - ▶ atom, or
 - ▶ conjunction of goals
- *(Horn) clause*: $A \leftarrow G$
 - ▶ *head* A : atom
 - ▶ *body* G : goal
- Naming conventions
 - ▶ *fact*: clause of form $A \leftarrow \top$
 - ▶ *rule*: all others
- *(logic) program*: finite set of Horn clauses
- predicate symbol *defined*: it occurs in head of a clause

Note that a Horn clause is a clause with at most one positive literal

LP Calculus – Syntax

<i>Atom:</i>	A, B	$::=$	$p(t_1, \dots, t_n), n \geq 0$
<i>Goal:</i>	G, H	$::=$	$\top \mid \perp \mid A \mid G \wedge H$
<i>Clause:</i>	K	$::=$	$A \leftarrow G$
<i>Program:</i>	P	$::=$	$K_1 \dots K_m, m \geq 0$

LP Calculus – State Transition System

We now define the semantics of LP in terms of a transition system.

- A *state* is a pair $\langle G, \theta \rangle$ where
 - ▶ G is a goal
 - ▶ θ is a substitution
- An *initial state* has the form $\langle G, \epsilon \rangle$
- A *successful final state* has the form $\langle \top, \theta \rangle$
- A *failed final state* has the form $\langle \perp, \epsilon \rangle$

Derivations, Goals

A derivation is

- *successful* if its final state is successful
- *failed* if its final state is failed
- *infinite* if there are an infinite sequence of states and transitions $S_1 \mapsto S_2 \mapsto S_3 \mapsto \dots$

A goal G is

- *successful* if it has a successful derivation starting with $\langle G, \epsilon \rangle$
- *finitely failed* if has only failed derivations starting with $\langle G, \epsilon \rangle$

Computer Answer Substitution

Given an initial goal G (or *query*) and a program P , if there exists a successful derivation (in P)

$$\langle G, \epsilon \rangle \mapsto^* \langle \top, \theta \rangle$$

then the substitution θ is called *Computed Answer Substitution* (c.a.s.) of G (in P)

- The c.a.s. is the result of the computation: the values associated by the c.a.s. to the variables in the goal provide the results.

LP Transition Rules

Unfold

If $(B \leftarrow H)$ is a fresh variant of a clause in P
and β is the most general unifier of B and $A\theta$
then $\langle A \wedge G, \theta \rangle \mapsto \langle H \wedge G, \theta\beta \rangle$

Failure

If there is no clause $(B \leftarrow H)$ in P
with a unifier of B and $A\theta$
then $\langle A \wedge G, \theta \rangle \mapsto \langle \perp, \epsilon \rangle$

Non-determinism

The **Unfold** transition exhibits two kinds of non-determinism.

- *don't-care non-determinism*:
 - ▶ any atom in $A \wedge G$ can be chosen as the atom A according to the congruence defined on states
 - ▶ affects length of derivation (infinitely in the worst case)
- *don't-know non-determinism*:
 - ▶ any clause $(B \leftarrow H)$ in P for which B and $A\theta$ are unifiable can be chosen
 - ▶ determines the computed answer of derivation

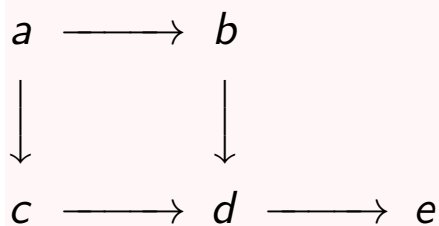
SLD Resolution

- Selection rule Driven Linear resolution for definite clauses.
- The idea is the use a *Selection function* to select the atom in the goal for the application of the unfolding rule (which corresponds to the selection of a literal for applying the resolution rule).
- This eliminates the don't care non determinism (the don't know remains)
- By using all the possible clauses with a specific selection rule we obtain a search tree called SLD tree
- A search strategy must be defined to eliminate also the don't know non determinism, i.e. to visit the SLD tree in order to find a successful derivation.

SLD Resolution: Prolog implementation

- Prolog uses a selection rule which selects the *leftmost* atom.
- Prolog uses for search strategy the textual order of clauses with *(chronological) backtracking*.
- This provides a left-to-right, depth-first exploration of the SLD tree. One can obtain efficient implementation by using a stack-based approach, but can get trapped in infinite derivations. (However breadth-first search far too inefficient).

Example - Accessibility in DAG



$\text{edge}(a,b) \leftarrow \top \text{ (e1)}$

$\text{edge}(a,c) \leftarrow \top \text{ (e2)}$

$\text{edge}(b,d) \leftarrow \top \text{ (e3)}$

$\text{edge}(c,d) \leftarrow \top \text{ (e4)}$

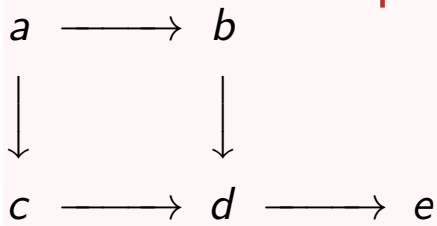
$\text{edge}(d,e) \leftarrow \top \text{ (e5)}$

$\text{path}(\text{Start},\text{End}) \leftarrow \text{edge}(\text{Start},\text{End}) \text{ (p1)}$

$\text{path}(\text{Start},\text{End}) \leftarrow \text{edge}(\text{Start},\text{Node}) \wedge \text{path}(\text{Node},\text{End}) \text{ (p2)}$

Note: e1 en and p1, p2 are names of rules in the metalanguage, they are part of the LP syntax.

Example - Accessibility in DAG (cont)

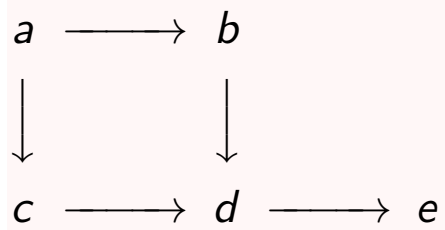


$$\begin{aligned}
 & \langle \text{path}(b, Y), \varepsilon \rangle \\
 \mapsto (p1) & \langle \text{edge}(S, E), \{S \mapsto b, E \mapsto Y\} \rangle \\
 \mapsto (e3) & \langle \top, \{S \mapsto b, E \mapsto d, Y \mapsto d\} \rangle
 \end{aligned}$$

With the second rule $p2$ for path selected:

$$\begin{aligned}
 & \langle \text{path}(b, Y), \varepsilon \rangle \\
 \mapsto (p2) & \langle \text{edge}(S, N) \wedge \text{path}(N, E), \{S \mapsto b, E \mapsto Y\} \rangle \\
 \mapsto (e3) & \langle \text{path}(N, E), \{S \mapsto b, E \mapsto Y, N \mapsto d\} \rangle \\
 \mapsto (p1) & \langle \text{edge}(N, E), \{S \mapsto b, E \mapsto Y, N \mapsto d\} \rangle \\
 \mapsto (e5) & \langle \top, \{S \mapsto b, E \mapsto e, N \mapsto d, Y \mapsto e\} \rangle
 \end{aligned}$$

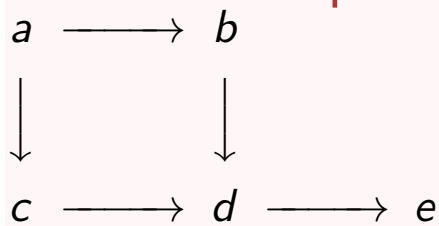
Example - Accessibility in DAG (cont 2)



Partial search tree:

As exercise

Example - Accessibility in DAG (cont 2)



With the first rule:

$$\begin{array}{ll}
 & \langle \text{path}(f, g), \varepsilon \rangle \\
 \mapsto (p1) & \langle \text{edge}(S, E), \{S \mapsto f, E \mapsto g\} \rangle \\
 \mapsto & \langle \perp, \varepsilon \rangle
 \end{array}$$

With the second rule (and special selection) we get an infinite derivation:

$$\begin{array}{ll}
 & \langle \text{path}(f, g), \varepsilon \rangle \\
 \mapsto (p2) & \langle \text{path}(N, E) \wedge \text{edge}(S, N), \{S \mapsto f, E \mapsto g\} \rangle \\
 \mapsto (p2) & \langle \text{edge}(S, N) \wedge \text{edge}(N, N1) \wedge \text{path}(N1, E), \{S \mapsto f, E \mapsto g\} \rangle
 \end{array}$$

Declarative Semantics

- Remember that the *implication* ($G \rightarrow A$) is a (Horn) clause, where all variables are implicitly considered universally quantified.
- The *logical reading of a program* P is obtained by considering the conjunction of the clauses of P .
- Note that this logical reading is incomplete, since only positive information can be derived.

Example. In DAG with nodes a, b, c, d, e : $P \not\models \text{path}(f, g)$ however we cannot derive from the program P that $\neg \text{path}(f, g)$ holds, indeed also $P \not\models \neg \text{path}(f, g)$

- In order to treat also negative information we should take the *completion* of a program, i.e. to transform the implications in double implications (i. e. add sufficient conditions). We will not consider it.

Soundness and Completeness of SLD resolution

Given P logic program, G goal and θ substitution:

- **Soundness:**

If θ is a computed answer of G , then $P \models G\theta$.

- **Completeness:**

If $P \models G\theta$, then a computed answer substitution σ of G exists, such that $G\theta = G\sigma\beta$.

Moreover if a c.a.s. σ of G can be obtained by using a selection rule r it can be obtained also by using a different selection rule r' .