Transparencies for the course TDDA41 Logic Programming, given at the Department of Computer and Information Science, Linköping University, Sweden.

Printout: November 25, 2006

Introduction: Overview

- Goals of the course.

- What is logic programming?

- Why logic programming?

Goals of the course

- Logic as a specification **AND** programming language;
- Theoretical foundation of logic programming;
- Practice of Prolog and constraint programming;
- Relations to other areas:
  - Databases
  - Formal/natural languages
  - Combinatorial problems
- To program **DECLARATIVELY**.

Declarative vs imperative languages

	Imperative	Declarative
Paradigm	Describe HOW TO solve the problem	Describe WHAT the problem is
Program	A sequence of commands	A set of statements
Examples	C, Fortran, Ada, Java	Prolog, Pure Lisp, Haskell, ML
Advantages	Fast, specialized programs	General, readable, correct(?) programs.

Declarative description A grandchild to x is a child of one of x's children.

Imperative description I To find a grandchild of x, first find a child of x. Then find a child of that child.

Imperative description II To find a grandchild of x, first find a parent-child pair and then check if the parent is a child of x.

Imperative description III To find a grandchild of x, compute the factorial of 123, then find a child of x. Then find a child of that child.

Compare . . .

```
read(person);
for i := 1 to maxparent do
  if parent[i;1] = person then
    for j := 1 to maxparent do
      if parent[j;1] = parent[i;2] then
        write(parent[j;2]);
      fi
    od
  fi
od
```

with . . .

```
gc(X,Z) :- c(X,Y), c(Y,Z).
```

Logic: Overview

- Syntax and semantics
- Vocabulary, terms and formulas
- Interpretations and models
- Logical consequence and equivalence
- Proofs/derivations
- Soundness and completeness

Predicate logic vocabulary

- Constants $(17, george, tEX, \ldots)$

- Functors $(cons/2, +/2, father/1, \ldots)$

- Predicate symbols
  $(member/2, </2, father/1, \ldots)$

- Variables $(X, X11, \_, \_123, TeX, \ldots)$

- Logical connectives $(\wedge, \vee, \supset, \neg, \leftrightarrow)$

- Quantifiers $(\forall, \exists)$

- Auxiliary symbols $(., (,), \ldots)$

Example

$$A = \{\texttt{volvo}; \texttt{owner}/1; \texttt{owns}/2, \texttt{happy}/1\}$$

Terms

Let A be a vocabulary.

The set of all *terms* over A is the least set such that

- every constant in A is a term;
- every variable is a term;
- if f/n is a functor in A and $t_1, \ldots, t_n$ are terms over A then $f(t_1, \ldots, t_n)$ is a term.

Ground terms

A term that contains no variables is called a *ground* term.

(Well-formed) formulas

Let A be a vocabulary.

The set of all *formulas* over A is the least set such that:

- if p/n is a predicate symbol in A and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula;

- if F and G are formulas, then $(F \wedge G), (F \vee G), (F \supset G), (F \leftrightarrow G)$ and $\neg F$ are formulas;

- if F is a formula and X a variable, then $\forall X \ F$ and $\exists X \ F$ are formulas.

Atoms

A formula of the form $p(t_1, \ldots, t_n)$ is called an *atomic formula* (atom).

Free occurrences of variables

An occurrence of X in a formula is said to be *free* iff the occurrence does not follow immediately after a quantifier, or in a formula immediately after $\forall X$ or $\exists X$.

Closed formulas

A formula that does not contain any free occurrences of variables is said to be *closed*.

Universal closure

Assume that $\{X_1, \ldots, X_n\}$ are the only free occurrences of variables in a formula F. The *universal closure* $\forall F$ of F is the closed formula $\forall X_1 \ \ldots \forall X_n \ F$.

The existential closure $\exists F$ is defined similarly.

Interpretations

Let A be a vocabulary.

An *interpretation* $\Im$ of A consists of (1) a non-empty set D (often written $|\Im|$) of objects (the domain of $\Im$) and (2) a function that maps:

- every constant c in A on an element $c_\Im$ in D;
- every functor f/n in A on a function $f_\Im : D^n \to D$;
- every predicate symbol p/n in A on a relation $p_\Im \subseteq D^n$.

Example

The vocabulary:

$$A = \{\texttt{volvo}; \texttt{owner}/1; \texttt{owns}/2, \texttt{happy}/1\}$$

Consider $\Im$ where $|\Im| = \{0, 1, 2, \ldots\}$ and were:

- $\texttt{volvo}_\Im = 0$
- $\texttt{owner}_\Im(x) = x + 1$
- $\texttt{owns}_\Im =$ greater-than
- $\texttt{happy}_\Im =$ nonzero-property

NOTE!

An interpretation defines how to interpret constants, functors and predicate symbols but it does not say what a variable denotes.

Valuation

A *valuation* is a function from variables to objects in the domain of an interpretation.

The interpretation of terms

Let $\Im$ be an interpretation of a vocabulary A. Let σ be a valuation.

The interpretation $\sigma_{\Im}(t)$ of the term t is an object in $\Im$'s domain:

- if t is a constant c then $\sigma_{\Im}(t) = c_{\Im}$;
- if t is a variable X then $\sigma_{\Im}(t) = \sigma(X)$;
- if t is a term $f(t_1, \ldots, t_n)$ then $\sigma_{\Im}(t) = f_{\Im}(\sigma_{\Im}(t_1), \ldots, \sigma_{\Im}(t_n))$.

Example

Consider $\Im$ where $|\Im| = \{0, 1, 2, \ldots\}$ and were:

- $\mathtt{volvo}_{\Im} = 0$

- $\mathtt{owner}_{\Im}(x) = x + 1$

Then:

$$\begin{aligned}
& \sigma_{\Im}(\mathtt{owner}(\mathtt{owner}(\mathtt{volvo}))) \\
= \;& \mathtt{owner}_{\Im}(\sigma_{\Im}(\mathtt{owner}(\mathtt{volvo}))) \\
= \;& (\sigma_{\Im}(\mathtt{owner}(\mathtt{volvo}))) + 1 \\
= \;& (\mathtt{owner}_{\Im}(\sigma_{\Im}(\mathtt{volvo}))) + 1 \\
= \;& ((\sigma_{\Im}(\mathtt{volvo})) + 1) + 1 \\
= \;& ((\mathtt{volvo}_{\Im}) + 1) + 1 \\
= \;& (0 + 1) + 1 \\
= \;& 2
\end{aligned}$$

Example

Consider also $\sigma(\mathtt{X}) = 3$. Then:

$$
\begin{aligned}
& \sigma_{\mathfrak{I}}(\mathtt{owner}(\mathtt{X})) \\
= \; & \mathtt{owner}_{\mathfrak{I}}(\sigma_{\mathfrak{I}}(\mathtt{X})) \\
= \; & (\sigma_{\mathfrak{I}}(\mathtt{X})) + 1 \\
= \; & (\sigma(\mathtt{X})) + 1 \\
= \; & 3 + 1 \\
= \; & 4
\end{aligned}
$$

The interpretation of formulas

The meaning of a formula is a truth-value—"true" or "false". Given an interpretation $\Im$ and a valuation σ we write

$\Im \models_\sigma F$ when F is true wrt $\Im$ and σ.
$\Im \not\models_\sigma F$ when F is false wrt $\Im$ and σ.

- $\Im \models_\sigma p(t_1, \ldots, t_n)$ iff $(\sigma_\Im(t_1), \ldots, \sigma_\Im(t_n)) \in p_\Im$;

- $\Im \models_\sigma \neg F$ iff $\Im \not\models_\sigma F$;

- $\Im \models_\sigma F \wedge G$ iff $\Im \models_\sigma F$ and $\Im \models_\sigma G$;

- $\Im \models_\sigma F \vee G$ iff $\Im \models_\sigma F$ and/or $\Im \models_\sigma G$;

The interpretation of formulas (cont'd.)

- $\Im \models_\sigma F \supset G$ iff $\Im \not\models_\sigma F$ and/or $\Im \models_\sigma G$;

- $\Im \models_\sigma F \leftrightarrow G$ iff $\Im \models_\sigma F$ exactly when $\Im \models_\sigma G$;

- $\Im \models_\sigma \forall X F$ iff $\Im \models_{\sigma[x \mapsto t]} F$ for every $t \in |\Im|$;

- $\Im \models_\sigma \exists X F$ iff $\Im \models_{\sigma[x \mapsto t]} F$ for some $t \in |\Im|$.

Example

Consider $\Im$ as before.

Then:

$$\Im \models \texttt{owns}(\texttt{volvo}, \texttt{volvo}) \supset \texttt{happy}(\texttt{volvo})$$

iff

$$\Im \not\models \texttt{owns}(\texttt{volvo}, \texttt{volvo})$$

or

$$\Im \models \texttt{happy}(\texttt{volvo})$$

iff

$$\langle \sigma_\Im(\texttt{volvo}), \sigma_\Im(\texttt{volvo}) \rangle \notin \texttt{owns}_\Im$$

or

$$\sigma_\Im(\texttt{volvo}) \in \texttt{happy}_\Im$$

iff

$$\langle 0, 0 \rangle \notin \texttt{owns}_\Im \text{ or } 0 \in \texttt{happy}_\Im$$

iff

$$0 \not> 0 \text{ or } 0 \neq 0$$

iff

$$\texttt{true}$$

Models

Let F be a closed formula.
Let P be a set of closed formulas.

An interpretation $\Im$ is a *model* of F iff $\Im \models F$.

An interpretation $\Im$ is a model of P iff $\Im$ is a model of every formula in P.

Satisfiability

F (resp. P) is *satisfiable* iff F (resp. P) have at least one model. (Otherwise F/P is unsatisfiable.)

Example

$\Im$ (defined as before) is a model of:

$$\mathtt{owns}(\mathtt{owner}(\mathtt{volvo}), \mathtt{volvo})$$

and:

$$\forall \mathtt{X}(\mathtt{owns}(\mathtt{X}, \mathtt{volvo}) \supset \mathtt{happy}(\mathtt{X}))$$

Logical consequence

F is a logical consequence of P ($P \models F$) iff F is true in all of P's models ($\mathsf{Mod}(P) \subseteq \mathsf{Mod}(F)$).

Theorem

$P \models F$ iff $P \cup \{\neg F\}$ is unsatisfiable.

Logical equivalence

Let $F, G, \forall X H(X)$ be formulas.

F and G are logically equivalent ($F \equiv G$) iff $\Im \models_\sigma F$ exactly when $\Im \models_\sigma G$.

$$
\begin{aligned}
F \supset G &\equiv \neg F \vee G \\
F \supset G &\equiv \neg G \supset \neg F \\
F \leftrightarrow G &\equiv (F \supset G) \wedge (G \supset F) \\
\neg(F \wedge G) &\equiv \neg F \vee \neg G \\
\neg(F \vee G) &\equiv \neg F \wedge \neg G \\
\neg \forall X H(X) &\equiv \exists X \neg H(X) \\
\neg \exists X H(X) &\equiv \forall X \neg H(X)
\end{aligned}
$$

In addition, if X does not occur free in F.

$$\forall X(F \vee H(X)) \equiv F \vee \forall X H(X)$$

Proofs (derivations)

A proof (derivation) is a sequence of formulas where each formula in the sequence is either a so-called *premise* or is obtained from previous formulas in the sequence by means of a collection of *derivation rules*.

Natural deductions

$$\frac{F \qquad F \supset G}{G} \qquad \frac{\forall X F(X)}{F(t)} \qquad \frac{F \qquad G}{F \wedge G}$$

Example

1.	$\mathtt{owns(owner(volvo), volvo)}$	P
2.	$\forall \mathtt{X}(\mathtt{owns(X, volvo)} \supset \mathtt{happy(X)})$	P
3.	$\mathtt{owns(owner(volvo), volvo)} \supset \mathtt{happy(owner(volvo))}$	
4.	$\mathtt{happy(owner(volvo))}$	

Proofs

Let P be a set of closed formulas (premises)
Let F be a closed formula.

We write $P \vdash F$ when *there is* a derivation of F from the premises P.

Soundness and completeness

If $P \vdash F$ then $P \models F$. (soundness)

If $P \models F$ then $P \vdash F$. (completeness)

Definite Programs: Overview

- Definite programs:
  - Rules;
  - Facts;
  - Goals.
- Herbrand-interpretations;
- Herbrand-models;
- Fixpoint-semantics.

Clauses

A clause is a formula:

$$\forall(A_1 \vee \ldots \vee A_m \vee \neg A_{m+1} \vee \ldots \vee \neg A_{m+n})$$

where $A_1, \ldots, A_m, A_{m+1}, \ldots, A_{m+n}$ are atoms and $m, n \geq 0$.

$$\forall(A_1 \vee \ldots \vee A_m \vee \neg A_{m+1} \vee \ldots \vee \neg A_{m+n})$$

$$\equiv$$

$$\forall((A_1 \vee \ldots \vee A_m) \vee \neg(A_{m+1} \wedge \ldots \wedge A_{m+n}))$$

$$\equiv$$

$$\forall((A_1 \vee \ldots \vee A_m) \leftarrow (A_{m+1} \wedge \ldots \wedge A_{m+n}))$$

Definite clauses

A definite clause is a clause where $m \leq 1$:

Rules

A rule is a clause where $m = 1$ and $n > 0$:

$$\forall(A_1 \leftarrow A_2 \wedge \ldots \wedge A_{m+n})$$

Facts

A fact is a clause where $m = 1$ and $n = 0$:

$$\forall(A_1)$$

(Definite) goals

A goal is a clause where $m = 0$ and $n \geq 0$:

$$\forall(\neg(A_1 \wedge \ldots \wedge A_{m+n}))$$

A goal where $m = n = 0$ is called the empty goal.

Notation

Rules:	$A_1 \leftarrow A_2, \ldots, A_{n+1}.$	$n > 0$
Facts:	$A_1.$	
Goals:	$\leftarrow A_1, \ldots, A_n.$	$n > 0$
	$\square$	$n = 0$

Logic Programming Anatomy

$$\begin{array}{ccc} \text{head} & \text{neck} & \text{body} \\ A_0 & \leftarrow & A_1, \ldots, A_n \end{array}$$

Logic programs

A definite program is a finite set of rules and facts.

A definite program P is used to answer "existential questions" (queries) such as:

"are there any odd integers?"

The query can be answered "yes" if e.g:

$$P \models \exists X \ \mathit{odd}(X)$$

This is equivalent to proving that:

$$P \cup \{\neg \exists X \ \mathit{odd}(X)\}$$

is unsatisfiable (has no models).

Resolution

Note that $\neg\exists(A_1 \wedge \ldots \wedge A_n)$ is equivalent to $\forall\neg(A_1 \wedge \ldots \wedge A_n)$. That is, a goal.

Resolution is used to prove that a set of clauses is unsatisfiable. As a side-effect resolution produces “witnesses” (variable bindings). See chapter 3.

Herbrand interpretations

Let P be a logic program based on the vocabulary A

Herbrand universe

The Herbrand universe of P (A really) is the set of all ground terms that can be built using constants and functors in P (A). Denoted U_P (U_A).

Herbrand base

The Herbrand base of P (A) is the set of all ground atoms that can be built using U_P and the predicate symbols of P (A). Denoted B_P (B_A).

Example

Vocabulary:

$$A = \{\texttt{volvo}; \texttt{owner}/1; \texttt{owns}/2, \texttt{happy}/1\}$$

Herbrand universe:

$$U_A = \{\texttt{volvo}, \texttt{owner}(\texttt{volvo}), \texttt{owner}(\texttt{owner}(\texttt{volvo})), \ldots\}$$

Herbrand base:

$$B_A = \{\texttt{happy}(s) \mid s \in U_A\} \cup \{\texttt{owns}(s,t) \mid s,t \in U_A\}$$

Herbrand interpretations

A Herbrand interpretation of P is an interpretation $\Im$ where $|\Im| = U_P$ and where:

- $c_\Im = c$ for every constant c;
- $f_\Im(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for every functor f/n;
- $p_\Im$ is a subset of $\underbrace{U_P \times \cdots \times U_P}_{n}$ for every predicate symbol p/n.

That is, the interpretation of a ground term is the term itself!

Observation I

Since all ground terms are interpreted as themselves, it is sufficient to specify the interpretation of the predicate symbols when describing a Herbrand interpretation; in other words, to specify a Herbrand interpretation $\Im$ it is sufficient to specify, for each predicate symbol, the set:

$$\{\langle t_1, \ldots, t_n \rangle \in U_P^n \mid p(t_1, \ldots, t_n) \text{ is true in } \Im\}$$

Observation II

Instead of describing a Herbrand interpretation $\Im$ as a family of sets we usually describe $\Im$ as a single set of all ground atoms that are true in $\Im$.

$$\Im = \{p(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n) \text{ is true in } \Im\}$$

Example

Alternative I

$$\begin{aligned}
\mathtt{owns}_{\Im} &= \{\langle \mathtt{owner(volvo)}, \mathtt{volvo}\rangle, \ldots\} \\
\mathtt{happy}_{\Im} &= \{\langle \mathtt{owner(volvo)}\rangle, \ldots\}
\end{aligned}$$

Alternative II

$$\begin{aligned}
\Im &= \{\mathtt{owns(owner(volvo), volvo)}, \ldots, \\
&\quad\ \mathtt{happy(owner(volvo))}, \ldots\}
\end{aligned}$$

Ground instances of P

Let C be a definite clause of the form

$$A_0 \leftarrow A_1, \ldots, A_n \quad (n \geq 0)$$

(C is considered to be a fact if $n = 0$.)

By a *ground instance* of C we mean the same clause with all variables replaced by ground terms (several occurrences of the same variable are replaced by the same term):

By $ground(C)$ we mean the set of all ground instances of C.

If P is a definite program then

$$ground(P) = \{C' \mid \exists C \in P \text{ s.t. } C' \in ground(C)\}$$

Why Herbrand Interpretations?

For an arbitrary interpretation $\Im$:

$$\Im \models_\sigma \forall X(happy(X) \leftarrow owns(X, volvo))$$
iff
$$\Im \models_{\sigma[X \mapsto a]} happy(X) \leftarrow owns(X, volvo)$$
for all $a \in |\Im|$

For a Herbrand interpretation $\Im$:

$$\Im \models_\sigma \forall X(happy(X) \leftarrow owns(X, volvo))$$
iff
$$\Im \models_\sigma happy(t) \leftarrow owns(t, volvo)$$
for any $t \in U_P$

No need to worry about valuations!!!

Herbrand models

A Herbrand model of F (resp. P) is a Herbrand interpretation which is a model of F (resp. all formulas in P).

Observation

A ground atom A is true in a Herbrand interpretation $\Im$ iff $A \in \Im$.

Theorem

Let P be a set of definite clauses (facts/rules/goals) and M be an arbitrary model of P. Then:

$$\Im := \{A \in B_P \mid M \models A\}$$

is a Herbrand model of P.

Theorem

Let $\{M_1, M_2, \ldots\}$ be a non-empty set of Herbrand models of P. Then also $\Im := \bigcap\{M_1, M_2, \ldots\}$ is a Herbrand model of P.

The Least Herbrand model

The intersection of all Herbrand models of P is called the least Herbrand model of P and is denoted M_P.

Theorem

$$M_P = \{A \in B_P \mid P \models A\}$$

“Construction” of M_P

Observation

In order for $\Im$ to be a model of P it is required that:

- If A is a ground instance of a fact then $A \in \Im$, and
- If $A \leftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq \Im$ then $A \in \Im$.

Immediate consequence operator

$$
\begin{aligned}
T_P(x) :=& \\
&\{A \in B_P \;\mid\; A \leftarrow A_1, \ldots, A_n \in ground(P) \\
&\qquad\qquad\quad \text{and } \{A_1, \ldots, A_n\} \subseteq x\}
\end{aligned}
$$

Theorem

$$M_P = T_P^n(\emptyset) \quad \text{when } n \to \infty$$

Example

```
gp(X,Y) :- p(X,Z), p(Z,Y).

p(X,Y) :- f(X,Y).
p(X,Y) :- m(X,Y).

f(adam,bill).
f(adam,carol).
f(bill,eve).
m(carol,david).
```

Example

- $\Im_0 = \emptyset$

- $\Im_1 = T_P(\emptyset) = \{f(a,b), f(a,c), f(b,e), m(c,d)\}$
  [$f(a,b) \in \Im_1$ since $(f(a,b) \leftarrow\ ) \in ground(P)$ and $\emptyset \subseteq \emptyset$.]

- $\Im_2 = T_P(\Im_1) = T_P^2(\emptyset) = \{p(a,b), p(a,c), p(b,e), p(c,d)\} \cup \Im_1$
  [$p(a,b) \in \Im_2$ since $(p(a,b) \leftarrow f(a,b)) \in ground(P)$ and $\{f(a,b)\} \subseteq \Im_1$.]

- $\Im_3 = T_P(\Im_2) = T_P^3(\emptyset) = \{gp(a,d), gp(a,e)\} \cup \Im_2$
  [$gp(a,d) \in \Im_3$ since $(gp(a,d) \leftarrow p(a,c), p(c,d)) \in ground(P)$ and $\{p(a,c), p(c,d)\} \subseteq \Im_2$.]

- $\Im_4 = T_P(\Im_3) = T_P^4(\emptyset) = \Im_3$

SLD-Resolution: Overview

- Substitutions;
- Unification;
- SLD-derivations;
- Soundness and completeness.

Substitutions

A substitution is a finite set
$\{X_1/t_1, \ldots, X_n/t_n\}$ where:

- every t_i is a term;
- every X_i is a variable distinct from t_i;
- if $i \neq j$ then $X_i \neq X_j$.

The empty substitution $\{\}$ is denoted ϵ.

Let θ be a substitution $\{X_1/t_1, \ldots, X_n/t_n\}$.

Domain and Range

The domain $Dom(\theta)$ of θ is $\{X_1, \ldots, X_n\}$ and the range $Range(\theta)$ is the set of all variables occurring in $t_1, \ldots, t_n$.

Application

Let E be a term or formula. The application $E\theta$ of θ to E is the term/formula obtained from E by simultaneously replacing all occurrences of X_i by t_i.

$E\theta$ is called an *instance* of E.

Composition

Let $\theta := \{X_1/s_1, \ldots, X_m/s_m\}$ and $\sigma := \{Y_1/t_1, \ldots, Y_n/t_n\}$ be substitutions. The composition $\theta\sigma$ of θ and σ is the substitution obtained from

$$\{X_1/s_1\sigma, \ldots, X_m/s_m\sigma, Y_1/t_1, \ldots, Y_n/t_n\}$$

by removing all $X_i/s_i\sigma$ where $X_i = s_i\sigma$ and all Y_i/t_i where $Y_i \in Dom(\theta)$.

More general substitution

A substitution θ is more general than σ ($\sigma \preceq \theta$) iff there exists a substitution ω such that $\theta\omega = \sigma$.

Theorem

Let θ, σ and γ be substitutions and E a term/formula. Then

- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$;
- $E(\theta\sigma) = (E\theta)\sigma$;
- $\epsilon\theta = \theta\epsilon = \theta$.

Unification

A *structure* is a term or an atomic formula.

Unifier

A unifier of two structures s and t is a substitution θ such that $s\theta = t\theta$.

Most general unifier (mgu)

A unifier θ of s and t is called a most general unifier of s and t iff $\sigma \preceq \theta$ for every unifier σ of s and t. NB: Two unifiable structures have at least one mgu (usually infinitely many).

Solved form

A set of equation $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is in solved form iff $s_1, \ldots, s_n$ are distinct variables none of which occur in $t_1, \ldots, t_n$.

Solution

A substitution θ is a solution to a set of equations $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ iff θ is a unifier of s_i and t_i ($1 \leq i \leq n$).

Theorem

If $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ is in solved form then $\{X_1/t_1, \ldots, X_n/t_n\}$ is an mgu of X_i and t_i ($1 \leq i \leq n$).

select an arbitrary $s \doteq t \in E$;

case $s \doteq t$ **of**

$f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)$

where $n \geq 0 \Rightarrow$

replace equation by $s_1 \doteq t_1, \ldots, s_n \doteq t_n$;

$f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)$

where $f/m \neq g/n \Rightarrow$

halt with $\perp$;

$X \doteq X \Rightarrow$

remove the equation;

$t \doteq X$ where t is not a variable $\Rightarrow$

replace equation by $X \doteq t$;

$X \doteq t$ where $X \neq t$ and X has more than one occurrence in $E \Rightarrow$

if X is a proper subterm of t **then**

halt with $\perp$

else

replace all other occurrences of X by t;

esac

Theorem

The algorithm always terminates. If s and t are unifiable then the algorithm returns a solved form whose mgu is an mgu of s and t. Otherwise the algorithm returns $\perp$.

Renaming

A substitution $\theta := \{X_1/Y_1, \ldots, X_n/Y_n\}$ where $Y_1, \ldots, Y_n$ is a permutation of $X_1, \ldots, X_n$ is called a renaming. The substitution $\{Y_1/X_1, \ldots, Y_n/X_n\}$ is called the inverse of θ (denoted θ^{-1}).

Theorem

Let θ and σ be mgu's of s and t. Then there exists a renaming γ such that $\theta\gamma = \sigma$ (and $\sigma\gamma^{-1} = \theta$).

Theorem

If θ is an mgu of s and t and σ a renaming, then $\theta\sigma$ is also an mgu of s and t.

In practice

The previous algorithm is worst-case exponential in the size of the structures. Take for instance

$$g(X_1, \ldots, X_n) = g(f(X_0, X_0), \ldots, f(X_{n-1}, X_{n-1})).$$

The reason is the *occurs check* (i.e. checking if X is a proper subterm of t).

There are also polynomial algorithms, but most Prolog implementations use the exponential algorithm, and simply drop the occurs check.

This rarely makes a difference, but does make Prolog unsound!!!

SLD-resolution rule

Let $H \leftarrow B_1, \ldots, B_n$ be a program clause renamed apart from $\leftarrow A_1, \ldots, A_i, \ldots, A_m$, and let θ be an mgu of A_i and H. Then:

$$\frac{\leftarrow A_1, \ldots, A_i, \ldots, A_m \qquad H \leftarrow B_1, \ldots, B_n}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta}$$

SLD-derivation

Let G_0 be a goal. An SLD-derivation of G_0 is a finite/infinite sequence:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n \cdots$$

of goals and (renamed) program clauses such that:

$$\frac{G_i \qquad\qquad C_i}{G_{i+1}}$$

```
gp(X,Y) :- p(X,Z), p(Z,Y).

p(X,Y) :- f(X,Y).
p(X,Y) :- m(X,Y).

f(adam,tom).
f(adam,mary).
f(tom,david).

m(mary,anne).
```

```
inv(0,1).
inv(1,0).

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).

nand(X,Y,Z) :- and(X,Y,W), inv(W,Z).
```

Computation rule

A computation rule $\Re$ is a (partial) function that given a goal returns an atom in that goal.

SLD-refutation

An SLD-refutation of G_0 is a finite SLD-derivation

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n$$

where $G_n = \Box$.

Failed derivation

A finite SLD-derivation

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n$$

is said to be failed if the selected atom in G_n does not unify with any program clause head.

Complete SLD-derivation

An SLD-derivation is complete if it is a refutation, a failed or infinite derivation.

Let

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n$$

be an SLD-derivation

Computed substitution

If θ_i is mgu i of the derivation then

$$\theta_1\theta_2 \ldots \theta_n$$

is called the computed substitution in the derivation.

Computed answer-substitution

The computed answer-substitution in a refutation of G_0 is the computed substitution of the refutation restricted to the variables occurring in G_0.

Let P be a logic program;
Let $\Re$ be a computation rule

SLD-tree

The SLD-tree of a goal G_0 is a tree where

- the root of the tree is G_0;

- if G_i is a node in the tree then G_i has a child G_{i+1} (connected via a branch labelled "C_i") iff there exists an SLD-derivation

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_i \overset{C_i}{\rightsquigarrow} G_{i+1}$$

with the computation rule $\Re$.

Soundness and completeness

Theorem (soundness)

Let P be a logic program, $\Re$ a computation rule and θ an $\Re$-computed answer-substitution of the goal $\leftarrow A_1, \ldots, A_n$. Then $\forall((A_1 \wedge \ldots \wedge A_n)\theta)$ is a logical consequence of P.

Theorem (completeness)

Let P be a logic program and $\Re$ a computation rule. If $\forall(A_1 \wedge \ldots \wedge A_n)\sigma$ is a logical consequence of P then there is a refutation of $\leftarrow A_1, \ldots, A_n$ with $\Re$-computed answer-substitution θ such that $(A_1 \wedge \ldots \wedge A_n)\sigma$ is an instance of $(A_1 \wedge \ldots \wedge A_n)\theta$.

Example

```
% leq(X,Y) - X is less than or equal to Y
leq(0, Y).
leq(s(X), s(Y)) :- leq(X, Y).

-------------------

:- leq(0, N).
yes
```

That is $P \models \forall N \; leq(0, N)$.

Note that it is impossible to obtain e.g. the answer $N = s(0)$). However, we get a more general answer.

Negation: Overview

- Closed World Assumption;
- Negation as Failure;
- Completion;
- SLDNF-resolution (part I);
- General (alt. normal) logic programs;
- Stratified logic programs;
- SLDNF-resolution (part II).

Program:

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

female(a).
female(d).

mother(X) :- parent(X,Y), female(X).
```

Least Herbrand model:

```
parent(a,b).
parent(a,c).
parent(c,d).
female(a).
female(d).
mother(a).
```

Program:

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

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

Least Herbrand model:

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

Closed World Assumption

Background Definite programs can only be used to describe positive knowledge; it is not possible to describe objects that are *not* related.

Solution I Closed world assumption:

$$\frac{P \not\models A}{\neg A}$$

Problem $P \not\models A$ is undecidable.

Negation as (finite) Failure

Solution II An SLD-tree is finitely failed iff it is finite and does not contain any refutations.

Observation If $\leftarrow A$ has a finitely failed SLD-tree then $P \not\models A$. (Follows from the soundness and completeness of SLD-resolution.)

The NAF rule

$$\frac{\leftarrow A \text{ has a finitely failed SLD-tree}}{\neg A}$$

Problem The NAF rule is not sound.

Completion

Thesis The program contains information that is not written out explicitly. The *completed program* is the program obtained after addition of the missing information.

Observation $\{a \leftarrow b, a \leftarrow c\} \equiv \{a \leftarrow b \vee c\}$.

Principle An implication $a \leftarrow b$ is replaced by an equivalence $a \leftrightarrow b$.

Let $Y_1, \ldots, Y_i$ be all variables in
$p(t_1, \ldots, t_m) \leftarrow A_1, \ldots, A_n$.

Step 1 Replace the clause by

$$p(X_1, \ldots, X_m) \leftarrow \\ \quad \exists Y_1 \ldots Y_i (X_1 \doteq t_1, \ldots, X_m \doteq t_m, A_1, \ldots, A_n)$$

Step 2 Take all clauses

$$\begin{array}{c} p(X_1, \ldots, X_m) \leftarrow E_1 \\ \vdots \\ p(X_1, \ldots, X_m) \leftarrow E_j \end{array}$$

that define p/m and replace by

$$\begin{array}{ll} p(X_1, \ldots, X_m) \leftarrow E_1 \vee \ldots \vee E_j & (j > 0) \\ p(X_1, \ldots, X_m) \leftarrow \Box & (j = 0) \end{array}$$

Step 3 Replace all implications with equivalences.

Step 4 Add the "free equality axioms":

$$X \doteq X$$
$$X \doteq Y \rightarrow Y \doteq X$$
$$X \doteq Y \wedge Y \doteq Z \rightarrow X \doteq Z$$
$$X_1 \doteq Y_1 \wedge \ldots \wedge X_m \doteq Y_m \rightarrow$$
$$\quad f(X_1, \ldots, X_m) \doteq f(Y_1, \ldots, Y_m)$$
$$X_1 \doteq Y_1 \wedge \ldots \wedge X_m \doteq Y_m \rightarrow$$
$$\quad (p(X_1, \ldots, X_m) \rightarrow p(Y_1, \ldots, Y_m))$$
$$f(X_1, \ldots, X_m) \neq g(Y_1, \ldots, Y_n) \text{ if } f/m \neq g/n$$
$$f(X_1, \ldots, X_m) \doteq f(Y_1, \ldots, Y_m) \rightarrow$$
$$\quad X_1 \doteq Y_1 \wedge \ldots \wedge X_m \doteq Y_m$$
$$f(\ldots X \ldots) \neq X$$

Soundness of “Negation as Failure”

Theorem Let P be a definite program. If $\leftarrow A$ has a finitely failed SLD-tree then $comp(P) \models \forall \neg A$.

Completeness of “Negation as Failure”

Theorem Let P be a definite program. If $comp(P) \models \forall \neg A$ then there exists a finitely failed SLD-tree of $\leftarrow A$.

SLDNF-resolution for definite programs

A general goal is an expression

$$\leftarrow L_1, \ldots, L_n.$$

where each L_i is an atom (positive literal) or a negated atom (negative literal).

Combine SLD-resolution and "Negation as Failure"

Given a general goal — if the selected literal is positive then the next goal is obtained in the usual way. If the selected literal is negative ($\neg A$) and $\leftarrow A$ has a finitely failed SLD-tree then the next goal is obtained by removing $\neg A$ from the goal.

Soundness of SLDNF

Theorem Let P be a definite program and $\leftarrow L_1, \ldots, L_n$ a general goal. If $\leftarrow L_1, \ldots, L_n$ has an SLDNF-refutation with computed answer-substitution θ then $\forall(L_1 \wedge \cdots \wedge L_n)\theta$ is a logical consequence of $comp(P)$.

No completeness!!!

General (or normal) programs

A general clause is a clause of the form

$$A \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where $L_1, \ldots, L_n$ are positive/negative literals.

Completion

Completion of a general program is obtained in the same way as for definite programs. (Negative literals are handled like positive literals.)

Stratified programs

Problem Completion of a general program can be inconsistent (unsatisfiable).

Limitation A stratified program is a general program where "no relation is defined in terms of its own complement". That is, no predicate symbol depends on its own negation.

Stratified programs

A general program P is stratified iff there exists a partitioning $P_1, \ldots, P_n$ of P such that

- if $p(\ldots) \leftarrow \ldots, q(\ldots), \ldots \in P_i$ then $DEF(q) \subseteq P_1 \cup \ldots \cup P_i$.

- if $p(\ldots) \leftarrow \ldots, \neg q(\ldots), \ldots \in P_i$ then $DEF(q) \subseteq P_1 \cup \ldots \cup P_{i-1}$.

Theorem Completion of a stratified program is always consistent.

SLDNF-resolution for general programs

Let P be a general program, G_0 a general goal and $\Re$ a computation rule. The *SLDNF-forest* of G_0 is the least forest (modulo renaming) such that

1. G_0 is a root of one tree.

2. if G is a node and $\Re(G) = A$ then G has a child G' for each clause C such that G' is obtained from G and C. If there is no such clause, G has a single child $\mathbf{FF}$;

3. if G is a node of the form $\leftarrow L_1, \ldots, L_{i-1}, \neg A, L_{i+1}, \ldots, L_{i+j}$ and $\Re(G) = \neg A$, then

Cont'd

- the forest contains a tree with the root $\leftarrow A$;

- if the tree with the root $\leftarrow A$ has a leaf $\Box$ with the *empty* computed answer-substitution, then G has a child $\mathbf{FF}$.

- if the tree with root $\leftarrow A$ is finite and all leaves are $\mathbf{FF}$, then G has a single child $\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_{i+j}$.

Soundness of SLDNF-resolution

Let P be a general program, $\leftarrow L_1, \ldots, L_n$ a general goal and $\Re$ a computation rule. If θ is a computed answer-substitution in an SLDNF-refutation of $\leftarrow L_1, \ldots, L_n$ then $\forall((L_1 \wedge \ldots \wedge L_n)\theta)$ is a logical consequence of comp(P).

```
father(X) :-
    parent(X,Y),
    \+ mother(X,Y).

disjoint([],X).
disjoint([X|Xs],Ys) :-
    \+ member(X,Ys),
    disjoint(Xs,Ys).
```

```
founding(X) :-
    on(Y,X),
    on_ground(X).

on_ground(X) :-
    \+ off_ground(X).

off_ground(X) :-
    on(X,Y).

on(c,b).
on(b,a).
```

```
go_well_together(X,Y) :-
    \+ incompatible(X,Y).

incompatible(X,Y) :-
    \+ likes(X,Y).
incompatible(X,Y) :-
    \+ likes(Y,X).

likes(X,Y) :-
    harmless(Y).
likes(X,Y) :-
    eats(X,Y).

harmless(rabbit).

eats(python,rabbit).
```

```
father(X,Y) :-
    parent(X,Y),
    \+ mother(X,Y).

parent(a,b).
parent(c,b).

mother(a,b).
```

```
father(X,Y) :-
    parent(X,Y),
    \+ mother(X,Y).

mother(X,Y) :-
    parent(X,Y),
    \+ father(X,Y).

parent(a,b).
parent(c,b).
```

```
on_top(X) :-
    \+ blocked(X).

blocked(X) :-
    on(Y,X).

on(a,b).

%---------------------
| ?- \+ on_top(b).

| ?- \+ on_top(X).
```

Logic and Grammars: Overview

- Context free languages;
- Context sensitive languages;
- Definite Clause Grammars (DCGs);
- DCGs and Prolog.

Context free languages

- A context free grammar is a triple $\langle N, T, P\rangle$ where:
  - N is a finite set of *non-terminals*;
  - T is a finite set of *terminals* (and $N \cap T = \emptyset$);
  - $P \subseteq N \times (N \cup T)^*$ is a finite set of *production rules*.

- Examples of production rules:

$$
\begin{array}{lcl}
\langle expr \rangle & \rightarrow & \langle expr \rangle + \langle expr \rangle \\
\langle sent \rangle & \rightarrow & \langle np \rangle \, \langle vp \rangle
\end{array}
$$

Derivations

- Let $\alpha, \beta, \gamma \in (N \cup T)^*$. We say that $\alpha A \gamma$ *directly derives* $\alpha\beta\gamma$ iff $A \to \beta \in P$. Denoted

$$\alpha A \gamma \Rightarrow \alpha\beta\gamma$$

- We say that α_1 *derives* α_n iff there exists a sequence $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$. Denoted

$$\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n$$

- A terminal string $\alpha \in T^*$ *is in the language of* A iff $A \stackrel{*}{\Rightarrow} \alpha$.

Example: Context free grammar

$$
\begin{aligned}
&\langle sent \rangle \rightarrow \langle np \rangle\, \langle vp \rangle \\
&\langle np \rangle \rightarrow \texttt{the}\ \langle n \rangle \\
&\langle vp \rangle \rightarrow \texttt{runs} \\
&\langle n \rangle \rightarrow \texttt{engine} \\
&\langle n \rangle \rightarrow \texttt{rabbit}
\end{aligned}
$$

Naive implementation

$$
\begin{aligned}
&sent(Z) \leftarrow append(X, Y, Z), np(X), vp(Y).\\
&np([the|X]) \leftarrow n(X).\\
&vp([runs]).\\
&n([engine]).\\
&n([rabbit]).
\end{aligned}
$$

$$
\begin{aligned}
&append([\,], Xs, Xs).\\
&append([X|Xs], Ys, [X|Zs]) \leftarrow\\
&\qquad\qquad append(Xs, Ys, Zs).
\end{aligned}
$$

Usage of "Difference Lists"

- Assume that "$-/2$" denotes a partial function which given two strings $x_1 \ldots x_{m-1} x_m \ldots x_n$ and $x_m \ldots x_n$ returns the string $x_1 \ldots x_{m-1}$.

- Example

$$sent(X_0 - X_2) \leftarrow np(X_0 - X_1), vp(X_1 - X_2).$$

$$\underbrace{x_1 \ldots x_{i-1}\, \underbrace{x_i \ldots x_{j-1}\, \underbrace{x_j \ldots x_k}_{X_2}}_{X_1}}_{X_0}$$

Two Alternatives

$$
\begin{array}{l}
sent(X_0 - X_2) \leftarrow np(X_0 - X_1), vp(X_1 - X_2). \\
np(X_0 - X_2) \leftarrow \text{'C'}(X_0, the, X_1), n(X_1 - X_2). \\
vp(X_0 - X_1) \leftarrow \text{'C'}(X_0, runs, X_1). \\
n(X_0 - X_1) \leftarrow \text{'C'}(X_0, engine, X_1). \\
n(X_0 - X_1) \leftarrow \text{'C'}(X_0, rabbits, X_1). \\
\text{'C'}([X|Y], X, Y).
\end{array}
$$

$$
\begin{array}{l}
sent(X_0 - X_2) \leftarrow np(X_0 - X_1), vp(X_1 - X_2). \\
np([the|X_1] - X_2) \leftarrow n(X_1 - X_2). \\
vp([runs|X_1] - X_1). \\
n([engine|X_1] - X_1). \\
n([rabbit|X_1] - X_1).
\end{array}
$$

Partial deduction

```
grandparent(X,Y) :-
    parent(X,Z), parent(Z,Y).
    -----------

parent(X,Y) :-
    father(X,Y).
parent(X,Y) :-
    mother(X,Y).

%------------------------------------

grandparent(X,Y) :-
    father(X,Z), parent(Z,Y).
grandparent(X,Y) :-
    mother(X,Z), parent(Z,Y).

parent(X,Y) :-
    father(X,Y).
parent(X,Y) :-
    mother(X,Y).
```

Context sensitive languages

- Some languages cannot be described by context free grammars. For instance

$$\begin{aligned} ABC &= \{a^n b^n c^n \mid n \geq 0\} \\ &= \{\epsilon, abc, aabbcc, aaabbbccc, \ldots\} \end{aligned}$$

- The language ABC can be expressed in Prolog

$$\begin{aligned}
& abc(X_0\text{–}X_3) \leftarrow \\
& \qquad\qquad a(N, X_0\text{–}X_1), \\
& \qquad\qquad b(N, X_1\text{–}X_2), \\
& \qquad\qquad c(N, X_2\text{–}X_3). \\
& a(0, X_0\text{–}X_0). \\
& a(s(N), [a|X_1]\text{–}X_2) \leftarrow a(N, X_1\text{–}X_2). \\
& b(0, X_0\text{–}X_0). \\
& b(s(N), [b|X_1]\text{–}X_2) \leftarrow b(N, X_1\text{–}X_2). \\
& c(0, X_0\text{–}X_0). \\
& c(s(N), [c|X_1]\text{–}X_2) \leftarrow c(N, X_1\text{–}X_2).
\end{aligned}$$

Definite Clause Grammars (DCGs)

- A Definite Clause Grammar is a triple $\langle N, T, P \rangle$ where
  - N is a finite/infinite set of atoms;
  - T is a finite/infinite set of terms (and $N \cap T = \emptyset$);
  - $P \subseteq N \times (N \cup T)^*$ is a finite set of production rules.

Derivations

- Let $\alpha, \beta, \gamma \in (N \cup T)^*$. We say that $\alpha A \gamma$ *directly derives* $(\alpha\beta\gamma)\theta$ iff $A' \rightarrow \beta \in P$ and $mgu(A, A') = \theta$. Denoted

$$\alpha A \gamma \Rightarrow (\alpha\beta\gamma)\theta$$

- We say that α_1 derives α_n (denoted $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n$) iff there exists a sequence

$$\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$$

- A terminal string $\alpha \in T^*$ *is in the language of* A iff $A \stackrel{*}{\Rightarrow} \alpha$.

Example of DCG

```
sent(s(X,Y)) --> np(X, N)\ vp(Y, N).
np(john, singular(3)) --> [john].
np(they,plural(3)) --> [they].
vp(run,plural(X)) --> [run].
vp(runs,singular(3)) --> [runs].
```

Semantical (context sensitive) constraints

The following DCG describes the language $\{a^{2n}b^{2n}c^{2n} \mid n \geq 0\}$

```
abc      -->   a(N), b(N), c(N), even(N).

a(0)     -->   [].
a(s(N))  -->   [a], a(N).

...

even(0)  -->   [].
even(s(s(N))) -->  even(N).
```

Note

- The language of $even(X)$ contains only the string ϵ!!!

- This may be emphasized by writing

```
abc --> a(N), b(N), c(N), {even(N)}.
```

- and by defining $even/1$ as a logic program

$$
\begin{array}{l}
even(0). \\
even(s(s(X))) \leftarrow even(X).
\end{array}
$$

DCGs and Prolog

- Every production rule in a DCG can be compiled into a Prolog clause;

- The resulting Prolog program can be used as a (top-down) parser for the language (cf. “recursive descent”);

Compilation

- Assume that $X_0, \ldots, X_m$ are distinct variables that do not occur in

$$p(t_1, \ldots, t_n) \ \rightarrow \ T_1, \ldots, T_m$$

- The Prolog program will then contain a clause

$$p(t_1, \ldots, t_n, X_0, X_m) \leftarrow T'_1, \ldots, T'_m.$$

where each T'_i, $(1 \leq i \leq m)$, is of the form

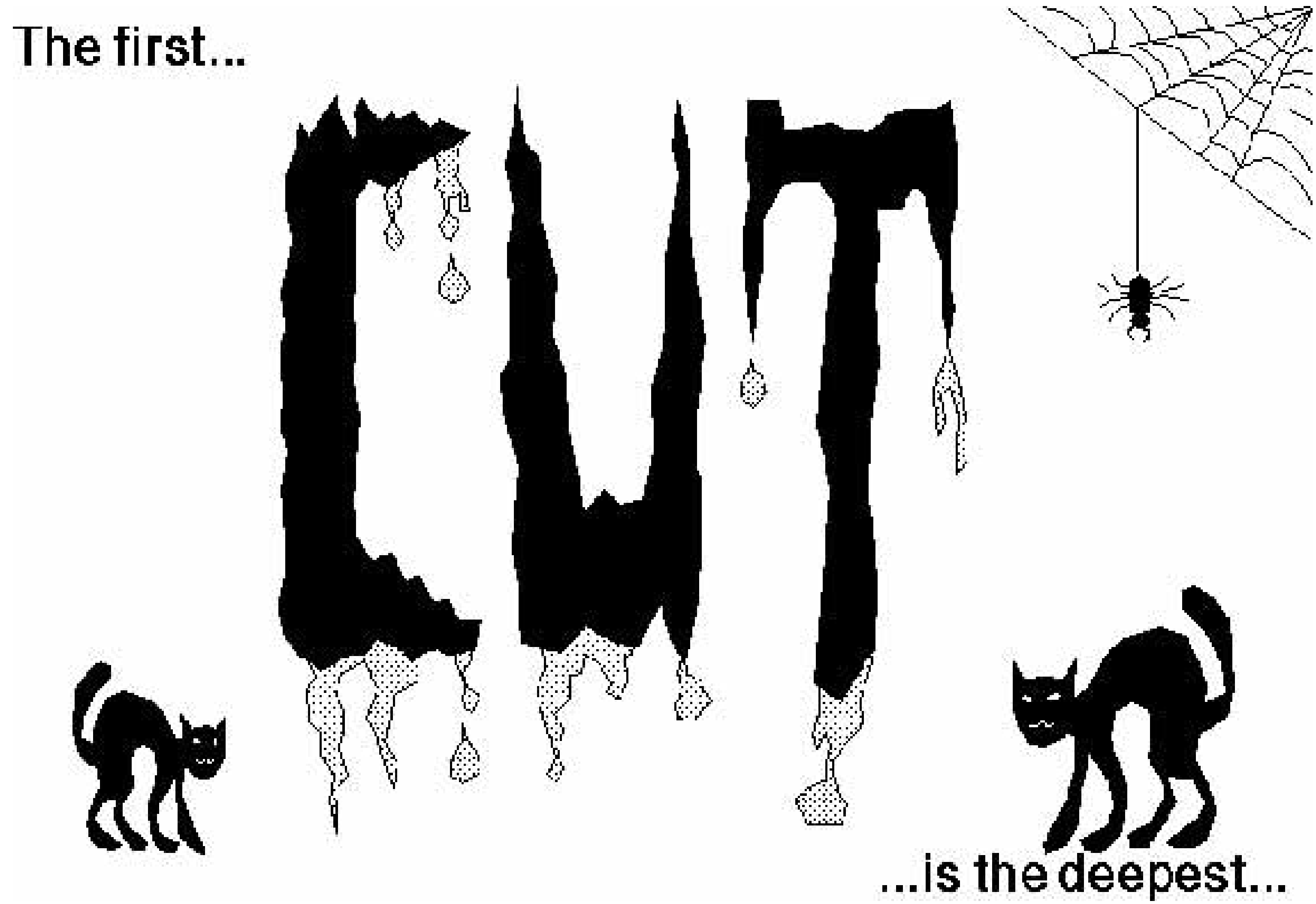
$$\begin{array}{rl} q(t_1, \ldots, t_n, X_{i-1}, X_i) & \text{if } T_i = q(t_1, \ldots, t_n) \\ \text{'C'}(X_{i-1}, t, X_i) & \text{if } T_i = [t] \\ T, X_{i-1} = X_i & \text{if } T_i = \{T\} \\ X_{i-1} = X_i & \text{if } T_i = [\,] \end{array}$$

Example

```
sent --> np, vp.
np --> [the], n.
vp --> [runs].
n --> [boy].

% Translates into...

sent(S0,S2) :- np(S0,S1), vp(S1,S2).
np(S0,S2) :- 'C'(S0,the,S1), n(S1,S2).
vp(S0,S1) :- 'C'(S0,runs,S1).
n(S0,S1) :- 'C'(S0,boy,S1).

'C'([X|Xs],X,Xs).
```

Summary

- Logic programming can be used to define
  - (Regular languages);
  - Context free languages;
  - Context sensitive languages;
  - (Recursively enumerable languages).
- Definite Clause Grammars (DCGs);
- Compilation of DCGs into Prolog.


The first...
CUT
...is the deepest...

Examples

```
% Membership in a ordered binary tree
member(X, node(Left, X, Right)).
member(X, node(Left, Y, Right)) :-
    X < Y,
    member(X, Left).
member(X, node(Left, Y, Right)) :-
    X > Y,
    member(X, Right).


% Property of being a father
father(X) :-
    parent(X, Y), male(X).
```

General

- Prolog constructs the SLD(NF)-tree by a depth-first search in combination with backtracking.

- By means of cut (!) the user can prohibit the Prolog engine from exploring certain branches in the tree.

- Cut (!) may only occur in the righthand sides of clauses and can be viewed as a regular (nullary) atom.

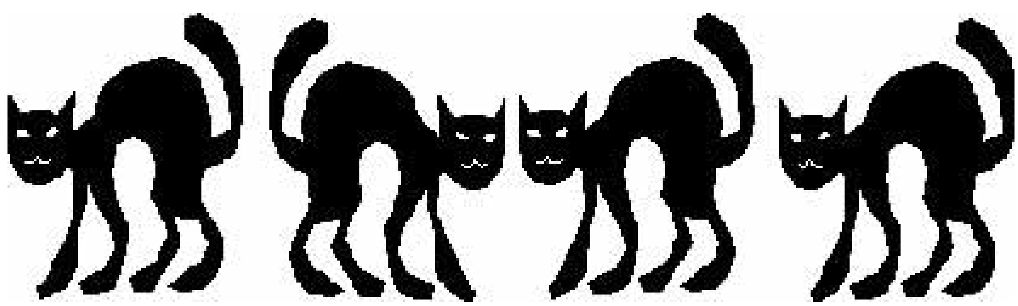
Principles

- Two principal uses
  - Prune infinite and failed branches (green cut);
  - Prune refutations (red cut).

- Acceptable "red cut":
  - Prune multiple occurrences of the same answer.

The Golden Rule

First write a correct program without cuts.
Then add cuts in approprate places to
improve the efficiency.



Constraint logic programming

- Constraints

- Operations on constraints

- Constraint Logic Programming
  - Language
  - Operational semantics
  - Examples

Constraint

Given a set of variables, a *constraint* is a restriction on the possible values of the variables.

Example

Variables: X, Y.

Constraint I: $X^2 + Y^2 \leq 4$

Constraint II: $Y \geq 2 - 2 \cdot X$

Solution

The constraint $X^2 + Y^2 \leq 4$ has a set of *solutions* – variable assignments when the constraint is true, e.g:

$\{X \mapsto 2, Y \mapsto 0\}$
$\{X \mapsto 0, Y \mapsto 2\}$
$\{X \mapsto 1, Y \mapsto 1\}$

A mapping from variables to values is called a *valuation*. A valuation where the constraint is true is called a *solution*.

Domain of a constraint

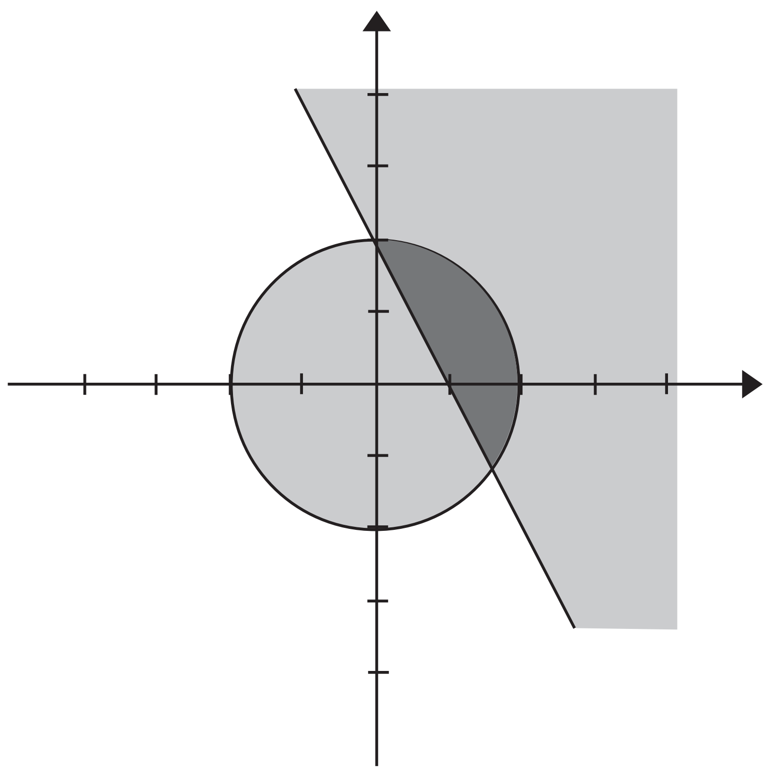
Whether a constraint has a solution or not depends on the values that the variables can take.

The constraint $X^2 = 2$ has a real solution, but not an integer or a rational solution.

The set of all possible values of the variables is called the *domain* of the constraint.

Conjunctive constraints

The conjunction of the primitive constraints $X^2 + Y^2 \leq 4$ and $Y \geq 2 - 2 \cdot X$ is a new (conjunctive) constraint:



Sets of primitive constraints represent conjunctive constraints.

Properties of constraints

A constraint is said to be *satisfiable* iff it has at least one solution.

A constraint C_1 *implies* a constraint C_2 (written $C_1 \models C_2$) iff every solution of C_1 is also a solution of C_2.

Two constraints are *equivalent* if they have the same set of solutions.

Optimal solutions

A solution σ of a set of constraints S is *maximal subject to* an expression E if $\sigma(E)$ is greater than $\sigma'(E)$ for any solution σ' of S.

Example

The solution $\{X \mapsto 1.6, Y \mapsto -1.2\}$ is a maximal solution of

$$\begin{array}{rcl} X^2 + Y^2 & \leq & 4 \\ Y & \geq & 2 - 2 \cdot X \end{array}$$

subject to $-Y$.

Constraint Logic Programming

```
sorted([]).
sorted([X]).
sorted([Fst,Snd|Rst]) :-
      Fst =< Snd, sorted([Snd|Rst]).

------------------------------------

:- sorted([X1,X2,X3]).

ARITHMETIC ERROR!!!
```

Language

- Functors and predicate symbols divided into:
  - Uninterpreted symbols (Herbrand terms/atoms);
  - Interpreted symbols (constraints).
- Special *solvers* handle constraints;
- SLD(NF)-resolution is used for Herbrand atoms;

Language (cont'd.)

- A clause is an expression

$$A_0 \leftarrow C_1, \ldots, C_m, A_1, \ldots, A_n$$

  where

  - $A_0, \ldots, A_n$ are Herbrand atoms;
  - $C_1, \ldots, C_m$ are constraints.

- A goal is an expression

$$\leftarrow C_1, \ldots, C_m, A_1, \ldots, A_n$$

CLP(X): A Family of Languages

CLP(R) Linear equations over reals

CLP(Q) Linear equations over rationals

CLP(B) Booleans

CLP(FD) Finite domains

Example CLP(R)

```
mortgage(Loan,Years,AInt,Bal,APay) :-
    { Years>0,
      Years <= 1,
      Bal=Loan*(1+Years*AInt)-APay }.
mortgage(Loan,Years,AInt,Bal,APay) :-
    { Years>1,
      NewLoan = Loan*(1+AInt)-APay,
      Years1 = Years-1 },
    mortgage(NewLoan,Years1,AInt,Bal,APay).
-------------------------------------------
?- mortgage(120000,10,0.1,0,AnnPay).
AnnPay=19529.4

?- mortgage(Loan,10,0.1,0,19529.4).
Loan=120000

?- mortgage(Loan,10,0.1,0,AnnPay).
Loan=6.14457*AnnPay
```

Resolution with constraints

A state is a pair $(G\,;\,S)$ where G is a goal, and S is a *constraint store*. Given a program P a derivation is a sequence of states:

- $(\leftarrow A, B\,;\,S) \Rightarrow (\leftarrow A = A', B', B\,;\,S)$ if $A' \leftarrow B' \in P$

- $(\leftarrow C, G\,;\,S) \Rightarrow (\leftarrow G\,;\,\{C\} \cup S)$

- $(G\,;\,S) \Rightarrow \mathit{fail}$ if sat$(S) = \mathit{false}$;

- $(G\,;\,S) \Rightarrow (G\,;\,S')$ if S and S' are equivalent.

- $(G\,;\,\{X = t\} \cup S) \Rightarrow (G\,;\,S)\{X/t\}$

Example: Arithmetic

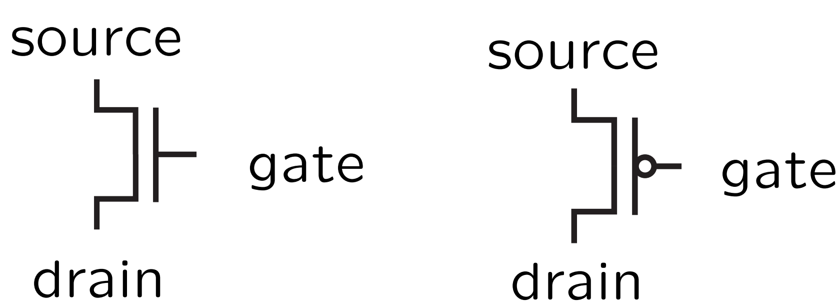
```
:- res(ser(r(10),r(20)),X).

---------------------------

res(r(X),Y) :-
    {X=Y}.
res(cell(X),Y) :-
    {Y=0}.
res(ser(X1,X2),R) :-
    {R=R1+R2}, res(X1,R1), res(X2,R2).
res(par(X1,X2),R) :-
    {1/R=1/R1+1/R2}, res(X1,R1), res(X2,R2).
```

Modeling with Boolean constraints

Boolean operations

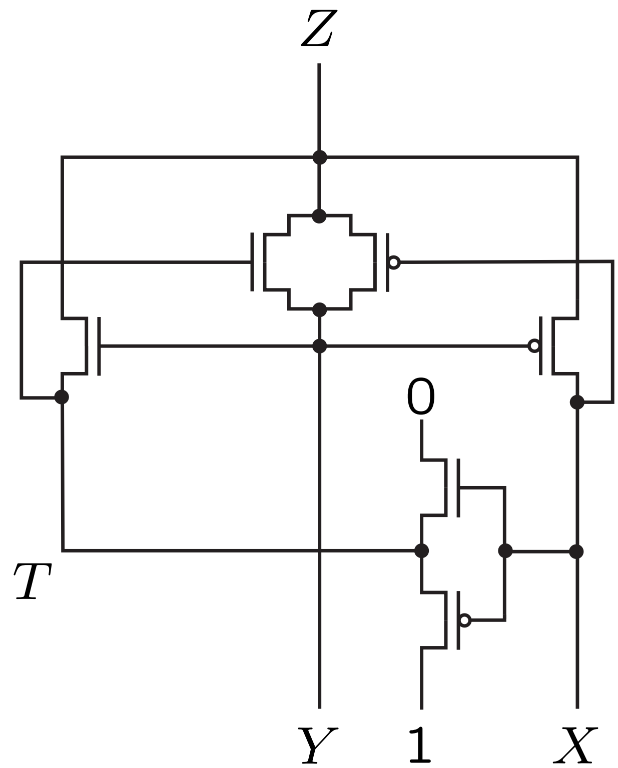
+	Disjunktion	*	Conjunction
=<	Implikation	=:=	Equivalence
#	Exclusive or	~	Negation

MOS transistors




```
nmos(S,G,D) :- sat(S * G =:= D * G).
pmos(S,G,D) :- sat(S * ~G =:= D * ~G).
```

Design of XOR-gate




```
circuit(X,Y,Z) :-
      pmos(X,Y,Z),
      pmos(1,X,T),
      nmos(T,X,0),
      nmos(T,Y,Z),
      nmos(Y,T,Z),
      pmos(Y,X,Z).
```

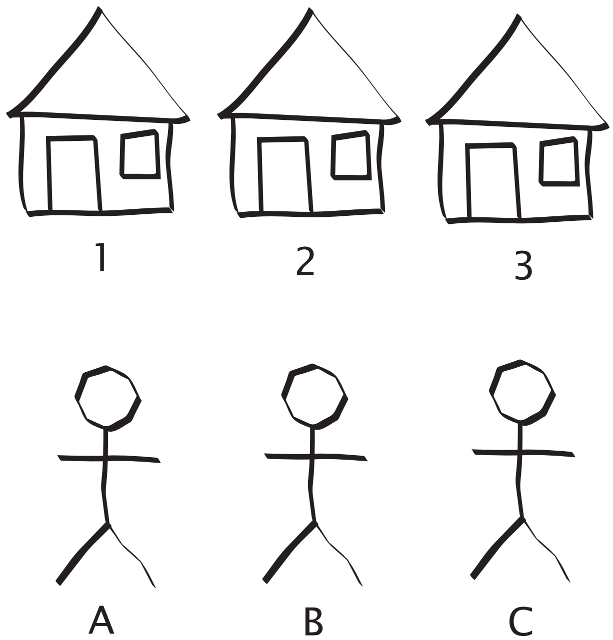
Verification of correctness

```
?- circuit(X,Y,Z), taut(Z =:= X#Y, 1).
yes
```

CLP with Finite Domains

- Constraints and constraint problems
- Primitive constraints
- CLP(FD)
- Optimization
- Global constraints

Example

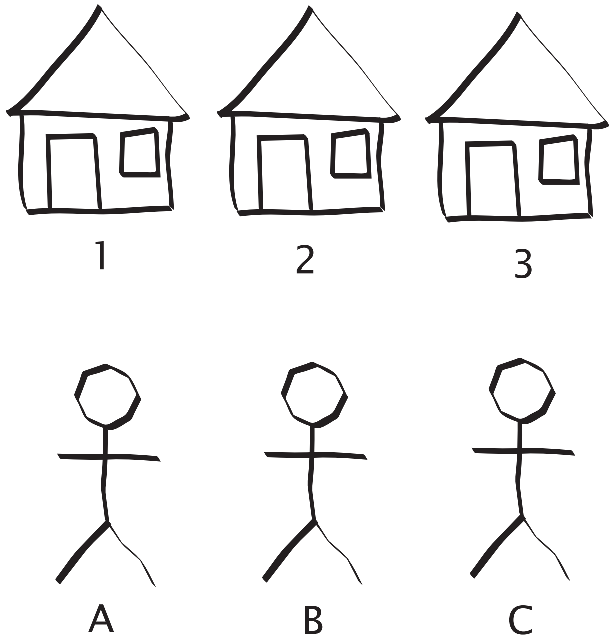


- A, B and C live in different houses
- C lives left of B
- B has two neighbors

Constraint problem

- A *constraint problem* consists of a finite set of *problem variables*,
- Each variable takes its value from a given *domain*
- Constraints are *relations* that restrict the values that can be assigned to the problem variables

Mathematical reformulation



- $A, B, C \in \{1, 2, 3\}$

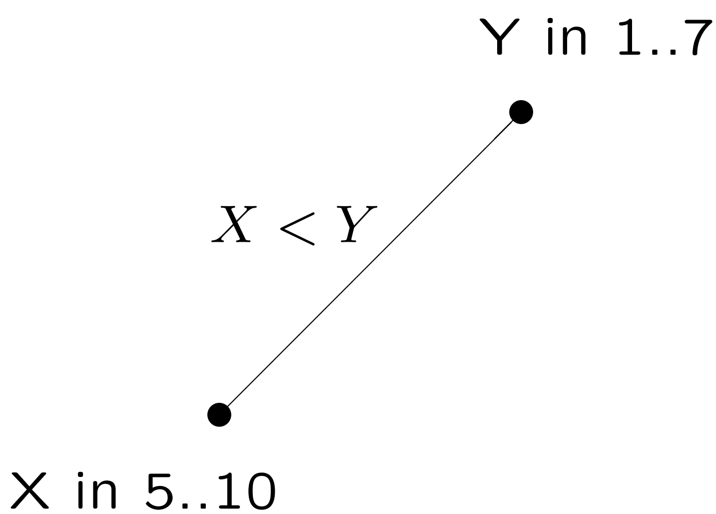
- $A \neq B$, $A \neq C$ and $B \neq C$

- $C < B$

- $(A < B < C)$ or $(C < B < A)$

Example

Two problem variables X and Y with the integer domains 5..10 and 1..7. One constraint (relation) X<Y:



New domains imposed by the constraint:

X in 5..6

Y in 6..7

Operations on constraints

- **Satisfiability:** Does a given set of constraint have at least one solution?

- **Entailment:** Is every solution of a set S of constraints also a solution of a constraint C (denoted $S \models C$)?

- **Equality:** Do two sets of constraints have the same set of solutions?

- **Optimality:** Find the best solution (given some criterion of optimality)

- **Simplification:** Given a set S of constraints, find a simpler set of constraints S' equivalent to S.

Primitive Finite Domain constraints

```
| ?- X in 3..8.
X in 3..8

| ?- X in 3..8, Y in 1..4, Z #= X+Y.
X in 3..8,
Y in 1..4,
Z in 4..12

| ?- X in 5..10, Y in 1..7, X #< Y.
X in 5..6,
Y in 6..7
```

Domains vs solutions

Note that domains are not identical to solutions:

```
?- X in 5..10, Y in 1..7, X #< Y.
```

Produces the domains:

```
X in 5..6.
Y in 6..7.
```

But the domains *contain* all solutions:

```
X = 5, Y = 6
X = 5, Y = 7
X = 6, Y = 7
```

More examples

```
| ?- X in 0..9, Y in 0..1, X #< Y.
X = 0,
Y = 1

| ?- X in 4..6, Y in 1..3, X #< Y.
no

| ?- X in 1..12, Y in 1..12, X #= 2*Y.
X in 2..12,
Y in 1..6

| ?- X in 1..2, Y in 1..2, Z in 1..2,
     X #\= Y, X #\= Z, Y #\= Z.
X in 1..2,
Y in 1..2,
Z in 1..2
```

Parallel declaration of domains

```
| ?- domain([X,Y,Z], 0, 9).
```

Labeling

Domains approximate solutions...

```
| ?- X in 1..2, Y in 1..3, X #< Y.
X in 1..2,
Y in 2..3
```

Systematically assign values to a variable from its domain.

```
| ?- X in 1..2, Y in 1..3, X #< Y,
     labeling([],[X,Y]).
X=1, Y=2
X=1, Y=3
X=2, Y=3
```

```
| ?- X in 1..12, Y in 1..12, X #= 2*Y,
     labeling([],[X,Y]).
X=2, Y=1
X=4, Y=2
...
```

CLP(X)

A *logic program* is a set of rules

$$A_0 \texttt{ :- } A_1, \ldots, A_n$$

or facts

$$A_0$$

where $A_0, A_1, \ldots, A_n$ are atomic formulas; i.e. formulas of the form $p(t_1, \ldots, t_n)$.

Note: A constraint is an atomic formula!

A constraint logic program is a logic program where some of $A_1, \ldots, A_n$ may be (some pre-defined) constraints over some algebraic structure X.

CLP(X)

- CLP(R), reals
- CLP(Q), rational numbers
- CLP(B), Boolean values
- CLP(FD), finite domains
- CLP(Sets), sets

CLP(FD)

```
queens(N, L) :-
    length(L, N),
    domain(L, 1, N),
    safe(L),
    labeling([], L).

safe([]).
safe([X|Xs]) :-
    safe_between(X, Xs, 1),
    safe(Xs).

safe_between(X, [], M).
safe_between(X, [Y|Ys], M) :-
    no_attack(X, Y, M),
    M1 is M+1,
    safe_between(X, Ys, M1).

no_attack(X, Y, N) :-
    X #\= Y, X+N #\= Y, X-N #\= Y.
```

General Strategy

```
solution(L) :-
    create_variables(L),
    constrain_variables(L),
    solve_constraints(L).
```

Optimization

```
| ?- X in 1..9, Y in 4..6, Z #= X-Y,
     labeling([maximize(Z)],[X,Y]).
```

```
items(A,B,C,S,P) :-
    domain([A,B,C],0,10),
    AS #= 2*A, AP #= 3*A,
    BS #= 3*B, BP #= 4*B,
    CS #= 7*C, CP #= 10*C,
    S  #>= AS+BS+CS,
    P  #= AP+BP+CP,
    labeling([maximize(P)],[P,S,A,B,C]).
```

Global Constraints

all_different($[X_1, \ldots, X_n]$)

```
smm([S,E,N,D,M,O,R,Y]) :-
    domain([S,E,N,D,M,O,R,Y], 0, 9),
    S #> 0, M #> 0,
    all_different([S,E,N,D,M,O,R,Y]),
    sum(S,E,N,D,M,O,R,Y),
    labeling([], [S,E,N,D,M,O,R,Y]).

sum(S, E, N, D, M, O, R, Y) :-
              1000*S+100*E+10*N+D
             +1000*M+100*O+10*R+E
   #= 10000*M+1000*O+100*N+10*E+Y.
```

```
cumulative(Ss,Ds,Rs,L)

| ?- domain([S1,S2,S3],0,4),
     S1 #< S3,
     cumulative([S1,S2,S3],[3,4,2],[2,1,3],3),
     labeling([],[S1,S2,S3]).
```

Resource allocation

```
shower(S, Done) :-
    D = [5,3,8,2,7,3,9,3,3,5,7],
    R = [1,1,1,1,1,1,1,1,1,1,1],
    length(D, N),
    length(S, N),
    domain(S, 0, 100),
    Done in 0..100,
    ready(S, D, Done),
    cumulative(S, D, R, 3),
    labeling([minimize(Done)], [Done|S]).

ready([], [], _).
ready([S|Ss], [D|Ds], Done) :-
    Done #>= S+D,
    ready(Ss, Ds, Done).
```

element(X,[$X_1, \ldots, X_n$],Y)

```
| ?- element(X, [1,2,3,5], Y).

| ?- X in 2..3, element(X, [1, X, 4, 5], Y).
```

`circuit`$([X_1, \ldots, X_n])$

Traveling Salesman

	X_1	X_2	X_3	X_4	X_5	X_6	X_7
X_1	–	4	8	10	7	14	15
X_2	4	–	7	7	10	12	5
X_3	8	7	–	4	6	8	10
X_4	10	7	4	–	2	5	8
X_5	7	10	6	2	–	6	7
X_6	14	12	8	5	6	–	5
X_7	15	5	10	8	7	5	–

Traveling Salesman (cont'd)

```
tsp(Cities, Cost) :-
    Cities = [X1,X2,X3,X4,X5,X6,X7],
    element(X1,[ 0, 4, 8,10, 7,14,15],C1),
    element(X2,[ 4, 0, 7, 7,10,12, 5],C2),
    element(X3,[ 8, 7, 0, 4, 6, 8,10],C3),
    element(X4,[10, 7, 4, 0, 2, 5, 8],C4),
    element(X5,[ 7,10, 6, 2, 0, 6, 7],C5),
    element(X6,[14,12, 8, 5, 6, 0, 5],C6),
    element(X7,[15, 5,10, 8, 7, 5, 0],C7),
    Cost #= C1+C2+C3+C4+C5+C6+C7,
    circuit(Cities),
    labeling([minimize(Cost)], Cities).
```

Deductive Databases: Overview

- Top-down evaluation;
- Relational databases;
- Bottom-up evaluation;
- "Magic templates"

Logic programs as Databases

- Powerful language for representation of relational data.
  - Explicit data
  - Views
  - Queries
  - Integrity constraints

- How to compute answers to database queries?

- Does not address issues such as concurrency control, updates, crashes etc.

Top-down $\Rightarrow$ Recomputation

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

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

Top-down ⇒ Infinite computations

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

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

Properties: Top-down

- Advantages:
  - Efficient handling of search space;
  - Goal-directed (Backward-chaining);
- Disadvantages:
  - Termination;
  - Recomputations;

How to compute database queries?

Example:

Father

X	Y
tom	mary
john	tom
$\vdots$	$\vdots$

Mother

X	Y
mary	billy
kate	tom
$\vdots$	$\vdots$

New derived relations using relational algebra:

$$\begin{aligned} P &:= F(X,Y) \cup M(X,Y) \\ GP &:= \pi_{X,Z}(P(X,Y) \bowtie P(Y,Z)) \end{aligned}$$

Bottom-up evaluation (Cf. T_P)

$$
\begin{aligned}
S_P(X) = \\
\{A_0\theta \mid\ & A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and} \\
& A'_1, \ldots, A'_n \in X \text{ and} \\
& mgu\{A_1 = A'_1, \ldots, A_n = A'_n\} = \theta\}
\end{aligned}
$$

Naive evaluation

fun *naive*(P)
begin
  x := *facts*(P);
  repeat
    y := x;
    x := $S_P(y)$;
  until $x = y$;
  return x;
end

Bottom-up evaluation (cont'd.)

$$\Delta S_P(X, \Delta X) =$$

$$\{A_0\theta \mid A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and } A'_1, \ldots, A'_n \in X, \exists A'_i \in \Delta X \text{ and } mgu\{A_1 = A'_1, \ldots, A_n = A'_n\} = \theta\}$$

Semi-naive evaluation

fun *seminaive*(P)
begin
  Δx := *facts*(P);
  $x := \Delta x$;
  repeat
    $\Delta x := \Delta S_P(x, \Delta x) \setminus x$;
    $x := x \cup \Delta x$;
  until $\Delta x = \emptyset$;
  return x;
end

Properties: Bottom-up

- Advantages:
  - Termination;
  - Re-use of already computed results;

- Disadvantages:
  - Not goal-directed;
  - Termination;

Magic Templates

Let $magic(P)$ be the least program such that if $A_0 \leftarrow A_1, \ldots, A_n \in P$ then:

- $A_0 \leftarrow call(A_0), A_1, \ldots, A_n \in magic(P)$

- $call(A_i) \leftarrow call(A_0), A_1, \ldots, A_{i-1} \in magic(P)$

In addition $call(A) \in magic(P)$ if $\leftarrow A$.

Compute *naive*(*magic*(P)).

Example

```
%-----------ORIGINAL PROGRAM--------------
p(X,Y) :- e(X,Y).
p(X,Z) :- p(X,Y), e(Y,Z).

e(a,b).
e(b,a).
e(b,c).

:- p(a,X).
%--------------MAGIC PROGRAM--------------
p(X,Y) :- call(p(X,Y)), e(X,Y).
p(X,Z) :- call(p(X,Z)), p(X,Y), e(Y,Z).
e(a,b) :- call(e(a,b)).
e(b,a) :- call(e(b,a)).
e(b,c) :- call(e(b,c)).
%
call(e(X,Y)) :- call(p(X,Y)).
call(p(X,Y)) :- call(p(X,Z)).
call(e(Y,Z)) :- call(p(X,Z)), p(X,Y).
%
call(p(a,X)).
```

Bottom-up with Magic Templates

- Advantages:
  - Termination;
  - Re-use of results;
  - Goal-directed;
- Disadvantages:
  - Sometimes slower than Prolog (when Prolog terminates);

Logic programming with Equations

- What is equality?
- E-unification.
- Logic programs with Equations
- SLDE-resolution

What is equality?

We sometimes want to express that two terms should be interpreted as the same object.

Example

Let Γ be:

$$\begin{array}{l} \textit{person}(X) \leftarrow \textit{female}(X). \\ \textit{female}(\textit{queen}). \\ \textit{silvia} \doteq \textit{queen}. \end{array}$$

Then $\Gamma \models \textit{person}(\textit{silvia})$.

Equations

An equation is an atom $s \doteq t$ where s and t are terms.

The predicate $\doteq$ is *always* interpreted as the identity relation.

That is, $\Im \models_\sigma s \doteq t$ iff $\sigma_\Im(s) = \sigma_\Im(t)$.

Example

$$\begin{array}{rcl} X + 0 & \doteq & X. \\ X + s(Y) & \doteq & s(X + Y). \\ 1 & \doteq & s(0). \\ 2 & \doteq & 1 + 1. \\ 3 & \doteq & 2 + 1. \\ & \vdots & \end{array}$$

Equality theory

$E \vdash s \doteq t$: "$s \doteq t$ is derived from E"

$$\{\ldots, s \doteq t, \ldots\} \vdash s \doteq t$$

$$E \vdash s \doteq s$$

$$\frac{E \vdash s \doteq t}{E \vdash s\sigma \doteq t\sigma}$$

$$\frac{E \vdash s \doteq t}{E \vdash t \doteq s}$$

$$\frac{E \vdash r \doteq s \qquad E \vdash s \doteq t}{E \vdash r \doteq t}$$

$$\frac{E \vdash s_1 \doteq t_1 \quad \cdots \quad E \vdash s_n \doteq t_n}{E \vdash f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)}$$

$***$

$s \equiv_E t$ iff $E \vdash s \doteq t$

Theorem

The relation $\equiv_E$ is an equality relation.

Theorem

$E \models s \doteq t$ iff $s \equiv_E t$ (iff $E \vdash s \doteq t$) .

E-unification

Two terms s and t are E-unifiable iff $s\theta \equiv_E t\theta$. The substitution θ is called an E-unifier.

Problem

- E-unification is undecidable;

- In general there is no single "most general unifier" but only "complete sets of E-unifiers";

- This set may be infinite.

Unification...

...can be carried out using e.g. *narrowing*.

Logic programs with Equations

Programs consist of two components

- A set of definite clauses that do not include the predicate symbol $\doteq/2$;

- A set of equations;

Observation

Herbrand interpretations are uninteresting!

Patch

Consider interpretations whose domain consists of sets (equivalence classes) of ground terms.

Every equivalence class consists of "equivalent term".

Interpretations with domain $U_P/\equiv_E$ are of special interest.

Let $\Im$ be an interpretation where $|\Im| = U_P/_{\equiv_E}$:

That is, $\overline{s} = \{t \in U_P \mid E \vdash s \doteq t\}$.

Theorem

$$\begin{array}{lll} \Im \models s \doteq t & \text{iff} & \overline{s} = \overline{t} \\ & \text{iff} & s \equiv_E t \\ & \text{iff} & E \models s \doteq t \end{array}$$

NB: Herbrand interpretations as a special case!

The Least Model

Every program P, E has a least model $M_{P,E}$:

$$P, E \models p(t_1, \ldots, t_n) \text{ iff } \overline{p(t_1, \ldots, t_n)} \in M_{P,E}$$

Fixed point semantics

$$\begin{aligned} T_{P,E}(x) := \{\overline{A} \quad | \quad & A \leftarrow B_1, \ldots, B_n \in ground(P) \\ & \wedge \overline{B_1}, \ldots, \overline{B_n} \in x\} \end{aligned}$$

SLDE-Resolution

Given a goal

$$\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$$

with selected literal A_i. If

- $H \leftarrow B_1, \ldots, B_m$ is a renamed program clause
- H and A_i have a non-empty *set* Θ of E-unifiers
- $\theta \in \Theta$

then

$$\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta$$

is a new goal.

Theorem [Soundness]

If $\leftarrow A_1, \ldots, A_n$ has a computed answer substitution θ then $P, E \models \forall(A_1 \wedge \cdots \wedge A_n)\theta$.

Theorem [Completeness]

Similar to SLD-resolution.