1. Introduction

Freitag, 10. April 2015 08:30

Logic Programming

Learning several prog. languages:

- express ideas during SW development
- needed to decide which language to use in project
- eases learning of new languages
- needed to design new languages

Imper. + Fct. Languages:
programs compute functions

Logic Languages:

- programs describe relations
- execution: ask queries, program tries to prove queries
- main application area: AI, expert systems, deductive data bases, ...

Prolog-Implementation: SWI Prolog (see web page)

Example: Family Tree

— : married

↙ : children

Facts and Queries

Knowledge has to be translated to Prolog-Syntax.

Prolog = <u>Pro</u>gramming in <u>Log</u>ic

Prolog program consists of (special) logic formulas, so-called clauses:

- facts
- rules (allow to deduce new knowledge from existing knowledge)

Syntax of facts:

predicate(obj1, ..., objn). ← Boolean statements

symbol

↑

Strings starting with lower-case letter

Relations are not symmetric.

Syntax for comments:

% ... end of line or

/*
⋮
*/

Execution: ask queries

?- statement.

Closed World Assumption:
everything that can't be deduced from prog. clauses is false

Variables in Programs

Variables: strings starting with capital letter or with _

Variables in programs are universally quantified

Ex: all X are human
(i.e., everything is human)

Same variables in one clause have to be instantiated in the same way:

likes(X,X). - everybody likes himself

likes(X,Y). - everybody likes everybody

Variables in Queries

Variables in queries are existentially quantified — can be used to let the program compute solutions.

Ex: Who is the mother of susanne? (Is there an X such that ... ?)

Prolog returns a suitable answer substitution.

If there are several solutions: ; makes Prolog continue searching for answers.

Prolog searches through its prog. clauses from top to bottom.

Same program can be used to compute mothers or children ⟹

Prolog programs have no fixed input/output, but input/output depends on query.

?- motherOf(X,Y).
X = monika, Y = Karin ;
⋮

?- human(Z). ← Prolog returns the most general instantiations that make the query true.
true

Combined Queries

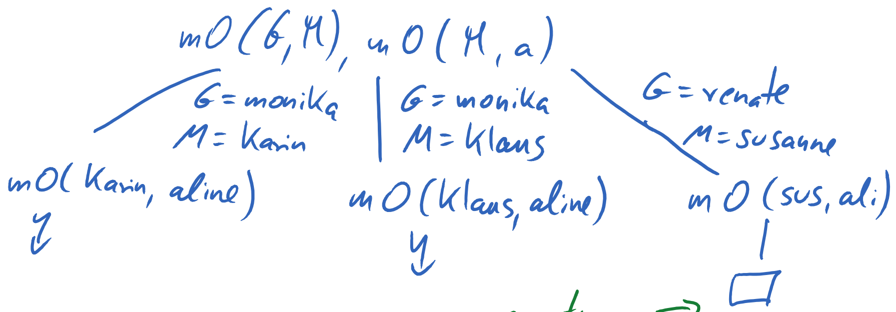
Combined Queries

, $\hat{=}$ and ; $\hat{=}$ or

Ex: Is gerd the father of susanne?

Combined queries are executed from **left to right**

- first solve query married(gerd, W)
  $\Rightarrow$ finds an instantiation of W
- then solve second query motherOf(W, susanne) for this instantiation of W
- If second query fails, then backtrack to the first query and try the next solution.
- Prolog computes a proof tree (so-called SLD tree)



empty clause → □ means that this path of the proof tree was successful

Rules

rules allow to deduce new knowledge from existing knowledge

Ex: F is the father of C if (:-) there exists a W such that

F is married to W and
W is the mother of C.

Rules: head :- $\underbrace{\text{statement}_1, \dots, \text{statement}_n}_{\text{body of the rule}}$.

means: in order to prove head,
one can instead prove the statements
in the body

Ex: fO(gerd, Y)
| F = gerd, C = Y
married(gerd, W), motherOf(W, Y)
| W = renate
motherOf(renate, Y)
/ Y = susanne   \ Y = peter
□   □

Several rules for the same predicate

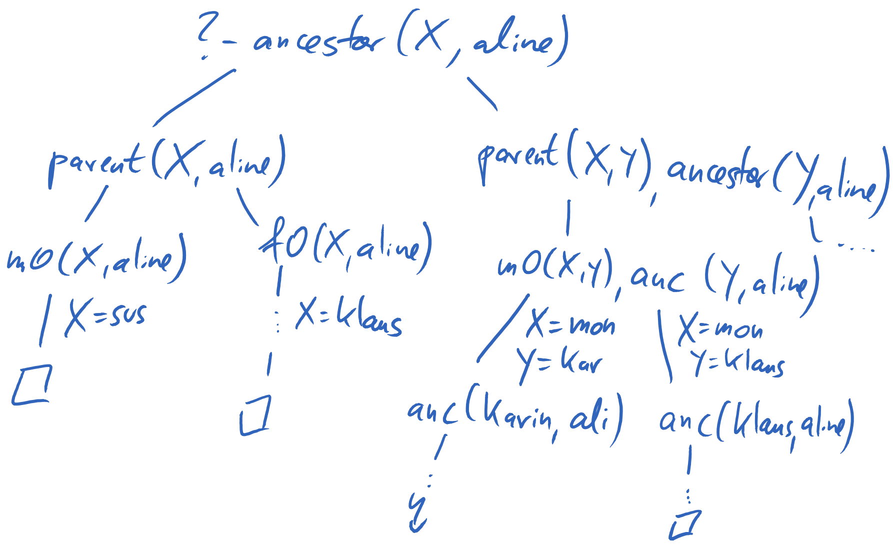
alternative:
parent(X, Y) :- motherOf(X, Y) ; fatherOf(X, Y).

(; is defined by 2 clauses)

?- parent(X, susanne).
Mother will be found first due to the order
of prog. clauses.

Recursive Rules

Ex: ancestor predicate
2nd rule is recursive



Characteristics of Logic Programming:

- no control structures, just facts + rules
- prog. execution ≙ automated theorem proving
- particularly suitable for AI

Plan for the lecture:

Ch. 1: Introduction to LP
Ch 2: Predicate Logic
Ch 3: Resolution (Proof Technique used in LP)
Ch 4: Syntax + Semantics of LP
Ch 5: Prog. Language Prolog

Organisation

- english
- german course notes (web)
- english notes from the lecture + slides (web)
- lecture: 8:30 – 10:00 mon + fri
  exercise: 10:15 – 11:45 fri
- video recording from 2013

- V3+Ü2 lecture, 2 variants (for Bachelor + Master Students)
  called V3B + V3M (Math students: V3B)
- Web site: http:/ verify.rwth-aachen.de / lp15
- Exercises:
  - weekly exercise sheet
  - groups of 2
  - registering for exercises: via our web site
    (until Friday next week)
  - 50% of exercise points needed to participate in the exam
  - exam: August 19 + September 14
- Vorgezogene Masterprüfung: register via ZPA
  (June 8 - 18)

2.1 Syntax of Predicate Logic

Freitag, 10. April 2015 10:00

2.1: Syntax
2.2: Semantics } of Predicate Logic

2.1. Syntax of Predicate Logic

Syntax: determines which symbols constitute the words of a language and in which order these symbols may occur

First: define alphabet for formulas of predicate logic

Def 2.1.1 (Signature)

A signature (Σ, Δ) is a pair with $\Sigma = \bigcup_{n \in \mathbb{N}} \Sigma_n$ and $\Delta = \bigcup_{n \in \mathbb{N}} \Delta_n$. The sets Σ_n and Δ_n are pairwise disjoint. Every $f \in \Sigma_n$ is a function symbol of arity n, every $p \in \Delta_n$ is a predicate symbol of arity n.
The elements of Σ_0 are also called constants. We always require $\Sigma_0 \neq \emptyset$.

Ex. 2.1.2: Signature (Σ, Δ) for the logic prog. from Chapter 1. Here $\Sigma = \Sigma_0 \cup \Sigma_3$, $\Delta = \Delta_1 \cup \Delta_2$.
date is an additional fct. symbol of arity 3
(for dates consisting of day, month, year)

Fct symbols create objects (terms)

Pred symbols create statements (formulas)

Def 2.13 (Terms)

Let (Σ, Δ) be a signature, let $\mathcal{V}$ be a set of variables with $\Sigma \cap \mathcal{V} = \emptyset$. Then $\mathcal{T}(\Sigma, \mathcal{V})$ is the set of all terms over Σ and $\mathcal{V}$. $\mathcal{T}(\Sigma, \mathcal{V})$ is the smallest set such that:
- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ if $f \in \Sigma_n$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ for some $n \in \mathbb{N}$.

$\mathcal{T}(\Sigma)$ stands for $\mathcal{T}(\Sigma, \emptyset)$, i.e., the set of ground terms (terms without variables)

For any term t, let $\mathcal{V}(t)$ be the set of all variables in t.

Ex 2.14 Let Σ be as in Ex 2.1.2, let $\mathcal{V} = \{X, Y, Z, \ldots\}$.

Terms in $\mathcal{T}(\Sigma, \mathcal{V})$: X, monika, 42, date$(10, 4, 2015)$, date$(X$, monika, date$(10, 4, 2015))$, ...

Def 2.15 (Formulas)

Let (Σ, Δ) be a signature and $\mathcal{V}$ be a set of variables. The set of atomic formulas over (Σ, Δ) and $\mathcal{V}$ is defined as

$$At(\Sigma, \Delta, \mathcal{V}) = \{p(t_1, \ldots, t_n) \mid p \in \Delta_n, t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}.$$

$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ is the set of all formulas over (Σ, Δ) and $\mathcal{V}$. $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ is the smallest set such that
- $At(\Sigma, \Delta, \mathcal{V}) \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- if $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ then $\neg\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

"not φ"

"or"

"implies"

- if $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, then $\neg \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

"or" "plies"

- if $\varphi_1, \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$,

"and" $\quad (\varphi_1 \leftrightarrow \varphi_2) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

"is equivalent to"

- if $X \in \mathcal{V}$ and $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, then $(\forall X \ \varphi)$, $(\exists X \ \varphi) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

"for all" "exists"

For a formula φ, $\mathcal{V}(\varphi)$ is the set of variables occurring in φ.

A variable X occurs <u>free</u> in a formula φ iff — "if and only if"

- φ is an atomic formula and $X \in \mathcal{V}(\varphi)$ or
- $\varphi = \neg \varphi_1$ and X occurs free in φ_1 or
- $\varphi = (\varphi_1 \circ \varphi_2)$ with $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and X occurs free in φ_1 or in φ_2 or
- $\varphi = (Q Y \ \varphi_1)$ with $Q \in \{\forall, \exists\}$, X occurs free in φ_1, and $X \neq Y$.

A formula is <u>closed</u> iff it does not contain free variables.

A formula is <u>quantifier-free</u> iff it does not contain $\forall$ or $\exists$.

We usually omit $(\ldots)$ whenever possible.

<u>Ex 2.16</u> We use the signature of Ex. 2.12.

$female(monika) \in At(\Sigma, \Delta, \mathcal{V})$

$motherOf(X, susanne) \in At(\Sigma, \Delta, \mathcal{V})$

$born(monika, date(15, 10, 1966)) \in At(\Sigma, \Delta, \mathcal{V})$

$\forall W\,(married(gerd, W) \wedge motherOf(W, C)) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

gerd is married with all women W and they all are the mother of C

only free variable: C

$married(gerd, W) \wedge \neg(\forall W\, motherOf(W, C)) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

free variables: W, C

We abbreviate $\forall X_1 (\ldots (\forall X_n \ \varphi)\ldots)$ by $\forall X_1, \ldots, X_n \ \varphi$

$\exists X_1 (\ldots (\exists X_n \ \varphi)\ldots)$ by $\exists X_1, \ldots, X_n \ \varphi$

To distinguish variables from fct. and pred. symbols:

Variables start with upper-case letters

fct. + pred. symbols start with lower-case letters

Ex 2.1.7 Every logic program stands for a set of formulas. Here, the variables are universally quantified (i.e., with $\forall$).

Variables in terms and formulas stand for arbitrary objects

$\Rightarrow$ they can be substituted by objects (i.e., by terms).

Def 2.1.8 (Substitution)

A mapping $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ is a substitution iff $\sigma(X) \neq X$ holds for finitely many $X \in \mathcal{V}$. $DOM(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ is the domain of σ and

$RANGE(\sigma) = \{\sigma(X) \mid X \in DOM(\sigma)\}$ is the range of σ.

A substitution can be denoted as $\{X / \sigma(X) \mid X \in DOM(\sigma)\}$.

A subst. σ is a ground substitution :⇔ $\sigma(X)$ ...

$\ldots \} = \{X / \sigma(X) \mid X \in DOM(\sigma)\}$.

A subst. σ is a <u>ground substitution</u> iff $\sigma(X)$ contains no variables for all $X \in DOM(\sigma)$.

A substitution σ is a <u>variable renaming</u> iff it is injective and $\sigma(X) \in \mathcal{V}$ for all $X \in \mathcal{V}$.

Ex: $\sigma = \{X/Y, Y/Z, Z/X\}$.

$\sigma(X) = Y$
$\sigma(Y) = Z$
$\sigma(Z) = X$
$\sigma(U) = U$

Substitutions can be extended to terms, i.e., $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$:

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

Ex:

$\sigma(\text{date}(X, \text{monika}, Y)) =$
$\text{date}(\sigma(X), \text{monika}, \sigma(Y)) =$
$\text{date}(Y, \text{monika}, Z)$

Substitutions can also be extended to formulas:

- $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$
- $\sigma(\neg \varphi_1) = \neg\, \sigma(\varphi_1)$
- $\sigma(\varphi_1 \circ \varphi_2) = \sigma(\varphi_1) \circ \sigma(\varphi_2)$ for $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$
- $\sigma(QX\ \varphi_1) = QX\ \sigma(\varphi_1)$, if $X \notin DOM(\sigma) \cup \underbrace{\mathcal{V}(RANGE(\sigma))}_{\text{Variables occurring in the RANGE of } \sigma}$

  for $Q \in \{\forall, \exists\}$

Reason: $\forall X \; \text{human}(X)$
and $\forall Y \; \text{human}(Y)$
should be treated in the same way.
$\Rightarrow$ if σ modifies X or if the application of σ

should be treated in the same way.
$\Rightarrow$ if σ modifies X or if the application of σ introduces X, then first rename the bound var. X to a fresh variable

- $\sigma(QX\ \varphi_1) = Q\,X'\ \sigma(\delta(\varphi_1))$ for $Q \in \{\forall, \exists\}$, $X \in DOM(\sigma) \cup \mathcal{V}(RANGE(\sigma))$.
  Here, X' is a fresh variable with
  $X' \notin DOM(\sigma) \cup \mathcal{V}(RANGE(\sigma)) \cup \mathcal{V}(\varphi_1)$ and
  $\delta = \{X/X'\}$.

<u>Ex. 2.19</u> $\sigma = \{X/date(X,Y,Z),\ Y/monika,\ Z/date(Z,Z,Z)\}$

$\sigma(date(X,Y,Z)) = date(date(X,Y,Z),\ monika,\ date(Z,Z,Z))$

$\sigma(\forall Y\ married(X,Y)) =$

$\sigma(\forall Y'\ married(X,Y')) =$

$\forall Y'\ married(date(X,Y,Z),\ Y')$

Problem 1: $\sigma(X)$ contains Y

Problem 2: $Y \in DOM(\sigma)$

An instance $\sigma(t)$ or $\sigma(\varphi)$ of a term t (resp. a $\underbrace{\text{formula}}$ quantifier-free
φ) is a <u>ground instance</u> iff it doesn't contain variables.

2.2 Semantics of Predicate Logic

Montag, 13. April 2015 08:30

- Ex. Sheet 1 on the web (due on Apr. 20)
- FR, April 17: 2 lectures (lecture instead of exercise course)
- MO, April 20: ex. course instead of lecture
- register for exercises on our web site (until Fri, April 17)

2.2 Semantics of Pred. Logic

Goal: Describe the meaning of formulas and terms

Use interpretations I:

function

assigns a meaning α_f to every fct. symbol f

and — " — α_p — " — pred. symbol p

relation

Def 2.2.1 (Interpretation, Structure, Satisfiability, Model)

Let (Σ, Δ) be a signature. An interpretation for (Σ, Δ) is a triple $I = (\mathcal{A}, \alpha, \beta)$. $\mathcal{A}$ is the carrier of the interpr. where $\mathcal{A}$ is a set with $\mathcal{A} \neq \emptyset$. The mapping α maps every $f \in \Sigma_n$ to a function $\alpha_f : \mathcal{A}^n \to \mathcal{A}$ and every $p \in \Delta_n$ with $n > 0$ to a set $\alpha_p \subseteq \mathcal{A}^n$. For $p \in \Delta_0$, we have $\alpha_p \in \{TRUE, FALSE\}$. Here, α_f and α_p are the meaning of f and p, resp. The mapping $\beta : \mathcal{V} \to \mathcal{A}$ is called a variable assignment.

For every interpretation I, we get a function

$I : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{A}$:

$I(X) = \beta(X)$ for all $X \in \mathcal{V}$

$I(f(t_1, \ldots, t_n)) = \alpha_f(I(t_1), \ldots, I(t_n))$

Ex 2.2.2 Consider the following interpretation:

$I = (\mathcal{A}, \alpha, \beta)$ with

$\mathcal{A} = \mathbb{N}$

$\alpha_n = n$ for all $n \in \mathbb{N}$

$\alpha_{monika} = 0, \alpha_{karin} = 1, \alpha_{renate} = 2, \ldots$

$\alpha_{date}(n_1, n_2, n_3) = n_1 + n_2 + n_3$ for all $n_1, n_2, n_3 \in \mathbb{N}$

$\alpha_{female} = \{ n \mid n \text{ is even} \}, \alpha_{male} = \{ n \mid n \text{ is odd} \},$

$\alpha_{human} = \mathbb{N}, \alpha_{married} = \{ (n, m) \mid n > m \}, \ldots$

$\beta(X) = 0, \beta(Y) = 1, \beta(Z) = 2, \ldots$

Meaning of the term $date(1, X, karin)$ under this interpr.:

$I(date(1, X, karin)) = \alpha_{date}(\alpha_1, \beta(X), \alpha_{karin})$

$= 1 + 0 + 1 = 2$

Def 2.2.1 (contd.)

For $X \in \mathcal{V}$ and $a \in \mathcal{A}$, let $\beta[\![X/a]\!]$ be the var. assignment with $\beta[\![X/a]\!](X) = a$

$\beta[\![X/a]\!](Y) = \beta(Y)$ for all $Y \neq X$

Similarly, for $I = (\mathcal{A}, \alpha, \beta)$, let

$I[\![X/a]\!] = (\mathcal{A}, \alpha, \beta[\![X/a]\!])$.

An interpretation $I = (\mathcal{A}, \alpha, \beta)$ satisfies a formula $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ (denoted $I \models \varphi$) iff

- $\varphi = p(t_1, \ldots, t_n)$ with $p \in \Delta_n$ with $n > 0$ and $(I(t_1), \ldots, I(t_n)) \in \alpha_p$ or
- $\varphi = p$ with $p \in \Delta_0$ and $\alpha_p = TRUE$ or
- $\varphi = \neg \varphi_1$ and $I \not\models \varphi_1$ or

- $\varphi = \varphi_1 \wedge \varphi_2$ and $I \models \varphi_1$ and $I \models \varphi_2$ or
- $\varphi = \varphi_1 \vee \varphi_2$ and ($I \models \varphi_1$ or $I \models \varphi_2$) or
- $\varphi = \varphi_1 \rightarrow \varphi_2$ and if $I \models \varphi_1$, then $I \models \varphi_2$ or
- $\varphi = \varphi_1 \leftrightarrow \varphi_2$ and ($I \models \varphi_1$ iff $I \models \varphi_2$) or
- $\varphi = \forall X\ \varphi_1$ and $I[\![X/a]\!] \models \varphi_1$ for all $a \in \mathcal{A}$ or
- $\varphi = \exists\ \varphi_1$ and $I[\![X/a]\!] \models \varphi_1$ for some $a \in \mathcal{A}$

Ex 222 (contd.)

$I \models \text{marriedOf}(\text{date}(1, X, \text{Karin}), \text{Karin})$

iff $\big(\underbrace{I(\text{date}(1,X,\text{Karin}))}_{2}, \underbrace{I(\text{Karin})}_{\alpha_{\text{Karin}}=1}\big) \in \underbrace{\alpha_{\text{marriedOf}}}_{\{(n,m) \mid n>m\}}$

$\Rightarrow I$ satisfies this formula, since $2 > 1$.

$I \models \forall X\ \text{female}(\text{date}(X, X, \text{monika}))$.

iff $I[\![X/a]\!] \models \text{female}(\text{date}(X,X,\text{monika}))$ for all $a \in \underbrace{\mathcal{A}}_{\mathbb{N}}$

iff $I[\![X/a]\!](\text{date}(X,X,\text{monika})) \in \alpha_{\text{female}}$ for all $a \in N$

iff $\alpha_{\text{date}}(a, a, \underbrace{\alpha_{\text{monika}}}_{0}) \in \underbrace{\alpha_{\text{female}}}_{\{n \mid n \text{ is even}\}}$ for all $a \in \mathbb{N}$

iff $a + a + 0 \in \{n \mid n \text{ is even}\}$ for all $a \in \mathbb{N}$

true!

Def 221 (contd.)

An interpret. I is a <u>model</u> of φ iff $I \models \varphi$. I is a model of $\Phi \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

iff $I \models \varphi$ for all $\varphi \in \Phi$. (We write $I \models \Phi$)
Two formulas φ_1, φ_2 are equivalent iff $I \models \varphi_1$ iff $I \models \varphi_2$ for all interpretations I

A formula (or set of formulas) is satisfiable iff it has a model. It is called valid iff it is satisfied by every interpretation.

An interpretation $S = (\mathcal{A}, \alpha)$ without var. assignment is called structure. For closed formulas, it suffices to regard structures:

$S \models \varphi$ iff $I \models \varphi$ for some interpretation $I = (\mathcal{A}, \alpha, \beta)$

Similarly, we can define $S(t)$ for ground terms t.

The formulas in Ex. 2.22 were satisfiable, but not valid (there exist interpretations that don't satisfy them).
Example for a valid formula:

$\varphi \vee \neg\varphi$ for any $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

Ex. for unsatisfiable formula:

$\varphi \wedge \neg\varphi$

Lemma 2.2.3 clarifies the connections between:

substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ – syntax

Variable assignment $\beta : \mathcal{V} \to \mathcal{A}$ – Semantics

Lemma 223 (Substitution Lemma)
Let $I = (\mathcal{A}, \alpha, \beta)$ be an interpretation, let $\sigma = \{X_1/t_1, \dots, X_n/t_n\}$ be a substitution.

(a) $I(\sigma(t)) = I[\![X_1/I(t_1), \dots, X_n/I(t_n)]\!](t)$

(b) $I \models \sigma(\varphi)$ iff

$I[\![X_1/I(t_1), \dots, X_n/I(t_n)]\!] \models \varphi$.

Ex 224 Let I be the interpr. from Ex. 222.

Let $\sigma = \{X / date(1, X, karin)\}$, let $t = date(X, Y, Z)$.

$I(\sigma(t)) = I(date(date(1, X, karin), Y, Z))$

$$= \alpha_{date}(\alpha_{date}(\underbrace{\alpha_1}_{1}, \underbrace{\beta(X)}_{0}, \underbrace{\alpha_{karin}}_{1}), \underbrace{\beta(Y)}_{1}, \underbrace{\beta(Z)}_{2}))$$

$= 1 + 0 + 1 + 1 + 2$

$= 5$

$$I[\![X / \underbrace{I(date(1, X, karin))}_{\alpha_{date}(\alpha_1, \beta(X), \alpha_{karin}) = 2}]\!](date(X, Y, Z)) =$$

$I[\![X/2]\!](date(X, Y, Z)) =$

$$\alpha_{date}(\underbrace{\beta[\![X/2]\!](X)}_{2}, \underbrace{\beta[\![X/2]\!](Y)}_{1}, \underbrace{\beta[\![\dots]\!](Z)}_{2})$$

$= 5$

Proof of the substitution lemma 223:

(a) To prove: for all terms t we have

$I(\sigma(t)) = I[\![X_1/I(t_1), \dots, X_n/I(t_n)]\!](t)$

To prove such statements on inductively defined data structures (like terms): structural induction

(As ind. hypothesis, one may assume that the statement already holds for direct subterms.)

Ind. Base: $t \in \mathcal{V}$

Case 1: $t \in \{X_1, \dots, X_n\}$, e.g. $t = X_i$

$I(\sigma(X_i)) = I(t_i)$

$I[\![X_1/I(t_1), \dots, X_n/I(t_n)]\!](X_i) = I(t_i)$ ✓

Case 2 $t \in \mathcal{V} \setminus \{X_1, \dots, X_n\}$, e.g. $t = Y$

$\mathcal{I}(\sigma(Y)) = \mathcal{I}(Y)$

$\mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](Y) = \mathcal{I}(Y)$ ✓

Ind. Step: $t = f(s_1, \dots, s_k)$ (where $k = 0$ is possible)

$\mathcal{I}(\sigma(f(s_1, \dots, s_k))) =$

$\mathcal{I}(f(\sigma(s_1), \dots, \sigma(s_k))) =$

$\alpha_f(\underline{\mathcal{I}(\sigma(s_1))}, \dots, \underline{\mathcal{I}(\sigma(s_k))})$

$\mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](f(s_1, \dots, s_k)) =$

$\alpha_f(\underline{\mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](s_1)}, \dots, \underline{\mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](s_k)})$

Ind. Hypothesis:
$\mathcal{I}(\sigma(s_i)) = \mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](s_i)$ for all $i \in \{1, \dots, k\}$ ✓

(b) analogous to (a)

$\boxtimes$

Def 225 (Entailment)

A set of formulas Φ entails the formula φ (denoted $\Phi \models \varphi$) iff

$\mathcal{I} \models \Phi$ implies $\mathcal{I} \models \varphi$ for all interpretations $\mathcal{I}$.

Instead of "$\emptyset \models \varphi$" we also write "$\models \varphi$".

means: φ is valid

Ex 226 Entailment is checked when executing logic programs: $\Phi \models \varphi$

program clauses query

If Φ are the clauses for the example program, then ?- male(gerd).
means that we want to check

$$\Phi \models male(gerd)$$ holds ✓

The query ?- human(gerd)

means $\Phi \models human(gerd)$.

This holds, because:

$$\mathcal{I} \models \Phi$$

$\leadsto$ $\mathcal{I} \models \forall X \; human(X)$

$\leadsto$ $\mathcal{I}[\![X/a]\!] \models human(X)$ for all $a \in \mathcal{A}$

$\leadsto$ $\mathcal{I}[\![X/\mathcal{I}(gerd)]\!] \models human(X)$

$\leadsto$ $\mathcal{I} \models human(gerd)$ by the subst. lemma.

3.1 Skolem Normal Form

Freitag, 17. April 2015 08:30

Today: 2 lectures

Monday: exercise course instead of lecture

exercise sheets:

- first sheet due on Monday
- second sheet: on the web, due next Friday
- groups of two or three
- students looking for exercise partners: meet in between the 2 lectures in AH 1

$I \models \varphi$ I satisfies formula φ

$\Phi \models \varphi$ Φ entails φ

program clauses (Φ), query (φ)

means: for every interpretation I:

$I \models \Phi$ implies $I \models \varphi$

Ex: Example LP

Query: ?- motherOf(X, susanne).

This means that we have to check:

$\Phi \models \exists X \ \text{motherOf}(X, \text{susanne})$

prog. clauses (Φ)

This indeed holds:

$I \models \Phi$

$\curvearrowright I \models \text{motherOf}(\text{renate}, \text{susanne})$

substitution

$\leadsto I \models motherOf(X, susanne)\,[X/renate]$

$\leadsto I\,[\![X/I(renate)]\!] \models motherOf(X, susanne)$

by the subst. lemma 2.2.3 (a)

$\leadsto I \models \exists X\; motherOf(X, susanne)$

How does Prolog perform proofs of the form

$\Phi \models \varphi$?

3. Resolution

Problem: Entailment is defined semantically

Not suitable for automation (one would have to check <u>all possible</u> interpretations).

Solution: Check entailment syntactically

Define a <u>calculus</u> with syntactic rules that define when a formula φ can be <u>deduced</u> from Φ.

entailment (semantic) | deduction (syntactic)

Calculus is <u>sound</u> iff deduction $\Rightarrow$ entailment

(i.e. if φ is deduced from Φ, then $\Phi \models \varphi$)

Calculus is complete iff entailment $\Rightarrow$ deduction

Calculus is <u>complete</u> iff entailment $\Rightarrow$ deduction
(i.e., if $\Phi \models \varphi$, then φ can be deduced from Φ).

Unfortunately, entailment in predicate logic is <u>undecidable</u>: There is no program which always terminates and which finds out for any Φ, φ whether $\Phi \models \varphi$.
$\Rightarrow$ there is no automatable, always terminating calculus that is sound + complete.

But: Entailment is <u>semi-decidable</u>
$\Rightarrow$ there is a program such that for every Φ, φ:

prog. terminates with "Yes" iff $\Phi \models \varphi$
(But if $\Phi \not\models \varphi$, then the prog. might not terminate).

We will now introduce such a sound + complete calculus which terminates if $\Phi \models \varphi$, but which might not terminate if $\Phi \not\models \varphi$.

Resolution Calculus: sound, complete, automatable, ~~terminating~~

<u>Plan</u>

- First introduce a simpler calculus

(also sound, complete, automatable).

- Then refine this calculus step by step to increase efficiency.

Idea of the resolution calculus:
express entailment problems
as unsatisfiability problems.

Lemma 3.0.1. (Entailment vs. Unsatisfiability)
Let $\varphi_1, \dots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$.
Then $\{\varphi_1, \dots, \varphi_k\} \models \varphi$ iff
$\varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg\varphi$ is unsatisfiable.

Proof: $\{\varphi_1, \dots, \varphi_k\} \models \varphi$
$\curvearrowright$ for all int. I: $I \models \{\varphi_1, \dots, \varphi_k\}$ implies $I \models \varphi$
$\curvearrowright$ there is no I with
$I \models \{\varphi_1, \dots, \varphi_k\}$ and $I \models \neg\varphi$
$\curvearrowright$ $\varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg\varphi$ is unsatisfiable. $\boxed{\text{q}}$

Goal: Check unsatisfiability of formulas automatically.

Since this is undecidable, but semi-decidable:
Develop a technique which always finds out unsatisfiability, but may not terminate for satisfiable formulas.

3.1. Skolem Normal Form

First step to check whether a formula φ is unsatisfiable: transform φ into a normal form:

1. prenex normal form

$$\forall X_1 \exists X_2 \exists X_3 \forall X_4 \quad \psi$$

quantifier-free

2. Skolem normal form

$$\forall X_1 \forall X_2 \ldots \forall X_n \quad \psi$$

no variables except $X_1, \ldots, X_n$

Def 3.1.1. (Prenex NF)
A formula φ is in prenex normal form iff it has the form $Q_1 X_1 \ldots Q_n X_n \psi$ where $Q_1, \ldots, Q_n \in \{\forall, \exists\}$ and ψ is quantifier-free.

Thm 3.1.2. (Transformation to prenex NF)
For every formula φ, one can automatically generate an equivalent formula φ' in prenex normal form.

Proof: An algorithm for this transformation works as follows:

First replace sub-formulas $\varphi_1 \leftrightarrow \varphi_2$ by $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

Then replace sub-formulas $\varphi_1 \rightarrow \varphi_2$ by $\neg \varphi_1 \vee \varphi_2$.

Then use the following alg. PRENEX(φ):

- if φ is quantifier-free then return φ [illegible]

Then use the following alg. PRENEX(φ):

- if φ is quantifier-free, then return φ
- if $\varphi = \neg\varphi_1$, then compute
  $\text{PRENEX}(\varphi_1) = Q_1X_1 \dots Q_nX_n\ \psi_1$.
  Return $\overline{Q_1}X_1 \dots \overline{Q_n}X_n\ \neg\psi_1$,
  where $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.

$\neg\forall X\, p(X) \Rightarrow \exists X\, \neg p(X)$

- if $\varphi = \varphi_1 \circ \varphi_2$ where $\circ \in \{\wedge, \vee\}$, then compute
  $\text{PRENEX}(\varphi_1) = Q_1X_1 \dots Q_nX_n\ \psi_1$
  $\text{PRENEX}(\varphi_2) = R_1Y_1 \dots R_mY_m\ \psi_2$
  By renaming bound variables, we can ensure that $X_1, \dots, X_n$ do not occur in $R_1Y_1 \dots R_mY_m\ \psi_2$ and $Y_1, \dots, Y_m$ do not occur in $Q_1X_1 \dots Q_nX_n\ \psi_1$.
  Then return:
  $QX_1 \dots Q_nX_n R_1Y_1 \dots R_mY_m\ (\psi_1 \circ \psi_2)$
- if $\varphi = QX\ \varphi_1$ with $Q \in \{\forall, \exists\}$,
  then compute $\text{PRENEX}(\varphi_1) = Q_1X_1 \dots Q_nX_n\ \psi$.
  By renaming bound variables, we ensure that $X_1, \dots, X_n$ are different from X.
  Then return $QX\, Q_1X_1 \dots Q_nX_n\ \psi$. $\square$

Ex. 3.1.3 Transform the following formula to prenex NF:

$$\neg\exists X(\text{married}(X,Y) \wedge \underbrace{\neg\exists Y\, \text{motherOf}(X,Y)}_{\underbrace{\forall Y\, \neg\text{motherOf}(X,Y)}_{\forall Z\, \neg\text{motherOf}(X,Z)}})$$

$\forall Z \neg \mathit{motherOf}(X,Z)$

$\neg \exists X \; \forall Z (\mathit{married}(X,Y) \wedge \neg \mathit{motherOf}(X,Z))$

$\forall X \; \exists Z \; \neg (\mathit{married}(X,Y) \wedge \neg \mathit{motherOf}(X,Z))$

Ex 3.14 Consider our example LP and the query ?- motherOf(X, susanne).

We want to prove

$\mathit{motherOf}(\mathit{renate}, \mathit{sus}) \models \exists X \; \mathit{motherOf}(X, \mathit{susanne})$

To this end, we have to show unsatisfiability of

$\mathit{motherOf}(\mathit{ren}, \mathit{sus}) \wedge \neg \exists X \; \mathit{motherOf}(X, \mathit{sus})$.

First, this formula is transformed to prenex NF:

$\forall X \; (\mathit{motherOf}(\mathit{ren}, \mathit{sus}) \wedge \neg \mathit{motherOf}(X, \mathit{sus}))$

Def 3.15 (Skolem NF)

A formula φ is in Skolem normal form iff it is closed (i.e., it has no free variables) and it has the form $\forall X_1, \ldots, X_n \; \psi$ where ψ is quantifier-free.

Goal: obtain Skolem NF automatically

Solution: first transform to prenex NF, then remove free variables and $\exists$

There exist formulas φ where there is no equivalent formula φ' in Skolem NF.

Ex: female(X)

$\exists X$ female(X)

But: for every formula φ there exists a "satisfiability-equivalent" formula in Skolem NF.

Thm 3.1.6 (Transf. in Skolem NF)

For every formula φ, one can automatically construct a formula φ' in Skolem normal form such that φ is satisfiable iff φ' is satisfiable.

Proof: First, transform φ to prenex NF as in Thm 3.1.2. This results in φ_1.

Let $X_1, \ldots, X_n$ be the free variables of φ_1.

Then transform φ_1 into

$$\underbrace{\exists X_1, \ldots, X_n \quad \varphi_1}_{\varphi_2}.$$

$\leftarrow$ This is not equivalent to φ_1, but satisfiability-equivalent.

Finally, remove $\exists$ from φ_2 (φ_2 is closed and in prenex NF).

We remove $\exists$ from the outside to the inside:

If φ_2 has the form $\forall X_1, \ldots X_n \; \exists Y \; \psi$, then replace it by $\forall X_1, \dots, X_n \;\; \psi[Y / f(X_1, \dots, X_n)]$.

fresh fct. symbol of arity n

This is repeated until all $\exists$ have been removed.

The resulting formula is satisfiability-equivalent to the original formula (follows from substitution lemma). $\square$

Ex 3.17 In Ex 3.13 we obtained the following formula in prenex NF:

$$\forall X \; \exists Z \; \neg(\mathit{married}(X,Y) \lor \neg \mathit{motherOf}(X,Z))$$

$$\Downarrow$$

$$\exists Y \; \forall X \; \exists Z \; \neg(\mathit{married}(X,Y) \lor \neg \mathit{motherOf}(X,Z))$$

$$\Downarrow$$

$$\forall X \; \exists Z \; \neg(\mathit{married}(X,a) \lor \neg \mathit{motherOf}(X,Z))$$

$$\Downarrow$$

$$\forall X \;\; \neg(\mathit{married}(X,a) \lor \neg \mathit{motherOf}(X,f(X)))$$

3.2 Herbrand Structures

Friday, 17 April, 2015 10:00

3.2 Herbrand-Structures

Now the goal is to check unsatisfiability of a formula in Skolem NF.

$\Rightarrow$ we have to investigate all interpretations $I = (\mathcal{A}, \alpha, \beta)$ and check whether they satisfy the formula.

But: for formulas in Skolem NF, we can restrict ourselves to very special interpretations.

β: not necessary for closed formulas

$\mathcal{A}$: choose $\mathcal{A} := \mathcal{T}(\Sigma)$, i.e., we use the set of all ground terms as domain

α: we fix α_f to be "the function symbol itself".
Now one only has to search for α_p for $p \in \Delta$.
$\Rightarrow$ Search space is much smaller

<u>Def 3.2.1</u> (Herbrand Structures)

Let (Σ, Δ) be a signature. A <u>Herbrand structure</u> has the form $(\mathcal{T}(\Sigma), \alpha)$ where for all $f \in \Sigma_n$ we have:

$$\alpha_f(t_1, \dots, t_n) = f(t_1, \dots, t_n).$$

If a Herbrand structure is a model of a formula, we call it a <u>Herbrand model</u>.

<u>Ex. 3.2.2.</u> A Herbrand structure for the signature of Ex. 2.12 is: $S = (\mathcal{T}(\Sigma), \alpha)$ with

$\alpha_n = n$ for all $n \in \mathbb{N}$

$\alpha_{monika} = monika, \ldots$

$\alpha_{date}(t_1, t_2, t_3) = date(t_1, t_2, t_3)$ for all $t_1, t_2, t_3 \in \mathcal{T}(\Sigma)$

$\alpha_{female} = \{monika, Karin, \ldots\}$

$\alpha_{male} = \{werner, \ldots\}$

$\alpha_{human} = \mathcal{T}(\Sigma)$

$\alpha_{born} = \{(monika, date(17, 4, 2015)), \ldots\}$

(much nearer to the intuitive semantics)

Looking at H-structures is enough when checking for unsatisfiability of formulas in Skolem NF.

Thm 323 (Satisfiability Check by Herbrand Structures)

Let $\Phi \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ be a set of formulas in Skolem NF.

Then Φ is satisfiable iff it has a Herbrand model.

Proof: "$\Leftarrow$": trivial

"$\Rightarrow$": Let $S = (\mathcal{A}, \alpha)$ be a model of Φ.

We now construct a H-structure $S' = (\mathcal{T}(\Sigma), \alpha')$ that is also a model of Φ.

For every $f \in \Sigma_n$, we have $\alpha'_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$.

We define α'_p as follows:

for $p \in \Delta_n$ with $n \geq 1$ we define

$$(t_1, \ldots, t_n) \in \alpha'_p \quad \text{iff} \quad (S(t_1), \ldots, S(t_n)) \in \alpha_p$$

for $p \in \Delta_0$ we define

$$\alpha'_p = \alpha_p$$

Clearly, S' is a H-structure.

It remains to show that for every formula φ in Skolem NF, $S \models \varphi$ implies $S' \models \varphi$.

Since φ is in Skolem NF, it has the form $\forall X_1, \dots, X_n\ \psi$.

We prove "$S \models \varphi \curvearrowright S' \models \varphi$" by induction on n.

<u>Ind. Base</u>: $n = 0$

Here, φ is quantifier-free.

In this case, we even have $S \models \varphi$ iff $S' \models \varphi$ (easy structural induction on φ).

<u>Ind. Step</u>: $n > 0$

$\forall X_1, \dots, X_{n-1}\ \psi$ might contain the free var. X_n

Let $S[\![X_n/a]\!]$ denote an interpretation

$$(\underbrace{\mathcal{A}, \alpha}_{\text{same as for } S = (\mathcal{A}, \alpha)}, \beta[\![X_n/a]\!]) \text{ for some } \beta.$$

Then:

$S \models \forall X_1, \dots, X_n\ \psi$

$\curvearrowleft\!\!\curvearrowright\ S[\![X_n/a]\!] \models \forall X_1, \dots, X_{n-1}\ \psi$ for all $a \in \mathcal{A}$

$\curvearrowright\ S[\![X_n/S(t)]\!] \models \forall X_1, \dots, X_{n-1}\ \psi$ for all $t \in \mathcal{T}(\Sigma)$

$\curvearrowleft\!\!\curvearrowright\ S \models \forall X_1, \dots, X_{n-1}\ \psi[X_n/t]$ for all $t \in \mathcal{T}(\Sigma)$,

by the subst. lemma 2.2.3.

$\rightsquigarrow\ S' \models \forall X_1,\dots,X_{n-1}\ \varphi[X_n/t]$ for all $t \in \mathcal{T}(\Sigma)$,
by the ind. hypothesis

$\rightsquigarrow\ S'[\![X_n / \underbrace{S'(t)}_{t,\text{ because } S' \text{ is a H-structure}}]\!] \models \forall X_1,\dots,X_{n-1}\ \varphi$ for all $t \in \mathcal{T}(\Sigma)$

$\rightsquigarrow\ S' \models \forall X_1,\dots,X_n\ \varphi$ $\boxed{q}$

Ex. 324 Thm 323 only holds for formulas in Skolem NF.

Consider $\Phi = \{\ p(a),\ \exists X\, \neg p(X)\ \}$.

Φ is satisfiable, but it has no Herbrand model over the signature (Σ, Δ) where $\Sigma = \Sigma_0 = \{a\}$ and $\Delta = \Delta_1 = \{p\}$.

The following structure S is a model of Φ:

$S = (\{0,1\}, \alpha)$ where $\alpha_a = 0$

$\alpha_p = \{0\}$

$S \models p(a) \qquad S \models \exists X\ \neg p(X)$

but there is an element in the domain of S that does not correspond to any ground term:

$S(t) \neq 1$ for all $t \in \mathcal{T}(\Sigma)$

Since $\mathcal{T}(\Sigma) = \{a\}$, any H-structure S' has the domain $\{a\}$ and therefore $S' \models p(a)$ implies $S' \not\models \exists X\ \neg p(X)$.

For formulas in Skolem NF:

$$\forall X_1, \dots, X_n \quad \psi$$

One only has to instantiate $X_1, \dots, X_n$ by all possible ground terms and check whether all of the resulting formulas are satisfiable.

<u>Def 3.25</u> (Herbrand-expansion of a formula)
Let $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ be a formula in Skolem NF, i.e., $\varphi = \forall X_1, \dots, X_n \ \psi$ where ψ is quantifier-free.
The following set of formulas $E(\varphi)$ is called the <u>Herbrand-expansion</u> of φ:

$$E(\varphi) = \{ \psi[X_1/t_1, \dots, X_n/t_n] \mid t_1, \dots, t_n \in \mathcal{T}(\Sigma) \}$$

(i.e., it is the set of all <u>ground instances</u> of ψ).

---

terms (syntactic)

$\varphi[X_1/t_1, \dots, X_n/t_n]$ is φ with a <u>substitution</u> mapping X_i to t_i

$I[\![X_1/a_1, \dots, X_n/a_n]\!]$ is an interpretation with a <u>variable assignment</u> assigning a_i to X_i

elements of the domain (semantic)

<u>Ex. 3.26</u> To prove the query ?- motherOf(X, susanne).

one has to prove unsatisfiability (cf. Ex. 3.1.4.)

$\varphi = \forall X\, (\, motherOf(renate, susanne) \wedge \neg motherOf(X, susanne))$.

$E(\varphi) = \{\, mO(ren, sus) \wedge \neg mO(karin, susanne),$

$\quad \underline{mO(ren, sus) \wedge \neg mO(ren, sus)},$

$\quad mO(ren, sus) \wedge \neg mO(date(17, 4, 2015), sus),$

$\quad \vdots$

$\}$

We will see that

φ is satisfiable iff $E(\varphi)$ is satisfiable

Since the red subformula is unsat.

$\Rightarrow E(\varphi)$ is unsat

$\Rightarrow \varphi$ is unsat

$\Rightarrow$ query is true.

<u>Thm 3.2.7</u> (Satisfiability of Herbrand-expansion)

Let φ be a formula in Skolem NF.

Then φ is satisfiable iff $E(\varphi)$ is satisfiable.

<u>Proof</u>: φ has the form $\forall X_1, \dots, X_n\ \psi$ where ψ is quantifier-free.

$\underset{\sim}{\varphi}$ is satisfiable

$\forall X_1, \dots, X_n\ \psi$

iff there is a Herbrand-structure S with

$S \models \forall X_1, \dots, X_n\ \psi$ (Thm 3.2.3)

iff there is a H-str. S with

$S⟦X_1/t_1, \dots, X_n/t_n⟧ \models \psi$ for all $t_1, \dots, t_n \in T(\Sigma)$

iff there is a H-str. S with

$S \models \psi[X_1/t_1, \dots, X_n/t_n]$ for all $t_1, \dots, t_n \in T(\Sigma)$

(by the subst. lemma 2.2.3)

iff there is a H-str. S with

$S \models E(\varphi)$

iff $E(\varphi)$ is satisfiable

$\boxtimes$

For a formula φ in Skolem NF:

To check whether φ is unsatisfiable, we can construct $E(\varphi)$ and check whether some finite subset of $E(\varphi)$ is unsatisfiable.

(Compactness Theorem: if an infinite set of formulas is unsatisfiable, then there is already a finite subset that is unsatisfiable).

$\Rightarrow$ Algorithm of Gilmore

First semi-decision procedure for entailment/unsatisfiability.

Formulas without variables correspond to propositional logic:

- every occurring atomic sub-formula corresponds to a propositional variable (i.e., it can be either TRUE or FALSE)

- by trying out all truth assignments for these prop. variables, one can decide satisfiability of formulas without variables

Ex 328 We wanted to check satisfiability of

$E(\varphi) = \{ \underbrace{mO(ren, sus)}_{V_{mo(r,s)}} \wedge \neg \underbrace{mO(Karin, sus)}_{V_{mo(K,s)}},$

$\underbrace{mO(ren, sus)}_{V_{mo(r,s)}} \wedge \neg \underbrace{mO(ren, sus)}_{V_{mo(r,s)}}, \dots \}$

One can replace all atomic sub-formulas by propositional variables:

$$\{ V_{mo(r,s)} \wedge \neg V_{mo(K,s)},$$
$$V_{mo(r,s)} \wedge \neg V_{mo(r,s)}, \dots \}$$

Then construct finite subsets step by step and check satisfiability by mapping $V_{\dots}$ to {TRUE, FALSE}

3.3 Ground Resolution

Freitag, 24. April 2015 08:30

Drawbacks of Gilmore's Algorithm:

- unclear with which ground terms one should instantiate variables
- to check whether a ground formula is satisfiable:
  try out all possible assignments of $\underbrace{\text{atomic ground formulas}}$ to {TRUE, FALSE}

  correspond to propositional variables

Why is it not enough to check unsatisfiability of
ψ_1 or ψ_2 or ψ_3 or ... ?

Ex: p(0).
p(s(X)) :- p(X).

$\left(\begin{array}{c} 0 \hat{=} 0 \\ s(0) \hat{=} 1 \\ s(s(0)) \hat{=} 2 \\ \vdots \end{array} \right)$

?- p(s(s(0))).

$\varphi_1: \ p(0).$

$\varphi_2: \ \forall X \ \ p(X) \to p(s(X))$

$\varphi: \ \ p(s(s(0)))$

We have to check unsatisfiability of

$\psi: \forall X \ \ p(0) \wedge (p(X) \to p(s(X))) \wedge \neg p(s(s(0)))$

$[X/0]: \ \psi_1: \ p(0) \wedge (p(0) \to p(s(0))) \wedge \neg p(s(s(0)))$

Satisfiable: p(0): TRUE, p(s(0)): TRUE, p(s(s(0))): FALSE

$[X/s(0)]: \ \psi_2: \ p(0) \wedge (p(s(0)) \to p(s(s(0)))) \wedge \neg p(s(s(0)))$

satisfiable: $p(0)$: TRUE, $p(s(0))$: FALSE, $p(s(s(0)))$: FALSE

$[X/s(s(0))]$: ψ_3: ...
also satisfiable

$\Rightarrow$ all ψ_i on their own are satisfiable
but $\psi_1 \wedge \psi_2$ is unsatisfiable

Reason: the same rule has to be applied several times with different instantiations.

<u>Goal</u>: Improve the 2nd drawback of Gilmore's algorithm (i.e., check unsatisfiability of ground formulas)

<u>Solution</u>: Resolution (today: ground resolution)

3.3. Ground Resolution

Input: Formula $\forall X_1, \ldots, X_n\ \psi$ in Skolem NF

Goal: check unsatisfiability

First step: transform quantifier-free formula ψ to <u>Conjunctive normal form</u> (CNF)

<u>Def 3.3.1</u> (CNF)

A formula ψ is in <u>CNF</u> iff it is quantifier-free and it has the following form:

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{\ldots}\ )$$

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

Here, $L_{i,j}$ are <u>literals</u>, i.e., they are atomic or negated atomic formulas (i.e., they have the form $p(t_1, \ldots, t_n)$ or $\neg p(t_1, \ldots, t_n)$).

For every literal L we define its <u>negation</u> $\overline{L}$ as follows:

$$\overline{L} = \begin{cases} \neg A, & \text{if } L = A \in At(\Sigma, \Delta, \mathcal{V}) \\ A, & \text{if } L = \neg A \text{ for } A \in At(\Sigma, \Delta, \mathcal{V}) \end{cases}$$

A set of literals is called a <u>clause</u>.

Every formula φ in CNF corresponds to the following <u>clause set</u>:

$$\mathcal{K}(\varphi) = \{ \underbrace{\{L_{1,1}, \ldots, L_{1,n_1}\}}_{\text{clause}}, \ldots, \underbrace{\{L_{m,1}, \ldots, L_{m,n_m}\}}_{\text{clause}} \}$$

So a clause stands for the universally quantified disjunction of its literals and a clause set corresponds to the conjunction of its clauses.

The <u>empty clause</u> is denoted $\square$ and it is unsatisfiable by definition.

<u>Thm 332</u> (Transformation to CNF)

For every quantifier-free formula φ one can automatically construct an equivalent formula φ' in CNF.

<u>Proof</u>: First, replace sub-formulas $\varphi_1 \leftrightarrow \varphi_2$ by

$(\psi_1 \to \psi_2) \wedge (\psi_2 \to \psi_1)$.

Then replace sub-formulas $\psi_1 \to \psi_2$ by $\neg \psi_1 \vee \psi_2$.

Then apply the following algorithm $CNF(\psi)$:

- if ψ is atomic, then return ψ
- if $\psi = \psi_1 \wedge \psi_2$, then $CNF(\psi_1) \wedge CNF(\psi_2)$
- if $\psi = \psi_1 \vee \psi_2$, then compute

$$CNF(\psi_1) = \bigwedge_{i \in \{1, \dots, m_1\}} \psi_i'$$

$$CNF(\psi_2) = \bigwedge_{j \in \{1, \dots, m_2\}} \psi_j''$$

Then return

$$\bigwedge_{\substack{i \in \{1, \dots, m_1\} \\ j \in \{1, \dots, m_2\}}} (\psi_i' \vee \psi_j'')$$

$(\psi_1' \wedge \psi_2') \vee$

$(\psi_1'' \wedge \psi_2'')$

is equivalent to

$(\psi_1' \vee \psi_1'') \wedge$

$(\psi_1' \vee \psi_2'') \wedge$

$(\psi_2' \vee \psi_1'') \wedge$

$(\psi_2' \vee \psi_2'')$

Distribution Law

- if $\psi = \neg \psi_1$, then compute

$$CNF(\psi_1) = \bigwedge_{i \in \{1, \dots, m\}} \Big(\bigvee_{j \in \{1, \dots, n_i\}} L_{i,j} \Big)$$

Applying De Morgan Laws results in

$$\bigvee_{i \in \{1, \dots, m\}} \Big(\bigwedge_{j \in \{1, \dots, n_i\}} \overline{L_{i,j}} \Big)$$

Applying the distribution law yields the following formula that is returned:

$$\bigwedge_{j_1 \in \{1, \dots, n_1\}} \big(\overline{L_{1,j_1}} \vee \dots \vee \overline{L_{m,j_m}} \big)$$

$j_1 \in \{1,\dots,n_1\}$
$\vdots$
$j_m \in \{1,\dots,n_m\}$

$\dots, j_m$

Ex. 3.3.3 φ is the following formula with $p, q, r \in \Delta_0$:

$$\neg(\neg p \wedge (\neg q \vee r))$$

De Morgan laws yield:

$$p \vee (q \wedge \neg r)$$

Distribution law results in:

$$(p \vee q) \wedge (p \vee \neg r) \quad \leftarrow \text{in CNF}$$

Remaining goal: Check unsatisfiability of a ground formula in CNF, i.e., of a set of ground clauses.

Def 3.3.4 (Propositional Resolution)

Let K_1, K_2 be ground clauses. Then the clause R is a resolvent of K_1 and K_2 iff there is a literal $L \in K_1$ with $\overline{L} \in K_2$ and $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$.

For a clause set $\mathcal{K}$ we define

$$Res(\mathcal{K}) = \mathcal{K} \cup \{R \mid R \text{ is resolvent of two clauses from } \mathcal{K}\}.$$

Moreover, let

$$\mathrm{Res}^0(\mathcal{K}) = \mathcal{K}$$

$$\mathrm{Res}^{n+1}(\mathcal{K}) = \mathrm{Res}\left(\mathrm{Res}^n(\mathcal{K})\right) \text{ for all } n \geq 0.$$

So the set of all clauses that can be deduced by resolution is

$$\mathrm{Res}^*(\mathcal{K}) = \bigcup_{n \geq 0} \mathrm{Res}^n(\mathcal{K})$$

Obviously, we have $\square \in \mathrm{Res}^*(\mathcal{K})$ iff there is a sequence of clauses $K_1, \dots, K_m$ such that the following holds for all $1 \leq i \leq m$:

- $K_i \in \mathcal{K}$ or
- K_i is a resolvent of K_j and K_k for $j, k < i$.

To display resolution proofs, we often use diagrams:

$$\begin{array}{ccc} K_1 & & K_2 \\ & \searrow \quad \swarrow & \\ & R & \end{array}$$

means that R is resolvent of K_1 and K_2

Ex 335 $\square$ can be derived

We now have to show that

$$\underbrace{\square \in \mathrm{Res}^*(\mathcal{K})}_{\text{syntax, can be checked}} \quad \text{iff} \quad \underbrace{\mathcal{K} \text{ is unsatisfiable}}_{\text{semantics}}$$

$\Rightarrow$: soundness
$\Leftarrow$: completeness

of resolution

automatically

To prove soundness of ground resolution, we show that adding resolvents preserves equivalence.

<u>Lemma 336</u> (Propositional Resolution Lemma)
Let $\mathcal{K}$ be a set of ground clauses. If $K_1, K_2 \in \mathcal{K}$ and R is resolvent of K_1 and K_2, then $\mathcal{K}$ and $\mathcal{K} \cup \{R\}$ are equivalent.

<u>Proof</u>: "$\Leftarrow$": If there is a structure S with $S \models \mathcal{K} \cup \{R\}$, then also $S \models \mathcal{K}$.

"$\Rightarrow$": Let $S \models \mathcal{K}$.
There is a literal $L \in K_1$, $\overline{L} \in K_2$, $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$.

Assume that $S \not\models \mathcal{K} \cup \{R\}$, i.e., $S \not\models R$

If $S \models L$, then $S \models \mathcal{K}$ implies $S \models K_2$ which in turn implies $S \models K_2 \setminus \{\overline{L}\}$. Thus, $S \models R$ ↯.

If $S \not\models L$, then in a similar way one can show $S \models K_1 \setminus \{L\}$. Thus, $S \models R$ ↯.

$\boxed{q}$

<u>Thm 337</u> (Soundness and Completeness of propositional resolution)

Let $\mathcal{K}$ be a (possibly infinite) set of ground clauses.
Then: $\square \in Res^*(\mathcal{K})$ iff $\mathcal{K}$ is unsatisfiable

Then: $\square \in Res^*(\mathcal{K})$ iff $\mathcal{K}$ is unsatisfiable.

Proof: "$\Rightarrow$" (Soundness)

Resolution Lemma 336 states that $\mathcal{K}$ and $Res(\mathcal{K})$ are equivalent.

By induction, one can show that $\mathcal{K}$ is equivalent to $Res^n(\mathcal{K})$ for all $n \in \mathbb{N}$.

$\square \in Res^*(\mathcal{K})$

$\rightsquigarrow$ there is an $n \in \mathbb{N}$ such that $\square \in Res^n(\mathcal{K})$

$\rightsquigarrow$ $Res^n(\mathcal{K})$ is unsatisfiable

$\rightsquigarrow$ $\mathcal{K}$ is unsatisfiable.

"$\Leftarrow$": (Completeness)

$\mathcal{K}$ is unsatisfiable

$\rightsquigarrow$ there is a finite subset $\mathcal{K}' \subseteq \mathcal{K}$ that is unsatisfiable

We prove $\square \in Res^*(\mathcal{K}')$ by induction on the number n of different atomic formulas in $\mathcal{K}'$!

Ind Base: $n = 0$

There are only 2 clause sets without atomic formulas:

$\mathcal{K}' = \emptyset$ is valid (holds in every structure)

or $\mathcal{K}' = \{\square\}$ is unsatisfiable

Then $\square \in Res^0(\mathcal{K}') \subseteq Res^*(\mathcal{K}')$.

Ind Step: $n > 0$

Let A be an atomic formula occurring in $\mathcal{K}'$!

Let $\mathcal{K}^+$ result from $\mathcal{K}'$ by omitting all clauses that contain A. Moreover, $\neg A$ is removed from all remaining clauses:

$$\mathcal{K}^+ = \{\, K \setminus \{\neg A\} \mid K \in \mathcal{K}',\ A \notin K \,\}$$

$$\mathcal{K}^- = \{\, K \setminus \{A\} \mid K \in \mathcal{K}',\ \neg A \notin K \,\}$$

Clearly, A does not occur anymore in $\mathcal{K}^+$ and $\mathcal{K}^-$.
Thus: $\mathcal{K}^+$, $\mathcal{K}^-$ contain at most $n-1$ atomic formulas.

$\mathcal{K}^+$ is unsatisfiable:
If $S \models \mathcal{K}^+$, then S could be extended to a structure S' with $S' \models A$. Then: $S' \models \mathcal{K}'$. ↯ to the unsatisfiability of $\mathcal{K}'$.

Induction Hypothesis:

$$\square \in \text{Res}^*(\mathcal{K}^+), \quad \square \in \text{Res}^*(\mathcal{K}^-)$$

This means that there is a sequence of clauses $K_1, \dots, K_m$ with $K_m = \square$ and for all $1 \le i \le m$:

- $K_i \in \mathcal{K}^+$ or
- K_i is a resolvent of K_j and K_k for $j, k < i$

If those clauses $K_i \in \mathcal{K}^+$ that were used in the resolution proof are also contained in $\mathcal{K}'$, then this is already a resolution proof from $\mathcal{K}'$, i.e., $\square \in \text{Res}^*(\mathcal{K}')$.
Otherwise: re-insert $\neg A$ into those clauses K_i where it had been removed. This yields again a resolution proof from $\mathcal{K}'$ ending in $\{\neg A\}$.

from $\mathcal{K}'$ ending in $\{\neg A\}$.

(Reason:

$$\begin{tikzcd}[column sep=small] K_j \arrow[dr, dash] & & K_k \arrow[dl, dash] \\ & K_i & \end{tikzcd} \quad \rightsquigarrow \quad \begin{tikzcd}[column sep=small] K_j \cup \{\neg A\} \arrow[dr, dash] & & K_k \arrow[dl, dash] \\ & K_i \cup \{\neg A\} & \end{tikzcd}$$

)

$$\Rightarrow \{\neg A\} \in Res^*(\mathcal{K}').$$

Similarly, there is a resolution proof of $\square$ from $\mathcal{K}^-$.
If this proof used only clauses from $\mathcal{K}'$, then we directly have $\square \in Res^*(\mathcal{K}')$.
Otherwise: re-insert A into the clauses from $\mathcal{K}^-$

$$\Rightarrow \{A\} \in Res^*(\mathcal{K}').$$

One last resolution step yields $\square \in Res^*(\mathcal{K}')$:

$$\begin{tikzcd}[column sep=small] \{A\} \arrow[dr, dash] & & \{\neg A\} \arrow[dl, dash] \\ & \square & \end{tikzcd}$$

∎

Now we can improve the algorithm of Gilmore to the

<u>Ground Resolution Algorithm</u>.

Advantage over Gilmore's Alg: better check for unsatisfiability

Same disadvantage as Gilmore: step from predicate to propositional logic is done via Herbrand-expansion (instantiate variables by all possible ground terms)

Ground Res. Alg. is sound and complete:

- if $\{\varphi_1, \ldots, \varphi_n\} \models \varphi$, then alg. terminates and returns "true"
- if $\{\varphi_1, \ldots, \varphi_n\} \not\models \varphi$, then alg. does not return "true" (but it doesn't terminate in general)

Now: Improve the step from pred. to prop. logic
(avoid a blind guess by which ground terms one has to instantiate variables)

Ex 3.4.1 Up to now: before performing resolution, we have to instantiate variables by ground terms. This instantiation does not only have to enable the next resolution step, but one has to guess the right instantiation which also allows all needed future resolution steps.

We need an inst. for X and Y such that

$p(X)$ and $p(f(Y))$ become equal.

(i.e. we have to unify $p(X)$ and $p(f(Y))$)

But this instantiation should also allow future resolution steps (e.g., between $q(\ldots)$ and $\neg q(\ldots)$).

Solution: do not instantiate variables by ground terms, but also allow instantiations by arbitrary terms. Only look for most general unifiers, i.e., only instantiate them in such a way that the next resolution step is possible)

In the example: Finally one uses $\{Y/a\}$ to derive $\Box$.

Def 3.4.2 (Unification)
A clause $K = \{L_1, \ldots, L_n\}$ is unifiable iff there exists a substitution σ such that $\sigma(L_1) = \ldots = \sigma(L_n)$ (i.e.,

$|\underbrace{\sigma(K)}_{\{\sigma(L_1),\dots,\sigma(L_n)\}}| = 1$). Such a subst. σ is a <u>unifier</u> of K.

A unifier σ is a <u>most general unifier</u> (mgu) iff for any unifier σ' there exists a subst. δ such that

$$\sigma'(X) = \delta(\sigma(X)) \quad \text{for all } X \in \mathcal{V}.$$

In the example: $K = \{p(X), p(f(Y))\}$

mgu $\sigma = \{X/f(Y)\}$ $\quad |\sigma(K)| = |\{\sigma(p(X)), \sigma(p(f(Y)))\}| = |\{p(f(Y))\}| = 1$

alternative unifier $\sigma' = \{X/f(a), Y/a\}$

We have $\sigma' = \delta \circ \sigma$

for $\delta = \{Y/a\}$

<u>Observations</u>:

- If a clause is unifiable, then it also has an mgu.
- The mgu is unique up to variable renaming.

  Ex: $\{p(X), p(Y)\}$

  mgu $\sigma = \{X/Y\}$ or $\sigma' = \{Y/X\}$
- It is decidable whether a clause is unifiable and the mgu is computable.
- First unification algorithm by J. Robinson (1965).

  inventor of resolution

<u>Ex 343</u>

• Try to unify $\{ q(f(X,Y)), \quad q(g(X,Y)) \}$

$\sigma = \emptyset$

<u>clash failure</u>

• Try to unify $\{ q(X), \quad q(h(X)) \}$

$\sigma = \emptyset$

<u>occur failure</u>

• Try to unify $\{ p(f(Z, g(a,Y)), h(Z)), p(f(f(U,V), W), h(f(a,Y))) \}$

$\sigma = \emptyset$

$\sigma = \{ Z / f(U,V) \}$

$\{ p(f(f(U,V), g(a,Y)), h(f(U,V))), \; p(f(f(U,V), W), h(f(a,Y))) \}$

$\sigma = \{ W / g(a,Y) \} \circ \{ Z / f(U,V) \} = \{ W/g(a,Y), Z/f(U,V) \}$

$\{ p(f(f(U,V), g(a,Y)), h(f(U,V))), p(f(f(U,V), g(a,Y)), h(f(a,Y)) \}$

$\sigma = \{ U/a \} \circ \{ W/g(a,Y), Z/f(U,V) \} = \{ U/a, W/g(a,Y), Z/f(a,V) \}$

$\{ p(f(f(a,V), g(a,Y)), h(f(a,V))), p(f(f(a,V), g(a,Y)), h(f(a,Y)) \}$

$\sigma = \{ Y/V \} \circ \ldots = \{ Y/V, U/a, W/g(a,V), Z/f(a,V) \}$

$\{ p(\ldots), \; p(\ldots) \}$

are the same now

Friday (May 8): 2 lectures (lecture instead of exerc. course)

Monday (May 11): ex. course instead of lecture

<u>Thm 3.44</u> (Termination + Soundness of Unif. Alg.)

The unif. alg. terminates for every clause K and it is sound, i.e., it returns an mgu for K iff K is unifiable.

<u>Proof</u>: The alg. terminates because the number of variables in the clause decreases in each loop iteration.

If the alg. returns a subst. σ, then σ is a unifier of K (since it checks $|\sigma(K)| = 1$ in step 2).

Thus: if K is not unifiable

$\rightsquigarrow$ alg can't return a subst. σ

$\rightsquigarrow$ alg stops with failure. (since alg. terminates).

It remains to prove:

If K is unifiable, then alg. returns a mgu σ.

Let $m \geq 0$ be the number of loop iterations of the alg. for the input K. For every $0 \leq i \leq m$, let σ_i be the value of σ after the i-th loop iteration.

We prove the following for all $0 \le i \le m$:

For every unifier σ' of K, we have $\sigma' = \sigma' \circ \sigma_i$. $(\forall)$

This implies the soundness of the alg. if K is unifiable:

- If the alg would stop with failure in the $(m+1)$-th loop iteration, then $\sigma_m(K)$ would not be unifiable.

  But $(\forall)$ implies: $\sigma' = \sigma' \circ \sigma_m$ and there exists a unifier σ' of K.

  $$|\sigma'(K)| = 1$$

  $$\curvearrowright |\sigma'(\sigma_m(K))| = 1$$

  $\curvearrowright$ σ' is a unifier of $\sigma_m(K)$. $\lightning$

- So the alg. has to stop with success

  $\curvearrowright$ $|\sigma_m(K)| = 1$, i.e., σ_m is a unifier.

  Now $(\forall)$ implies that for every unifier σ' there exists a subst δ (viz. $\delta = \sigma'$) such that:

  $$\sigma' = \underbrace{\delta}_{\sigma'} \circ \sigma_m$$

  $\curvearrowright$ σ_m is mgu of K.

Now we prove the following for all $0 \le i \le m$ by induction on i:

For every unifier σ', we have $\sigma' = \sigma' \circ \sigma_i$ $(\forall)$

<u>Ind Base</u>: $i = 0$

$\sigma_0 = \emptyset$ $\curvearrowright$ $\sigma' = \sigma' \circ \sigma_0$ holds for all substitutions σ'. ✓

<u>Ind Step</u>: $i > 0$

Ind. Hypothesis: $\sigma' = \sigma' \circ \sigma_{i-1}$

To unify $\sigma_{i-1}(K)$ one has to replace a var. X by a term t

in Step 6. Thus: $\sigma_i = \{X/t\} \circ \sigma_{i-1}$.

We have:

$$
\begin{aligned}
& \sigma' \circ \sigma_i \\
= \; & \underbrace{\sigma' \circ \{X/t\}} \circ \sigma_{i-1} \qquad \text{(by def. of } \sigma_i\text{)} \\
= \; & \sigma' \circ \sigma_{i-1} \\
= \; & \sigma' \qquad \text{(by the ind. hyp.)}
\end{aligned}
$$

Reason for $\sigma' = \sigma' \circ \{X/t\}$:

For $Y \neq X$: $\sigma'(Y) = (\sigma' \circ \{X/t\})(Y)$

For X: $(\sigma' \circ \{X/t\})(X) = \sigma'(t) = \sigma'(X)$

Reason: σ' is also a unifier of $\sigma_{i-1}(K)$

(since $|\sigma'(\sigma_{i-1}(K))| \underset{\text{by ind. hyp}}{=} |\sigma'(K)| = 1$)

Therefore, σ' must make X and t equal. $\boxtimes$

<u>Def 345</u> (Resolution in Predicate Logic)

Let K_1, K_2 be clauses. Then a clause R is <u>resolvent</u> of K_1 and K_2 iff:

- There exist variable renamings ν_1, ν_2 such that $\nu_1(K_1)$ and $\nu_2(K_2)$ have no common variables.

  ok, since clause stands for universally quantified disjunction of its literals (i.e.: renaming of bound variables)

- There exist literals $L_1, \ldots, L_m \in \nu_1(K_1)$ and $L_1', \ldots, L_n' \in \nu_2(K_2)$ with

$L_1', \dots, L_n' \in \nu_2(K_2)$ with

(i.e.: renaming of bound variables)

$n, m \geq 1$ such that

$\{\overline{L}_1, \dots, \overline{L}_m, L_1', \dots, L_n'\}$ are unifiable with mgu σ

- $R = \sigma\big((\nu_1(K_1) \setminus \{L_1, \dots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \dots, L_n'\})\big)$

As before, we define the following for a clause set $\mathcal{K}$:

$$\mathrm{Res}(\mathcal{K}) = \mathcal{K} \cup \{R \mid R \text{ is resolvent of 2 clauses in } \mathcal{K}\}$$

$$\mathrm{Res}^0(\mathcal{K}) = \mathcal{K}$$

$$\mathrm{Res}^{n+1}(\mathcal{K}) = \mathrm{Res}(\mathrm{Res}^n(\mathcal{K})) \quad \text{for all } n \geq 0$$

$$\mathrm{Res}^*(\mathcal{K}) = \bigcup_{n \geq 0} \mathrm{Res}^n(\mathcal{K})$$

Clearly, propositional resolution is a special case of this form of resolution.

Ex 346

K_1: $\{p(f(X)), \neg q(Z), p(Z)\}$

K_2: $\{\neg p(X), r(g(X))\}$

$\nu_2(K_2)$: $\{\neg p(U), r(g(U))\}$

$\{\neg q(f(X)), r(g(f(X)))\}$

Variable renaming:

$\nu_1 = \emptyset$

$\nu_2 = \{X/U, U/X\}$ (must be injective)

$L_1 = p(f(X))$ $\qquad L_1' = \neg p(U)$

$L_2 = p(Z)$

$\sigma = mgu(\{\neg p(f(X)), \neg p(Z), \neg p(U)\}) = \{Z/f(X), U/f(X)\}$

Resolution in pred. logic is also sound + complete, i.e.: $\mathcal{K}$ is unsatisfiable iff $\square \in Res^*(\mathcal{K})$

Soundness can be shown in a similar way as in prop. logic.

Lemma 3.4.7 (Resolution Lemma for Pred. Logic)

Let $\mathcal{K}$ be a clause set. If $K_1, K_2 \in \mathcal{K}$ and R is a resolvent of K_1 and K_2, then $\mathcal{K}$ and $\mathcal{K} \cup \{R\}$ are equivalent.

Proof: similar to the propositional resolution lemma (Lemma 3.3.6) $\boxed{\varepsilon}$

This implies soundness of resolution:

$$\square \in Res^*(\mathcal{K})$$
$$\curvearrowright \quad \square \in Res^n(\mathcal{K}) \text{ for some } n \geq 0$$
$$\curvearrowright \quad Res^n(\mathcal{K}) \text{ is unsatisfiable}$$
$$\curvearrowright \quad \mathcal{K} \text{ is unsatisfiable}$$

(since the resolution lemma implies that $\mathcal{K}$ and $Res(\mathcal{K})$ are equivalent. Thus, by induction on n one can show that $\mathcal{K}$ and $Res^n(\mathcal{K})$ are equivalent.)

Now: Show completeness of resolution in pred. logic.

Goal: Use completeness of prop. resolution to show completeness of full resolution.

Solution: Lifting Lemma

shows how to "lift" a resolution proof from propositional to predicate logic.

Lemma 348 (Lifting Lemma)

Let K_1, K_2 be clauses with ground instances K_1', K_2'.
If R' is a (propositional) resolvent of K_1' and K_2',
then there exists a resolvent R of K_1 and K_2 such that
R' is a ground instance of R.

$$
\begin{tikzcd}
K_1 \arrow[d, "\text{ground instance}"] & & K_2 \arrow[d] \\
K_1' \arrow[dr, dash] & & K_2' \arrow[dl, dash] \\
& R' &
\end{tikzcd}
\quad \leadsto \quad
\begin{tikzcd}
K_1 \arrow[dr, dash] & & K_2 \arrow[dl, dash] \\
& R \arrow[d, "\text{ground instance}"] & \\
& R' &
\end{tikzcd}
$$

Ex 349 to illustrate the lifting lemma

K_1: $\{p(f(X)), \neg q(Z), p(Z)\}$ $\qquad$ K_2: $\{\neg p(X), r(g(X))\}$

$\downarrow$ ground subst $\{X/a, Z/f(a)\}$ $\qquad$ $\downarrow$ $\{X/f(a)\}$

K_1': $\{p(f(a)), \neg q(f(a))\}$ $\qquad$ K_2': $\{\neg p(f(a)), r(g(f(a)))\}$

$\{\neg q(f(a)), r(g(f(a)))\}$ — R'

The lifting lemma states that one could instead perform resolution on K_1, K_2 and obtain a resolvent R such that R' is a ground instance of R:

$\{p(f(X)), \neg q(Z), p(Z)\}$ — K_1 $\qquad$ $\{\neg p(X), r(g(X))\}$ — K_2

$\{\neg q(f(X))$ … — R

$$\{\neg q(f(X)), r(g(f(X)))\} - K$$

$\downarrow$ ground inst. with $\{X/a\}$

$$\{\neg q(f(a)), r(g(f(a)))\} - R'$$

Proof of the Lifting Lemma:

$$\begin{tikzcd}
K_1 \arrow[d] & & K_2 \arrow[d] \\
K_1' \arrow[dr, dash] & & K_2' \arrow[dl, dash] \\
& R' &
\end{tikzcd}$$

Let ν_1, ν_2 be var. renamings such that $\nu_1(K_1)$ and $\nu_2(K_2)$ are variable-disjoint.

Then K_1', K_2' are also ground instances of $\nu_1(K_1)$ and $\nu_2(K_2)$.

Since $\nu_1(K_1)$ and $\nu_2(K_2)$ are variable-disjoint, one can use the <u>same</u> ground subst. σ:

$$\sigma(\nu_1(K_1)) = K_1' \quad \text{and} \quad \sigma(\nu_2(K_2)) = K_2'$$

Since R' is resolvent of K_1' and K_2', there is a literal

$$L \in K_1' \text{ with } \overline{L} \in K_2' \text{ and}$$

$$R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}).$$

Let $L_1, \dots, L_m$ be <u>all</u> literals from $\nu_1(K_1)$ that are mapped to L by σ (i.e., $\sigma(L_1) = \dots = \sigma(L_m) = L$).

Let $L_1', \dots, L_n'$ be <u>all</u> literals from $\nu_2(K_2)$ that are mapped to $\overline{L}$ by σ (i.e., $\sigma(L_1') = \dots = \sigma(L_n') = \overline{L}$).

We must have $m, n \geq 1$.

Since σ is a unifier of $\{\overline{L}_1, \dots, \overline{L}_m, L_1', \dots, L_n'\}$, there also exists a mgu σ'.

Therefore, K_1 and K_2 have a resolvent

$$R = \sigma'\big((\nu_1(K_1) \setminus \{L_1, \dots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \dots, L_n'\}) \big).$$

Thus:

$$\begin{tikzcd}
K_1 \arrow[d, "\sigma"] & & K_2 \arrow[d, "\sigma"] \\
K_1' \arrow[dr] & & K_2' \arrow[dl] \\
& R' &
\end{tikzcd}
\quad \sim \quad
\begin{tikzcd}
K_1 \arrow[dr] & & K_2 \arrow[dl] \\
& R \arrow[d, "\delta"] & \\
& R' &
\end{tikzcd}
\Big\} \text{ This still has to be shown.}$$

Since σ is a unifier of $\{\overline{L}_1, \dots, \overline{L}_m, L_1', \dots, L_n'\}$ and σ' is a mgu, there exists a substitution δ such that

$$\sigma = \delta \circ \sigma'$$

We now show that R' is indeed an instance of R:

$$\begin{aligned}
R' &= (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}) \\
&= (\sigma(\nu_1(K_1)) \setminus \{L\}) \cup (\sigma(\nu_2(K_2)) \setminus \{\overline{L}\}) \\
&= \sigma\big((\nu_1(K_1) \setminus \{L_1, \dots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \dots, L_n'\})\big)
\end{aligned}$$

(since $L_1, \dots, L_m$ are all literals from $\nu_1(K_1)$ that are mapped to L by σ)

$$\begin{aligned}
&= \delta\big(\underbrace{\sigma'\big((\nu_1(K_1) \setminus \{L_1, \dots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \dots, L_n'\})\big)}_{R}\big) \\
&= \delta(R). \qquad \boxtimes
\end{aligned}$$

Thm 3.4.10 (Soundness + Completeness of Resolution in Pred. Logic)

Let $\mathcal{K}$ be a finite clause set. Then

$\mathcal{K}$ is unsatisfiable iff $\square \in Res^*(\mathcal{K})$.

Proof: "$\Leftarrow$" (Soundness): direct consequence of the resolution lemma 3.47.

"$\Rightarrow$" (Completeness):

$\mathcal{K}$ is unsatisfiable

$\Updownarrow$ by Thm 3.27: $\underbrace{\text{Herbrand-expansion of } \mathcal{K}}$ is also unsatisfiable

The set of clauses containing all ground instances of clauses from $\mathcal{K}$, i.e.,

$$\{ \sigma(K) \mid K \in \mathcal{K}, \sigma(K) \text{ contains no variables} \}$$

$\Updownarrow$ by Thm 3.37 (completeness of resolution in propositional logic):

One can deduce $\square$ from the Herbrand-exp. of $\mathcal{K}$, i.e.,

$\square \in Res^*(\{\sigma(K) \mid K \in \mathcal{K}, \sigma(K) \text{ has no variables}\})$.

$\Updownarrow$ There exists a sequence of ground clauses

$K_1', \ldots, K_m'$ with $K_m' = \square$ and

for all $1 \leq i \leq m$:

- K_i' is a ground instance of a clause from $\mathcal{K}$ or
- K_i' is resolvent of K_j' and K_k' for $j, k < i$

Now we "lift" this resolution proof to predicate logic, i.e., we create a sequence $K_1, \dots, K_m$ where each

- K_i' is an instance of K_i and
- $K_i \in Res^*(\neg\mathcal{K})$

Definition of K_i:

- if K_i' is a ground instance of some $K \in \neg\mathcal{K}$, then choose $K_i := K$
- otherwise, K_i' is a resolvent of K_j' and K_k' for $j, k < i$. We already defined K_j, K_k such that K_j' and K_k' are instances of K_j and K_k resp., and $K_j, K_k \in Res^*(\neg\mathcal{K})$.

$$\begin{tikzcd}[column sep=small] K_j \arrow[d] & & K_k \arrow[d] \\ K_j' \arrow[dr, dash] & & K_k' \arrow[dl, dash] \\ & K_i' & \end{tikzcd} \qquad \text{Lifting Lemma 3.4.8} \qquad \begin{tikzcd}[column sep=small] K_j \arrow[dr, dash] & & K_k \arrow[dl, dash] \\ & K_i \arrow[d] & \\ & K_i' & \end{tikzcd}$$

Lifting lemma states that there exists a resolvent of K_j and K_k such that K_i' is an instance of this resolvent. Choose K_i to be this resolvent.

$\Rightarrow K_i \in Res^*(\neg\mathcal{K})$

Thus: K_m' is an instance of K_m and $K_m \in Res^*(\neg\mathcal{K})$

$$\begin{tikzcd} K_m' \arrow[d, dash] \\ \Box \end{tikzcd}$$

$\Rightarrow K_m = \Box$ and $\Box \in Res^*(\neg\mathcal{K})$. $\blacksquare$

Now we can improve our algorithm to check entailment by using resolution in pred. logic.

Alg is a semi-decision procedure:

If $\{\varphi_1, \dots, \varphi_n\} \models \varphi$, then the alg. terminates and returns "true".

If $\{\varphi_1, \dots, \varphi_n\} \not\models \varphi$, then the alg may not terminate. But if it terminates, then it returns "false".

Alg. is feasible in practice for small problems, but still ~~too~~ inefficient in general.

Problem: one has to generate all resolvents repeatedly (one also has to resolve clauses that were created by earlier resolution steps).

Goal: Restrict resolution (i.e., do not create all possible resolvents) without losing completeness

3.5 Restrictions of Resolution

Freitag, 8. Mai 2015 10:00

4 restrictions of resolution:

- linear resolution (still complete)
- input resolution (no longer complete, but still complete on Horn clauses, i.e., on the clauses used in logic programs)
- SLD resolution (complete on Horn clauses)
- binary SLD resolution (— " ——)

This is the form of resolution used in logic programming.

3.5.1. Linear Resolution

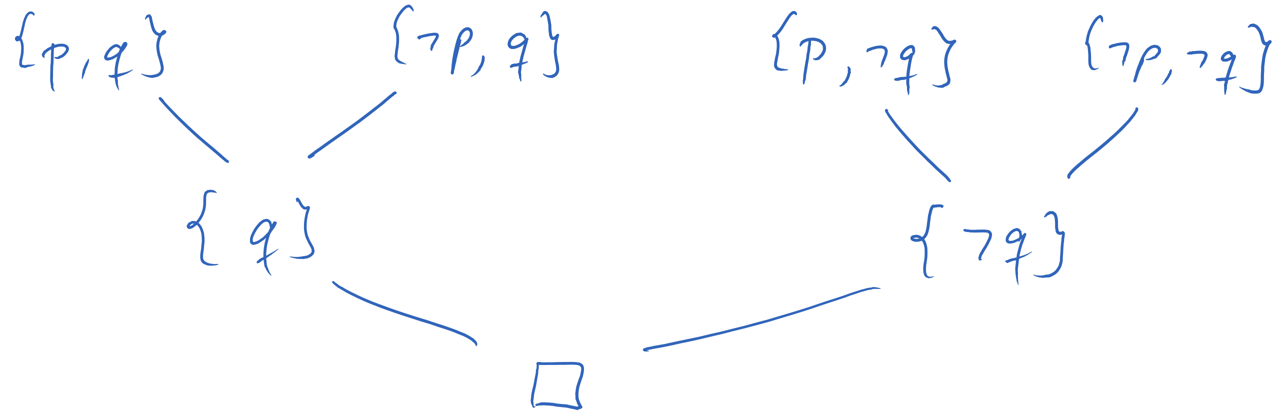
Restrict resolution in the following way: One of the parent clauses in the next resolution step must be the resolvent that was produced in the step before.

Def 3.5.1 (Linear Resolution)

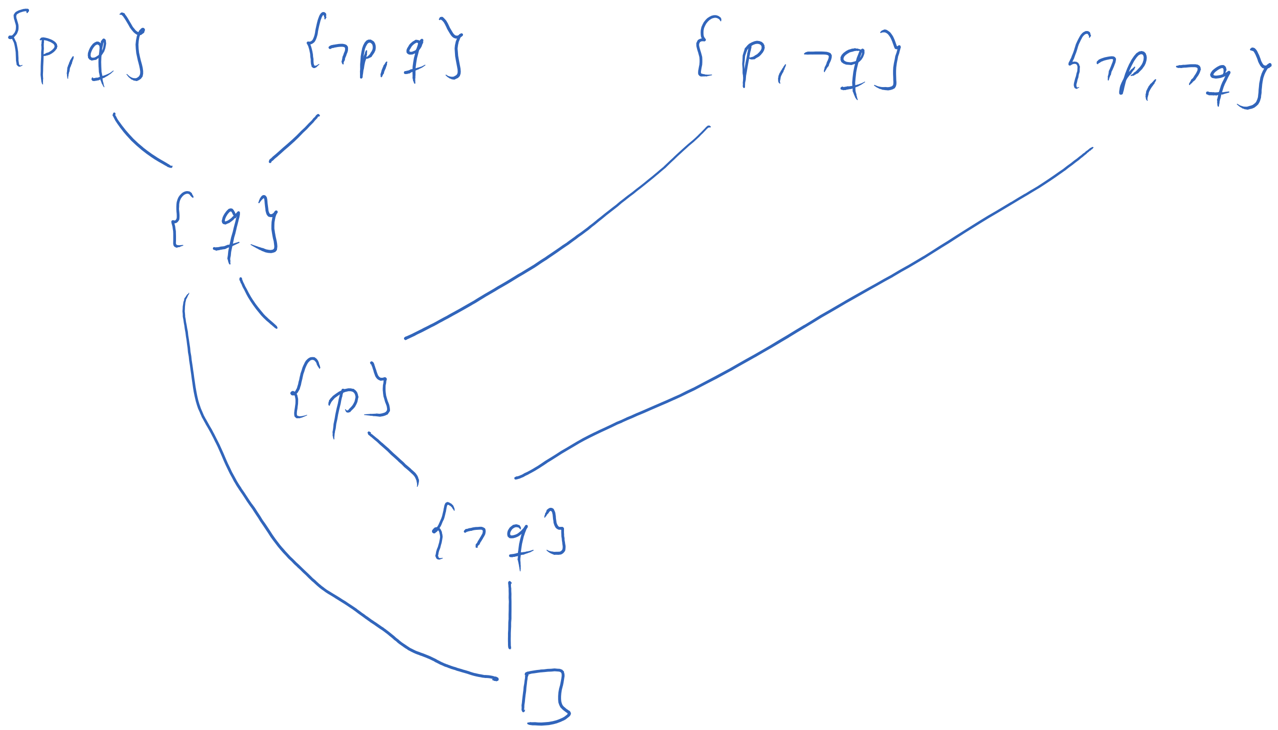
Let $\mathcal{K}$ be a clause set. $\square$ can be obtained from $K \in \mathcal{K}$ by linear resolution iff there is a sequence $K_1, \dots, K_m$ such that

- $K_1 = K \in \mathcal{K}$
- $K_m = \square$
- for all $2 \leq i \leq m$: K_i is a resolvent of K_{i-1} and a clause from $\{K_1, \dots, K_{i-1}\} \cup \mathcal{K}$

Ex 352



This is not a linear resolution proof.
Is linear resolution still complete?
Can we also derive □ by linear resolution?



Thm 353 (Soundness + Completeness of Linear Resolution)
Let $\mathcal{K}$ be a clause set. Then

$\mathcal{K}$ is unsatisfiable iff □ can be derived by linear resolution from some $K \in \mathcal{K}$.

If $\mathcal{K}$ is a <u>minimal</u> unsatisfiable clause set (i.e., for every $K \in \mathcal{K}$ the set $\mathcal{K} \setminus \{K\}$ is satisfiable), then $\Box$ can even be derived by linear resolution from <u>every</u> $K \in \mathcal{K}$.

<u>Proof:</u> "$\Leftarrow$" (Soundness): obvious, because every linear resolution proof is a proper resolution proof (and resolution is sound by Thm 3.4.10)

"$\Rightarrow$" (completeness):

- prove completeness of propositional linear resolution
- Then use the lifting lemma to lift any linear resolution proof of $\Box$ to a linear resolution proof in pred. logic.

$\boxed{\text{q.e.d.}}$

<u>3.5.2. Input- and SLD-Resolution</u>

Input Resolution: Special form of linear resolution (i.e., one parent clause must be the resolvent obtained in the last step). But now the other parent clause must be from the original clause set (i.e., it must not be a resolvent from an earlier step).

<u>Def 3.5.4</u> (Input Resolution)

Let $\mathcal{K}$ be a clause set. $\Box$ can be derived from a clause $K \in \mathcal{K}$ by <u>input resolution</u> iff there is a sequence of clauses

$K_1, \dots, K_m$ with
- $K_1 = K \in \mathcal{K}$
- $K_m = \Box$
- for all $2 \leq i \leq m$:
  K_i is a resolvent of K_{i-1} and a clause from $\mathcal{K}$.

Advantage: drastic reduction of search space for the next resolution step ($\mathcal{K}$ remains constant, i.e., no new clauses are added to $\mathcal{K}$)

Disadvantage: input resolution is no longer complete

Ex 355 Consider the clauses from Ex 352:

$\{p, q\}$ $\{\neg p, q\}$ $\{p, \neg q\}$ $\{\neg p, \neg q\}$

By input resolution we can deduce in the first step:

$\{q\}$, $\{\neg q\}$, $\{p\}$, $\{\neg p\}$, $\{p, \neg p\}$, or $\{q, \neg q\}$

In the second step, we obtain a clause from $\{q\}$, $\{\neg q\}$, $\{p\}$, $\{\neg p\}$ or from the input set.

$\Rightarrow$ By input resolution one can only deduce
- unit clauses ($\{q\}, \{\neg q\}, \{p\}, \{\neg p\}$)
- clauses from the input set
- tautologies ($\{p, \neg p\}, \{q, \neg q\}$)

One never reaches $\Box$.

While input resolution is not complete in general (not even in propositional logic), it is complete on Horn clauses (In LP, we only regard Horn clauses.)

<u>Def 356</u> (Horn clause)

A clause K is a <u>Horn clause</u> iff it contains at most one positive literal (all other literals must be negated atomic formulas).

A Horn clause is called <u>negative</u> iff it only contains negated literals (i.e., it has the form $\{\neg A_1, \dots, \neg A_k\}$ for atomic formulas $A_1, \dots, A_k$)).

A Horn clause is called <u>definite</u> iff it contains one positive literal (i.e, it has the form $\{B, \neg C_1, \dots, \neg C_n\}$ for atomic formulas $B, C_1, \dots, C_n$).

A set of definite Horn clauses corresponds to a conjunction of implications:

$$\{\ \{p, \neg q\},\ \{\neg r, \neg p, s\},\ \{s\}\ \}$$

is equivalent to

$$(p \vee \neg q) \wedge (\neg r \vee \neg p \vee s) \wedge s$$

which is equivalent to

$$(q \to p) \wedge (r \wedge p \to s) \wedge s.$$

This corresponds to the following logic program:

s.

s :- r, p.

p :- q.

← Restriction to Horn clauses means that we may not have negations in bodies of rules (p :- ¬q would correspond to $\{p, q\}$).

⇒ definite Horn clauses correspond to clauses of a logic program

- <u>facts</u> $\hat{=}$ definite Horn clauses without negative literals (e.g. $\{s\}$)
- <u>rules</u> $\hat{=}$ definite Horn clause with negative literals (e.g., $\{s, \neg r, \neg p\}$)
- <u>queries</u> $\hat{=}$ negative Horn clause (e.g., $\{\neg p, \neg q\}$).

?- p, q.

The negation of $p \wedge q$ would be added to the program clauses in order to prove unsatisfiability.

$\neg p \vee \neg q$, i.e., $\{\neg p, \neg q\}$

Restriction to Horn clauses improves efficiency substantially:

- input resolution instead of just linear resolution

(but unsatisfiability of Horn clauses remains undecidable in predicate logic)

- in propositional logic
  - (un)satisfiability of clauses is decidable, but NP-complete
  - (un)satisfiability of prop. Horn clauses can be checked in polynomial time

Instead of proving completeness of input resolution on Horn clauses, we restrict input resolution to SLD-resolution and then prove its completeness on Horn clauses.

Def 357 (SLD-Resolution)

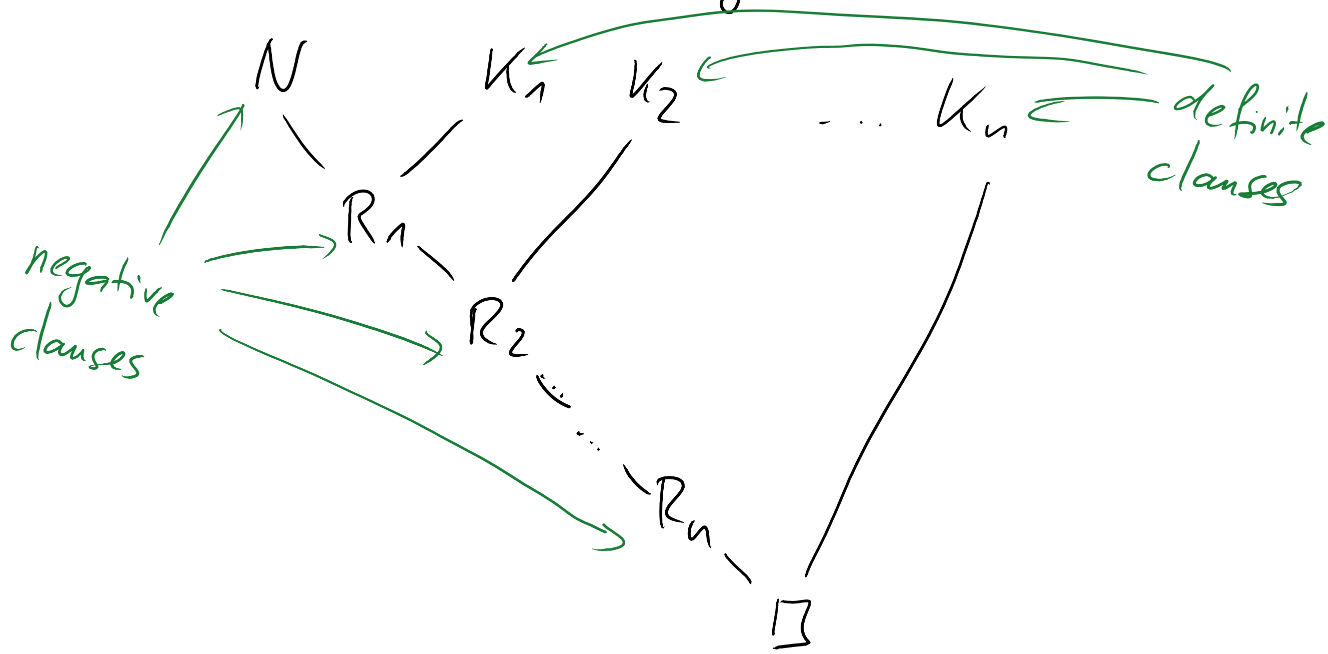
Let $\mathcal{K}$ be a set of Horn clauses with $\mathcal{K} = \mathcal{K}^d \uplus \mathcal{K}^n$, where $\mathcal{K}^d$ contains the definite and $\mathcal{K}^n$ contains the negative clauses of $\mathcal{K}$. $\Box$ can be derived from $K \in \mathcal{K}^n$ by SLD-resolution iff there is a sequence $K_1, \dots, K_m$ with

- $K_1 = K \in \mathcal{K}^n$
- $K_m = \Box$
- for each $2 \leq i \leq m$: K_i is a resolvent of K_{i-1} and a clause from $\mathcal{K}^d$.

Difference to input resolution:

• Start with negative clause K_1 (i.e., no resolution between 2 definite clauses)
⇒ negative clauses can only be resolved with definite clauses

⇒ K_2 is again a negative clause
⇒ … ⇒ all $K_1, \dots, K_n$ are negative clauses



"SLD-resolution" stands for

<u>l</u>inear resolution with <u>s</u>election function for <u>d</u>efinite clauses

Selection function needs to solve the remaining 2 indeterminisms:

1. Which program clause should be used in the next resolution step?
2. Which literal in the negative clause should be used for the next resolution step?

<u>Thm 358</u> (Soundness + Completeness of SLD-Resolution)

Let $\mathcal{K}$ be a set of Horn clauses. Then:

$\mathcal{K}$ is unsatisfiable iff $\square$ can be derived by SLD-resolution from some negative clause $N \in \mathcal{K}$.

Proof: "$\Leftarrow$" (Soundness) obvious, since SLD-resolution is a restriction of full resolution

"$\Rightarrow$" (Completeness):

Let $\mathcal{K}_{min} \subseteq \mathcal{K}$ be a minimal unsatisfiable subset of $\mathcal{K}$. $\mathcal{K}_{min}$ must contain a negative clause N, since any set of definite Horn clauses is satisfiable (the interpretation that satisfies all atomic formulas would be a model).

By Thm 3.5.3, $\square$ can be deduced by linear resolution from any clause in $\mathcal{K}_{min}$.

$\Rightarrow$ There is a linear resolution proof of $\square$ that starts with the negative clause $N \in \mathcal{K}_{min}$.

Any such linear resolution proof is also an SLD-resolution proof (since negative clauses can only be resolved with definite clauses and the resolvent is again a negative clause).

$\boxtimes$

Resolution Algorithm can now be improved by starting with a negative clause and by only performing SLD-resolution.

In logic programming, a resolution step only removes <u>one</u> literal in each parent clause (<u>binary</u> resolution).

<u>Binary resolution</u>: like ordinary resolution, but with $m = n = 1$. (i.e., in $\nu_1(K_1)$ one removes just L_1 and in $\nu_2(K_2)$ one removes just L_1').

In general, binary resolution is not complete.

<u>Ex. 359</u>

$$\{\overset{L_1}{p(X)}, \overset{L_2}{p(Y)}\} \qquad \{\overset{L_1'}{\neg p(U)}, \overset{L_2'}{\neg p(V)}\}$$

σ

$\square$

$\sigma = mgu\ \{\neg p(X), \neg p(Y), \neg p(U), \neg p(V)\} = \{X/V, Y/V, U/V\}$

This was not a binary resolution step.

$$\{\underline{p(X)}, p(Y)\} \qquad \{\underline{\neg p(U)}, \neg p(V)\}$$

not a
Horn clause

$\{p(Y), \neg p(V)\}$

$\square$ can't be derived with binary resolution.

But: binary resolution is complete for Horn clauses

Thm 3.5.10. (Soundness + Completeness of Binary SLD-Resolution)

Let $\mathcal{K}$ be a set of Horn clauses. Then:

$\mathcal{K}$ is unsatisfiable iff $\square$ can be deduced from a negative clause $N \in \mathcal{K}$ by binary SLD-resolution.

4. Logic Programs

4.1. Syntax and Semantics of Logic Programs

Horn clauses $\hat{=}$ clauses in logic programs

But in logic programming, the order of literals in a clause and of program clauses in a program plays a role.

Therefore, from now on:

clause = <u>sequence</u> of literals (literals can also occur repeatedly in a clause, order is important)

program/clause set = <u>sequence</u> of clauses

<u>Def 4.1.1</u> (Syntax of Logic Programs)

A non-empty finite set $\mathcal{P}$ of definite Horn clauses over a signature (Σ, Δ) is called a <u>logic program</u> over (Σ, Δ). The clauses in $\mathcal{P}$ are called <u>program clauses</u> and we distinguish the following forms of clauses:

- <u>facts</u>: clauses of the form $\{B\}$ where B is an atomic formula

- <u>rules</u>: clauses of the form $\{B, \neg C_1, \ldots, \neg C_n\}$ with $n \geq 1$ for atomic formulas $B, C_1, \ldots, C_n$.

A logic program is executed by submitting a

- <u>query</u> G of the form $\{\neg A_1, \ldots, \neg A_k\}$ with $k \geq 1$ where $A_1, \ldots, A_k$ are atomic formulas.

As usual: clause stands for universally quantified disjunction of its literals.

Calling a LP $\mathcal{P}$ with query $G = \{\neg A_1, \ldots, \neg A_k\}$ means that one wants to prove:

$$\mathcal{P} \models \exists \underbrace{X_1, \ldots, X_p}_{\text{Variables in } A_1, \ldots, A_k} . A_1 \wedge \ldots \wedge A_k$$

This is equivalent to unsatisfiability of

$\mathcal{P} \cup \{G\}$, i.e., to the unsatisfiability of

$$\mathcal{P} \cup \{\forall X_1, \ldots, X_p \; \neg A_1 \vee \ldots \vee \neg A_k\}$$

By Thm 3.3.9 (a) (Herbrand-Expansion) and compactness of prop. resolution: Equivalent to

There is a finite set of ground instantiations of $\mathcal{P} \cup \{\forall X_1, \ldots, X_p \; \neg A_1 \vee \ldots \vee \neg A_k\}$ that is unsatisfiable.

By completeness of SLD-resolution:

There are ground terms $t_1, \dots, t_p$ such that
$$\mathcal{P} \cup \{ (\neg A_1 \vee \dots \vee \neg A_k)[X_1/t_1, \dots, X_p/t_p] \}$$
is unsatisfiable.

Goal: Find those instantiations $t_1, \dots, t_p$ where
$$\mathcal{P} \cup \{ (\neg A_1 \vee \dots \vee \neg A_k)[X_1/t_1, \dots, X_p/t_p] \}$$ is unsatisfiable

resp.

where $\mathcal{P} \models A_1 \wedge \dots \wedge A_k \, [X_1/t_1, \dots, X_p/t_p]$

(i.e., we also want to know the answer substitutions)

Answer substitutions are constructed during the SLD-resolution proof.

Ex 4.12 Consider the LP:

motherOf(renate, susanne).

married(gerd, renate).

fatherOf(F, C) :- married(F, W), motherOf(W, C).

?- fatherOf(gerd, Y).

Goal: for which instantiations t is
$$\mathcal{P} \cup \{ \neg fatherOf(gerd, Y) [Y/t] \}$$ unsatisfiable?

To find this out: SLD-resolution on $\mathcal{P} \cup \{G\}$.

Answer substitution: Compose all used mgu's and restrict them to the variables occurring in the initial query.

Here: $\{Y / susanne\}$.

We have defined the syntax of LP.

Now: define the semantics of LP.

3 different (but equivalent) possibilities:

4.1.1. declarative semantics

4.1.2. procedural (or operational) semantics

4.1.3. fixpoint (or denotational) semantics

4.1.1. Declarative Semantics of Logic Prog.

Idea: use the semantics of predicate logic

All ground instances of a query G are "true" in a logic prog. $\mathcal{S}$ where $\mathcal{S}$ entails the instance in G

entailment $\models$ in pred. logic, defined via interpretations

Def 4.1.3 (Declarative Semantics of a LP)

Let $\mathcal{S}$ be a LP and $G = \{\neg A_1, .., \neg A_k\}$ be a query. Then the declarative semantics of $\mathcal{S}$ w.r.t. G is defined as:

$$D[\![\mathcal{S}, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \mathcal{S} \models \sigma(A_1 \wedge \ldots \wedge A_k), \ \sigma \text{ is a ground substitution}\}$$

Ex. 4.1.4

$D[\![\mathcal{P}, G]\!] = \{ fatherOf(gerd, susanne) \}$

If $\mathcal{P}$ also contained the fact motherOf(renate, peter), then

$D[\![\mathcal{P}, G]\!] = \{ fatherOf(gerd, susanne), fatherOf(gerd, peter) \}$.

4.1.2. Procedural Semantics of LP

Idea: provide an example-interpreter which does the "right" thing. In this way, one can define the meanings of programs.

Solution: perform SLD-resolution and collect the used mgu's to obtain the answer subst. in the end.

- operate on <u>configurations</u> (pairs of negative clause and substitution)
- start with $(G, \emptyset)$ ← empty/identical substitution

goal is to reach $(\square, \sigma)$.

Then the restriction of σ to the variables in G is the answer substitution.

- <u>Computation</u>: sequence of configurations where the step from one config. to the next is done by SLD-resolution.
- 3 modifications of SLD-resolution:
  - standardized SLD-resolution: only rename variables in prog. clauses, not in negative clauses
  - binary resolution: only resolve one literal in each clause in each resolution step

- clauses are regarded as <u>sequences</u> of literals (instead of <u>sets</u>). Thus: a literal can occur multiple times in a clause

<u>Def 4.15</u> (Procedural Semantics of LP)

Let $\mathcal{P}$ be a LP.

- A <u>configuration</u> is a pair (G, σ) where G is a negative Horn clause (possibly $\Box$) and σ is a substitution.
- We have a <u>computation step</u> $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ iff
  - $G_1 = \{\neg A_1, \dots, \neg A_k\}$ with $k \geq 1$
  - there is a program clause $K \in \mathcal{P}$ and a variable renaming ν with $\nu(K) = \{B, \neg C_1, \dots, \neg C_n\}$ and $n \geq 0$ such that
    - $\nu(K)$ has no common variables with G_1 or $RANGE(\sigma_1)$

      $\{\sigma_1(X) \mid X \in DOM(\sigma_1)\}$

    - there is an $1 \leq i \leq k$ such that A_i and B are unifiable with a mgu σ
  - $G_2 = \sigma(\{\neg A_1, \dots, \neg A_{i-1}, \neg C_1, \dots, \neg C_n, \neg A_{i+1}, \dots, \neg A_k\})$
  - $\sigma_2 = \sigma \circ \sigma_1$
- A <u>computation</u> of $\mathcal{P}$ with the query G is a (finite or infinite) sequence of configurations:

  $(G, \emptyset) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \dots$
- A computation $(G, \emptyset) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\Box, \sigma)$ is called <u>successful</u>. If $G = \{\neg A_1, \dots, \neg A_k\}$, then the <u>result</u> of the computation is $\sigma(A_1 \wedge \dots \wedge A_k)$.

The <u>answer substitution</u> is σ, restricted to the variables in G.

Now we can define the <u>procedural semantics</u> of $\mathcal{P}$ w.r.t. $G = \{\neg A_1, \ldots, \neg A_k\}$:

$$P[\![\mathcal{P}, G]\!] = \{\ \sigma'(A_1 \wedge \ldots \wedge A_k) \mid (G, \emptyset) \vdash_{\mathcal{P}}^{+} (\square, \sigma),\ \sigma'(A_1 \wedge \ldots \wedge A_k) \text{ is a ground instance of } \sigma(A_1 \wedge \ldots \wedge A_k)\ \}$$

"+" means transitive closure, i.e. $(G,\emptyset) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square,\sigma)$

<u>Ex. 4.16</u> $\mathcal{P}$, G as in Ex. 4.12

$(\{\neg fatherOf(gerd, Y)\}, \emptyset)$

$\vdash_{\mathcal{P}} (\{\neg married(gerd, W), \neg motherOf(W, C)\}, \{Y/C, F/gerd\})$

$\vdash_{\mathcal{P}} (\{\neg motherOf(renate, C)\}, \{W/renate, Y/C, F/gerd\})$

$\vdash_{\mathcal{P}} (\square, \{C/susanne, W/renate, Y/susanne, F/gerd\})$

Answer Subst: $\{Y/susanne\}$

Proc. Semantics has 2 indeterminisms:

1. choice of prog. clause K for the next resolution step
2. choice of literal $\neg A_i$ in the current goal for the next res. step.

Choices can influence success, length, result of computation:

<u>Ex 4.17</u> $\mathcal{P} = \{\ \{p(X,Z), \neg q(X,Y), \neg p(Y,Z)\},$
$\{p(U,U)\},$
$\{q(a,b)\}\ \}$

Query $G = \{\neg p(V,b)\}$

$(\{\neg \underline{p(V,b)}\}, \emptyset)$

$\vdash_{\mathcal{P}} (\{\neg \underline{q(V,Y)}, \neg p(Y,b)\}, \{X/V, Z/b\})$ Res. with first prog. cl.

$\vdash_{\mathcal{P}} (\{\neg \underline{p(b,b)}\}, \{V/a, Y/b, X/a, Z/b\})$ – Res. with first pr. cl.

$\vdash_{\mathcal{P}} (\{\neg q(b,Y'), \neg \underline{p(Y',b)}\}, \{X'/b, Z'/b, V/a, Y/b, X/a, Z/b\})$

$\vdash_{\mathcal{P}} (\{\neg q(b,b)\}, \{U/b, Y'/b, \dots\})$

finite failing computation (doesn't end in $\Box$).

If after the first 2 computation steps one would have used the 2nd prog clause, one would have reached

$(\Box, \{U/b, V/a, \dots\})$

Answer Subst: $\{V/a\}$. $\rightsquigarrow$ $p(a,b) \in P[\![\mathcal{P}, G]\!]$.

Moreover, one could have used the 2nd prog. clause in the first res step:

$(\{\neg p(V,b)\}, \emptyset)$

$\vdash_{\mathcal{P}} (\Box, \{U/b, V/b\})$.

Answer subst: $\{V/b\}$ $\rightsquigarrow$ $p(b,b) \in P[\![\mathcal{P}, G]\!]$.

<u>Thm 4.18</u> (Equivalence of declarative and procedural semantics)

Let $\mathcal{P}$ be a LP and G be a query.

Then $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$.

<u>Proof</u>: Based on soundness + completeness of

SLD-resolution. Moreover, one has to keep track of the substitutions. $\boxtimes$

4.1.3. Fixpoint Semantics of LP

Idea: • only regard the program $\mathcal{P}$

• in each step, extend the facts of $\mathcal{P}$ by those statements that can be inferred by one more application of a rule from $\mathcal{P}$.

Formally: use a function $trans_{\mathcal{P}}(M)$.

$p(t_1, \dots, t_n)$

set of atomic ground formulas.

It returns M extended by those ground atomic formulas that can be deduced from M by one application of a rule from $\mathcal{P}$.

Then: Set of all true statements about $\mathcal{P}$:

$$\emptyset \cup trans_{\mathcal{P}}(\emptyset) \cup \underbrace{trans_{\mathcal{P}}(trans_{\mathcal{P}}(\emptyset))}_{trans^2_{\mathcal{P}}(\emptyset)} \cup trans^3_{\mathcal{P}}(\emptyset) \cup \dots$$

Def 4.1.9. (Transformation $trans_{\mathcal{P}}$)

Let $\mathcal{P}$ be a LP over a signature (Σ, Δ).

Then $trans_{\mathcal{P}}$ is a function $trans_{\mathcal{P}}: \underbrace{Pot(\mathcal{A}t(\Sigma, \Delta, \emptyset))}_{\text{Set containing all subsets of } \mathcal{A}t(\Sigma,\Delta,\emptyset)} \to$

$Pot(\mathcal{A}t(\Sigma, \Delta, \emptyset))$

with

$trans_{\mathcal{P}}(M) = M \cup \{A' \mid \{A', \neg B_1', \dots, \neg B_n'\}$ is a ground instance

of a clause $\{A, \neg B_1, \dots, \neg B_n\} \in \mathcal{P}$
and $B_1', \dots, B_n' \in M\}$

Ex 4.1.10

$trans_{\mathcal{P}}^{0}(\emptyset) = \emptyset$

$trans_{\mathcal{P}}^{1}(\emptyset) = \{motherOf(ren, sus), married(gerd, ren)\}$

$trans_{\mathcal{P}}^{2}(\emptyset) = \{$ —— " ——————,
fatherOf(gerd, renate)$\}$

$trans_{\mathcal{P}}^{3}(\emptyset) = trans_{\mathcal{P}}^{2}(\emptyset)$

Ex 4.1.11 In general, the iteration of applying $trans_{\mathcal{P}}$ repeatedly can go on infinitely long.

p(a).
p(f(X)) :- p(X).

$trans_{\mathcal{P}}(\emptyset) = \{p(a)\}$

$trans_{\mathcal{P}}^{2}(\emptyset) = \{p(a), p(f(a))\}$

$trans_{\mathcal{P}}^{3}(\emptyset) = \{p(a), p(f(a)), p(f(f(a)))\}$

$\vdots$

$$\underbrace{\bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset)}_{} = \{p(f^{i}(a)) \mid i \in \mathbb{N}\}$$

We call this set $M_{\mathcal{P}}$.

We use $M_P = \bigcup_{i \in \mathbb{N}} trans_P^i(\emptyset)$ to define the semantics of P.

- M_P is a fixpoint of $trans_P$: $trans_P(M_P) = M_P$

  This means: M_P already contains all true statements about P.

- M_P is the least fixpoint of $trans_P$: for all other fixpoints M of $trans_P$, we have $M_P \subseteq M$

  least: smallest w.r.t. $\subseteq$

  This means: M_P only contains those statements that are enforced by P (i.e., that are really true in P).

Now: Prove formally that

$$M_P = \bigcup_{i \in \mathbb{N}} trans_P^i(\emptyset)$$

is the least fixpoint of $trans_P$. (A similar construction can be used to define the semantics of other prog. languages.)

A. Properties of $\subseteq$

- reflexive $\quad M_1 \subseteq M_1$
- transitive $\quad M_1 \subseteq M_2$ and $M_2 \subseteq M_3$ implies $M_1 \subseteq M_3$
- antisymmetric $\quad M_1 \subseteq M_2$ and $M_2 \subseteq M_1$ implies $M_1 = M_2$

— "ordering"

Moreover, $\subseteq$ is a complete reflexive ordering.

- $\subseteq$ must have a smallest element: $\emptyset$
- every chain has a least upper bound, i.e.:

Whenever there are sets $M_0, M_1, \ldots$ with

$M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$ (a so-called chain)

then there exists a least upper bound (lub) M.

This means: $M_i \subseteq M$ for all $i \in \mathbb{N}$

and for all other upper bounds M', we have

$M \subseteq M'$.

Solution: lub of $M_0, M_1, \ldots$ is

$$\bigcup_{i \in \mathbb{N}} M_i.$$

Reason: $\bigcup_{i \in \mathbb{N}} M_i$ is an upper bound of $M_0, M_1, \ldots$

because $M_i \subseteq \bigcup_{i \in \mathbb{N}} M_i$.

It is the lub: If there were another upper bound M' of $M_0, M_1, \ldots$,

then $M_0 \subseteq M'$, $M_1 \subseteq M'$, ...

$\curvearrowright \bigcup_{i \in \mathbb{N}} M_i \subseteq M'$.

Lemma 4.1.12 The subterm relation $\subseteq$ on $Pot(At(\Sigma, \Delta, \emptyset))$ is a complete reflexive order.

Proof: See above

B. Properties of $trans_\varrho$

$trans_\varrho$ has 2 important properties:

- $trans_\varrho$ is monotonic: $M_1 \subseteq M_2$ implies
$$trans_\varrho(M_1) \subseteq trans_\varrho(M_2)$$
- $trans_\varrho$ is continuous (stetig):

$$\begin{tikzcd}
M_0 \arrow[d] & \subseteq & M_1 \arrow[d] & \subseteq & \dots & \xrightarrow{lub} & M \arrow[d] \\
trans_\varrho(M_0) & \subseteq & trans_\varrho(M_1) & \subseteq & \dots & \xrightarrow{lub} & trans_\varrho(M)
\end{tikzcd}$$

Continuity means: the black and the green step yield the same solution

Lemma 4.1.13 (Monotonicity and Continuity of $trans_\varrho$)

(a) $trans_\varrho$ is monotonic, i.e., if $M_1 \subseteq M_2$ then $trans_\varrho(M_1) \subseteq trans_\varrho(M_2)$.

(b) $trans_\varrho$ is continuous, i.e.,
for every chain $M_0 \subseteq M_1 \subseteq M_2 \subseteq \dots$
we have $trans_\varrho\left(\bigcup_{i \in \mathbb{N}} M_i\right) = \bigcup_{i \in \mathbb{N}} trans_\varrho(M_i)$.

Proof: (a) follows immediately from the definition of $trans_\varrho$.
We now show (b).

First, show $trans_\rho(\bigcup_{i\in\mathbb{N}} M_i) \supseteq \bigcup_{i\in\mathbb{N}} trans_\rho(M_i)$.

This follows from monotonicity of $trans_\rho$:

$$\bigcup_{i\in\mathbb{N}} M_i \supseteq M_i$$

$$\rightsquigarrow trans_\rho(\bigcup_{i\in\mathbb{N}} M_i) \supseteq trans_\rho(M_i) \text{ for all } i\in\mathbb{N}$$

$$\rightsquigarrow trans_\rho(\bigcup_{i\in\mathbb{N}} M_i) \supseteq \bigcup_{i\in\mathbb{N}} trans_\rho(M_i)$$

Now we show $trans_\rho(\bigcup_{i\in\mathbb{N}} M_i) \subseteq \bigcup_{i\in\mathbb{N}} trans_\rho(M_i)$.

Let $A' \in trans_\rho(\bigcup_{i\in\mathbb{N}} M_i)$.

Then $\{A', \neg B_1', \dots, \neg B_n'\}$ is a ground instance of a clause $\{A, \neg B_1, \dots, \neg B_n\} \in \rho$ and $B_1', \dots, B_n' \in \bigcup_{i\in\mathbb{N}} M_i$.

Since $M_0 \subseteq M_1 \subseteq \dots$, there exists a $j\in\mathbb{N}$ such that $B_1', \dots, B_n' \in M_j$.

$$\rightsquigarrow A' \in trans_\rho(M_j) \subseteq \bigcup_{i\in\mathbb{N}} trans_\rho(M_i). \quad \boxed{\text{q.e.d.}}$$

Now we can show that M_ρ is indeed the least fixpoint of $trans_\rho$. (This theorem holds in general: every continuous function f over a complete ordering has a least fixpoint, which is the lub of the chain $\emptyset, f(\emptyset), f^2(\emptyset), \dots$. Here, $\emptyset$ is the smallest element of the ordering.)

Thm 4.1.14 (Fixpoint Theorem, Kleene+Tarski)

For every LP $\mathcal{P}$, the function $trans_{\mathcal{P}}$ has a least fixpoint $lfp(trans_{\mathcal{P}})$. Here:

$$lfp(trans_{\mathcal{P}}) = \bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset).$$

Proof: 1. $\bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset)$ is a fixpoint of $trans_{\mathcal{P}}$.

$$trans_{\mathcal{P}}\Big(\bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset)\Big)$$

$$= \bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i+1}(\emptyset) \qquad \text{(since } trans_{\mathcal{P}} \text{ is continuous)}$$

$$= \emptyset \cup \bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i+1}(\emptyset)$$

$$= \bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset).$$

2. $\bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset)$ is smaller or equal to any other fixpoint M of $trans_{\mathcal{P}}$.

Let M be another fixpoint of $trans_{\mathcal{P}}$.

We want to show: $\bigcup_{i \in \mathbb{N}} trans_{\mathcal{P}}^{i}(\emptyset) \subseteq M$.

It suffices to show: $trans_{\mathcal{P}}^{i}(\emptyset) \subseteq M$ for all $i \in \mathbb{N}$.

Prove this by induction on i.

Ind Base: $i = 0$

—

$trans^0_P(\emptyset) = \emptyset \subseteq M$ ✓

Ind Step: $i > 0$

Ind Hypothesis: $trans^{i-1}_P(\emptyset) \subseteq M$

By monotonicity of $trans_P$: $trans^i_P(\emptyset) \subseteq trans_P(M) = M$

because M is a fixpoint of $trans_P$. ☒

Finally, we can define the fixpoint semantics of LP:

Def 4.1.15 (Fixpoint Semantics of LP)

Let P be a LP, let $G = \{\neg A_1, \ldots, \neg A_k\}$ be a query. Then the fixpoint semantics of P w.r.t. G is defined as:

$$F[\![P, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \sigma(A_i) \in lfp(trans_P) \text{ for all } 1 \le i \le k\}.$$

Thm 4.1.16 (Equivalence of all 3 semantics-definitions)

Let P be a LP, G be a query.

Then $D[\![P, G]\!] = P[\![P, G]\!] = F[\![P, G]\!]$.

Proof: see course notes.

4.2 Universality of Logic Programming

Freitag, 22. Mai 2015 08:30

Goal: Show that LP is a Turing-complete language

for every computable function, there is a LP that computes it

⇝ LP is as powerful as C, Java, Haskell, ...

Defining computable functions (1930s):

- Turing: Turing machines
- Church: Lambda Calculus
- Kleene: μ-recursive functions

the set of computable functions is always the same

⇒ Church's thesis:

no prog. language can compute more functions than those expressible by Turing machines, λ-calculus, μ-recursion

Thus: to prove that LP is Turing-complete, show that for every μ-recursive function, there is a LP computing it.

All algebraic data structures (lists, trees, ...) can be encoded as natural numbers $\Rightarrow$ only regard algorithms on natural numbers.

Def 4.2.1. (μ-recursive functions)

The set of μ-recursive functions is the smallest set of functions such that:

1. For every $n \in \mathbb{N}$, the function $null_n : \mathbb{N}^n \to \mathbb{N}$ with $null_n(k_1, \dots, k_n) = 0$ is μ-recursive.
2. The successor function $succ : \mathbb{N} \to \mathbb{N}$ with $succ(k) = k+1$ is μ-recursive.
3. For every $n \geq 1$ and every $1 \leq i \leq n$, the projection function $proj_{n,i}(k_1, \dots, k_n) = k_i$ is μ-recursive.
4. μ-recursive functions are closed under composition: For all $m \geq 1$ and $n \geq 0$ we have: if $f : \mathbb{N}^m \to \mathbb{N}$ and $f_1, \dots, f_m : \mathbb{N}^n \to \mathbb{N}$ are μ-recursive, then the following fct. $g : \mathbb{N}^n \to \mathbb{N}$ is also μ-recursive:

$$g(k_1, \dots, k_n) = f\Big(f_1(k_1, \dots, k_n), \dots, f_m(k_1, \dots, k_n)\Big)$$

5. The μ-recursive functions are closed under <u>primitive recursion</u>: For all $n \geq 0$ we have:
if $f: \mathbb{N}^n \to \mathbb{N}$ and $g: \mathbb{N}^{n+2} \to \mathbb{N}$ are μ-recursive, then the following fct $h: \mathbb{N}^{n+1} \to \mathbb{N}$ is also μ-recursive:

$$h(k_1, \ldots, k_n, 0) = f(k_1, \ldots, k_n)$$

$$h(k_1, \ldots, k_n, k+1) = g(k_1, \ldots, k_n, k, h(k_1, \ldots, k_n, k))$$

Functions that can be expressed with principles 1-5 are called <u>primitive recursive</u>.

There exist computable functions that are not primitive recursive:

- partial functions (implemented by programs that do not always terminate)
- certain total functions (e.g., the Ackermann function), but almost all total computable functions used in practice are primitive recursive.

6. μ-recursive functions are closed under unbounded minimization: For all $n \geq 0$ we have:
if $f: \mathbb{N}^{n+1} \to \mathbb{N}$ is μ-recursive, then the following fct $g: \mathbb{N}^n \to \mathbb{N}$ is also μ-recursive:

$g(k_1,\dots,k_n)=k$ iff $f(k_1,\dots,k_n,k)=0$
and for all $0\le k'<k$,
$f(k_1,\dots,k_n,k')$ is defined and
$f(k_1,\dots,k_n,k')>0$.

If there is no such k, then $g(k_1,\dots,k_n)$ is undefined.

Now we will show that every μ-recursive fct can be computed by a LP.

Ex 4.22 Consider some well-known computable fcts on $\mathbb{N}$ and show that they are μ-recursive.

- plus: $\mathbb{N}^2\to\mathbb{N}$ is μ-recursive, even primitive recursive

$$plus(x,0)=proj_{1,1}(x)$$
$$plus(x,y+1)=\underbrace{f(x,y,plus(x,y))}_{plus(x,y)+1}$$

$$f(x,y,z)=succ(proj_{3,3}(x,y,z))$$

- times: $\mathbb{N}^2\to\mathbb{N}$ is also primitive recursive

$$times(x,0)=null_1(x)$$

$$times(x,y+1)=\underbrace{g(x,y,times(x,y))}_{times(x,y)+x}$$

$$g(x,y,z) = plus\,(proj_{3,1}(x,y,z),\ proj_{3,3}(x,y,z))$$

- The predecessor function is also primitive recursive:

  $p: \mathbb{N} \to \mathbb{N}$ with $p(0) = 0$, $p(x+1) = x$

$$p(0) = null_0$$
$$p(x+1) = proj_{2,1}(x,\ p(x))$$

- The funct. minus: $\mathbb{N}^2 \to \mathbb{N}$ is also prim. recursive, where $minus(x,y) = 0$ if $x \leq y$ and $minus(x,y) = x-y$ otherwise.

$$minus(x,0) = proj_{1,1}(x)$$
$$minus(x,y+1) = \underbrace{h(x,y,minus(x,y))}_{p(minus(x,y))}$$

$$h(x,y,z) = p\,(proj_{3,3}(x,y,z))$$

- div: $\mathbb{N}^2 \to \mathbb{N}$ is also μ-recursive, where

$$div(x,y) = \left\lceil \frac{x}{y} \right\rceil \text{ if } y \neq 0$$
$$div(0,0) = 0$$
$$div(x,0) \text{ is undefined if } x \neq 0$$

Idea: $div(x,y) = z$ iff $\frac{x}{y} = z$

iff $x = y \cdot z$

iff $X - Y \cdot Z = 0$

$\Rightarrow$ use a function $i(X,Y,Z) = X - Y \cdot Z$ and search for the smallest Z where $i(X,Y,Z) = 0$.

$div(X,Y) = Z$ iff $i(X,Y,Z) = 0$ and forall $0 \leq Z' < Z$, $i(X,Y,Z')$ is defined and $i(X,Y,Z') > 0$

where $i(X,Y,Z)$ computes $X - Y \cdot Z$. This function i is primitive recursive:

$$i(X,Y,Z) = minus(proj_{3,1}(X,Y,Z), j(X,Y,Z))$$
$$j(X,Y,Z) = times(proj_{3,2}(X,\underline{Y},Z), proj_{3,3}(X,Y,\underline{Z}))$$

How can a LP "compute" an arithmetic function?

- A LP only "evaluates" predicate symbols, not function symbols.
  Solution: to compute a function $f: \mathbb{N}^n \to \mathbb{N}$, use a predicate symbol $\underline{f}$ of arity $n+1$ where $\underline{f}(K_1, \ldots, K_n, K)$ is true iff $f(K_1, \ldots, K_n) = K$.
- LPs operate on terms, not on natural numbers.

Solution: represent natural numbers by terms using $0 \in \Sigma_0$ and $s \in \Sigma_1$.

Then the term 0 represents the number 0,

$s(0)$ — 1

$s(s(0))$ — 2

$\vdots$

Def 423 (Computing arithmetic functions with logic programs)

- Every $k \in \mathbb{N}$ is represented by the term $\underline{k} \in \mathcal{T}(\Sigma, \mathcal{V})$ where $\underline{k} = s^k(0)$, where $0 \in \Sigma_0$, $s \in \Sigma_1$
- A LP $\mathcal{S}$ over (Σ, Δ) computes an arithmetic fct $f: \mathbb{N}^n \to \mathbb{N}$ iff there is a pred. symbol $\underline{f} \in \Delta_{n+1}$ such that

$$f(k_1, \dots, k_n) = k \quad \text{iff} \quad \mathcal{S} \models \underline{f}(\underline{k_1}, \dots, \underline{k_n}, \underline{k}).$$

Reason: To compute $f(k_1, \dots, k_n)$, one can then ask the query $?- \underline{f}(\underline{k_1}, \dots, \underline{k_n}, X)$.

Ex. 424 The example functions in Ex. 422 can all be computed by a LP:

$\underline{plus}(X, 0, X).$

$\underline{plus}(X, s(Y), s(Z)) \text{ :- } \underline{plus}(X, Y, Z).$

$\vdots$

Thm 4.25 (Universality of LP)

Every μ-recursive fct. can be computed by a LP.

Proof: Induction according to the construction principle for μ-recursive fcts.

1. $\underline{null}_n(X_1, \dots, X_n, 0).$
2. $\underline{succ}(X, s(X)).$
3. $\underline{proj}_{n,i}(X_1, \dots, X_n, X_i).$
4. By ind. hypothesis, there are predicates $\underline{f}, \underline{f_1}, \dots, \underline{f_m}$ that compute $f, f_1, \dots, f_m$.

$$\underline{g}(X_1, \dots, X_n, Z) \text{ :- } \underline{f_1}(X_1, \dots, X_n, Y_1), \dots, \underline{f_m}(X_1, \dots, X_n, Y_m), \underline{f}(Y_1, \dots, Y_m, Z).$$

5. By ind. hyp., there are predicates $\underline{f}$ and $\underline{g}$:

$$\underline{h}(X_1, \dots, X_n, 0, Z) \text{ :- } \underline{f}(X_1, \dots, X_n, Z).$$

$$\underline{h}(X_1,\dots,X_n,s(X),Z) :- \underline{h}(X_1,\dots,X_n,X,Y),$$
$$\underline{g}(X_1,\dots,X_n,X,Y,Z).$$

6. By ind. hyp, there is a pred $\underline{f}$.

We introduce an additional predicate $\underline{f}'$ such that

$\underline{f}'(X_1,\dots,X_n,Y,Z)$ is true iff

$f(X_1,\dots,X_n,Z) = 0$ and

$f(X_1,\dots,X_n,X) > 0$ for all X with $Y \leq X < Z$

$$\underline{g}(X_1,\dots,X_n,Z) :- \underline{f}'(X_1,\dots,X_n,0,Z).$$
$$\underline{f}'(X_1,\dots,X_n,Y,Y) :- \underline{f}(X_1,\dots,X_n,Y,0).$$
$$\underline{f}'(X_1,\dots,X_n,Y,Z) :- \underline{f}(X_1,\dots,X_n,Y,s(U)),$$
$$\underline{f}'(X_1,\dots,X_n,s(Y),Z).$$

$\Box$

Ex 426 The construction principle from the proof of Thm 425 could be directly used to convert μ-recursive functions to LPs.

$$\underline{plus}(X,0,U) :- \underline{proj_{1,1}}(X,U).$$

$\underline{plus}(X, s(Y), U) \text{ :- } \underline{plus}(X, Y, Z), \underline{f}(X, Y, Z, U).$

$\underline{f}(X, Y, Z, V) \text{ :- } \underline{proj_{3,3}}(X, Y, Z, U), \underline{succ}(U, V).$

$\underline{succ}(X, s(X)).$

$\underline{proj_{1,1}}(X, X).$

$\underline{proj_{3,3}}(X, Y, Z, Z).$

4.3 Indeterminism and Evaluation Strategies

Monday, 01 June, 2015 8:30

Procedural Semantics has 2 indeterminisms:

Indeterminism 1: Which program clause K is used for the next res. step?

Indeterminism 2: Which A_i in the current goal is used for the next resolution step?

$\Rightarrow$ For one configuration (G_1, σ_1) there can be <u>several</u> successor configurations with
$(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$.

<u>Ex. 4.3.1</u> Query: ?- ancestor(Z, aline).

$(\{\neg anc(Z, aline)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg mO(Z, aline)\}, \{V/Z, X/aline\})$

$(\{\neg anc(Z, aline)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg mO(Z, Y), \neg anc(Y, aline)\}, \{V/Z, X/aline\})$

Indet 1 influences the solution.

Indet 2 influences the termination

To implement LP (on a deterministic computer), one has to resolve these 2 indeterminisms.

We first look at indeterminism 2.

It will turn out that this indet. is "harmless":
it does not influence the solution, i.e., if one

resolves this indeterminism (e.g., by only taking the leftmost literal), then one still finds all solutions to the query.

Main reason: Exchangement Lemma:
For a query $\{\neg A_1, \dots, \neg A_k\}$, it does not matter whether one first resolves with A_i and then with A_j or vice versa.

Lemma 4.32 (Exchangement Lemma)
Let $\{\neg A_1, \dots, \neg A_k\}$, $\{B, \neg C_1, \dots, \neg C_n\}$, $\{D, \neg E_1, \dots, \neg E_m\}$ be variable-disjoint Horn clauses. Let σ_1 be the mgu of A_i and B, let σ_2 be the mgu of $\sigma_1(A_j)$ and D. Then the 2 resolution steps on the slide are possible (first resolve with $\neg A_i$, then with $\neg A_j$).
Then there exists an mgu σ_1' of A_j and D, and an mgu σ_2' of $\sigma_1'(A_i)$ and B. So it is also possible to resolve with A_j first and then with A_i.
Then $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are identical up to variable renaming (i.e., there is a variable renaming ν such that $\sigma_2' \circ \sigma_1' = \nu \circ \sigma_2 \circ \sigma_1$.

Ex 4.33 Illustration of the exchangement lemma:

p(Z,Z) :- r(Z).

q(W).

$?- p(X,Y), q(X).$ ← start resolving with the p-literal

$(\{\neg \underline{p(X,Y)}, \neg q(X)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg r(Z), \neg \underline{q(Z)}\}, \{X/Z, Y/Z\})$

$\vdash_{\mathcal{P}} (\{\neg r(Z)\}, \underbrace{\{W/Z\} \circ \{X/Z, Y/Z\}}_{\{X/Z,\, Y/Z,\, W/Z\}})$

Exchangement lemma states that one could exchange these 2 resolution steps (i.e., first resolve on q, then on p). Then we get the same substitution up to variable renaming.

$(\{\neg p(X,Y), \neg \underline{q(X)}\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg \underline{p(W,Y)}\}, \{X/W\})$

$\vdash_{\mathcal{P}} (\{\neg r(Y)\}, \underbrace{\{W/Y, Z/Y\} \circ \{X/W\}}_{\{X/Y,\, W/Y,\, Z/Y\}})$

The resulting substitutions can be made equal by applying the variable renaming $\nu = \{Y/Z, Z/Y\}$.

<u>Proof of the exchangement lemma 4.3.2:</u>

Since the clauses are variable-disjoint, the mgu σ_1 of A_i and B does not modify the variables in D, i.e. $\sigma_1(D) = D$.

σ_2 is the mgu of $\sigma_1(A_j)$ and $\underbrace{D}_{\sigma_1(D)}$

$\Rightarrow \sigma_2 \circ \sigma_1 (A_j) = \sigma_2 \circ \sigma_1 (D)$

$\Rightarrow \sigma_2 \circ \sigma_1$ is a unifier of A_j and D.

$\Rightarrow A_j$ and D have an mgu σ_1' and there exists a substitution σ such that

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'. \qquad (*)$$

So we can perform the first resolution step using the mgu σ_1'.
Now we have to show that one can also perform the second res. step, i.e., that $\sigma_1'(A_i)$ and $\mathcal{B}$ are unifiable.
This indeed holds, since σ is a unifier of $\sigma_1'(A_i)$ and $\mathcal{B}$:

$$\begin{aligned}
\sigma(\sigma_1'(A_i)) &= \sigma_2(\sigma_1(A_i)) && \text{by } (*) \\
&= \sigma_2(\sigma_1(\mathcal{B})) && \text{since } \sigma_1 \text{ unifies } A_i \text{ and } \mathcal{B} \\
&= \sigma(\sigma_1'(\mathcal{B})) && \text{by } (*) \\
&= \sigma(\mathcal{B}) && \text{since } \sigma_1' \text{ does not modify the variables of } \mathcal{B} \text{ (} \sigma_1' \text{ is mgu of } A_j \text{ and } \mathcal{D}\text{)}
\end{aligned}$$

Since σ is a unifier of $\sigma_1'(A_i)$ and $\mathcal{B}$, they also have an mgu σ_2'. Thus, there exists a substitution δ with

$$\sigma = \delta \circ \sigma_2'. \qquad (**)$$

Hence, one can exchange the resolution steps and first perform resolution on $\neg A_j$, then on $\neg A_i$.
We still have to show that $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are the same up to variable renaming.
To show this: Prove that $\sigma_2' \circ \sigma_1'$ is an instance of $\sigma_2 \circ \sigma_1$ and $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$.

$\sigma_2 \circ \sigma_1 = \delta \circ \sigma_2' \circ \sigma_1'$ holds.

Reason: $\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'$ by (*)
$= \delta \circ \sigma_2' \circ \sigma_1'$ by (**).

In a similar way, one can show that there also exists a subst. δ' with $\sigma_2' \circ \sigma_1' = \delta' \circ \sigma_2 \circ \sigma_1$. ☒

The exchangement lemma implies that one can impose an arbitrary ordering on literals in a clause and restrict ourselves to resolution steps with the "first" literal in the clause (w.r.t. the ordering).

⇒ Regard clauses as sequences of literals and use an arbitrary selection function to select some literal from the clause for the next resolution step.
(SLD ≙ selection fct.)

Prolog uses the selection fct. that always takes the leftmost literal. Computation steps that use the first literal in the goal are called canonical.

Def 434 (Canonical Computations)
A computation $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$ is called canonical iff each resolution step is done using the first literal of the respective goal G_i.

Thm 435 (Resolving Indeterminism 2)
Let $\mathcal{P}$ be a LP, let G be a query.
For every successful computation $(G, \emptyset) \vdash_{\mathcal{P}}^{+} (\Box, \sigma)$, there also exists a canonical computation

there also exists a canonical computation $(G,\emptyset) \vdash_{\mathcal{S}}^{+} (\square, \sigma')$ of the same length and σ and σ' are identical up to variable renaming.

Proof: apply the exchangement lemma repeatedly to the original computation $(G,\emptyset) \vdash_{\mathcal{S}}^{+} (\square, \sigma)$ until it is canonical. $\boxtimes$

$\Rightarrow$ Completeness of SLD-resolution still holds if one is restricted to canonical computations.

$\Rightarrow$ improves efficiency $\rightarrow$ derivation tree does not have to explore the different possibilities resulting from indet. 2.

Ex 436 In the derivation tree of Ex 431, we can restrict ourselves to canonical computations without losing any solutions.
In this example the resulting tree becomes finite.

Ex 437 Indet 2 can influence the termination behavior:

p :- p.
q(a).

?- q(b), p.

terminates in Prolog, because there is no canonical computation starting in $(\{\neg q(b), \neg p\}, \emptyset)$.

But there exists an infinite non-canonical computation
$(\{\neg q(b), \underline{\neg p}\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg q(b), \neg p\}, \emptyset) \vdash_{\mathcal{P}} \dots$

2 Indeterminisms

1. Which rule of the LP is used for the next res. step?
2. Which literal of the current goal is used for the next step?

Def 4.38 (SLD Tree)

Let $\mathcal{P}$ be a LP, G be a query. The <u>SLD tree</u> of $\mathcal{P}$ w.r.t. the query G is a finite or infinite tree whose nodes are marked with sequences of atomic formulas. Its edges are marked with substitutions. The SLD tree is the smallest tree such that

- If $G = \{\neg A_1, \dots, \neg A_k\}$, then the root of the tree is marked with $A_1, \dots, A_k$.
- If a node is marked with $B_1, \dots, B_n$ and B_1 is unifiable with the positive literals of the prog. clauses $K_1, \dots, K_k$ (where K_1 occurs before K_2, K_2 before K_3 etc. in $\mathcal{P}$), then the node has k successors. Then the i-th child is marked by those atoms that result from a canonical resolution step with K_i.

So if $(\{\neg B_1, \dots, \neg B_n\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg C_1, \dots, \neg C_m\}, \sigma)$ is this canonical res. step, then the i-th child is marked with $C_1, \dots, C_m$ and the edge to this child is marked with σ (restricted to the variables in $B_1, \dots, B_n$).

The <u>answer substitutions</u> can be obtained from paths ending in $\square$. If the edges from the root to the leaf $\square$ are marked with $\delta_1, \dots, \delta_\ell$, then we obtain the answer subst: $\delta_\ell \circ \dots \circ \delta_2 \circ \delta_1$ (restricted to the variables in G).

Thm 435 guarantees that by regarding the SLD-tree, we still find all answer substitutions.

Finite Paths that end in a clause different from $\square$: <u>finite failures</u>.

$B_1, \dots, B_n$ where B_1 cannot be unified with the positive literal of any prog. clause.

Moreover, there can be infinite paths.

Indet 2: resolved by the SLD-tree

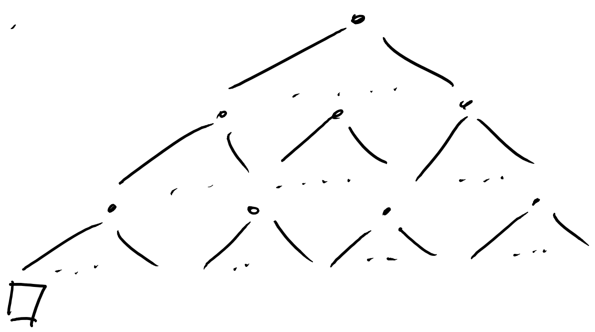
Indet 1: To resolve this indet., we have to fix the order in which the SLD-tree is constructed / traversed. (Evaluation Strategy) We also have to decide whether to stop as soon as one has found the first $\square$ or whether to search for all solutions?
(In Prolog: Stop at the first $\square$, but entering ";" makes Prolog continue to the next $\square$, etc.)

<u>Options</u>:

- **breadth-first search**: first construct all nodes of height 0, then of height 1, etc.

  advantage: completeness of canonical SLD-resolution is "preserved", i.e., every □ in the SLD-tree will be found. So if the query follows from the LP, this will be found out.

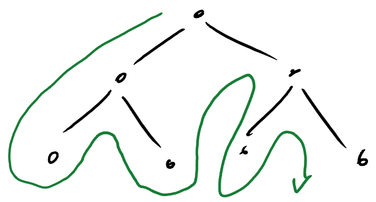
  disadvantage:



  building up the whole SLD-tree up to the level of the first □ can take very long

  ⇒ too inefficient

- **depth-first search**: used by Prolog (left-to-right)

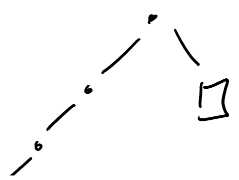


  backtrack if a path ends in finite failure or if ";" is entered after reaching □

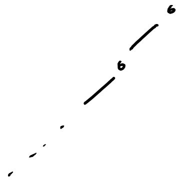
  advantage:

  □ is found very quickly if it is in the left-most paths

  disadvantage: this strategy is incomplete



  Here, depth-first search does not terminate and thus doesn't find □.



□ does not terminate and thus, does not find □ in the SLD-tree.

⇒ The programmer should take Prolog's evaluation strategy into account and ensure that solutions are found quickly before entering infinite paths.
(Choose suitable order of the literals in prog. clauses and of the clauses in the prog.)

(Violation of the principle of declarative programming:
The programmer should think (a bit) about how Prolog operates to solve queries:
1. Prog. clauses are used from top to bottom.
2. Literals in a goal are solved from left to right.)

Ex. 4.3.9 Illustrate the effect of exchanging literals in a prog. clause.
Here: the rightmost path of the tree becomes infinite

Ex. 4.3.10 Illustrate the effect of exchanging clauses in a program.
Here: the leftmost path becomes infinite
⇒ prog. does not terminate and does not find the solutions.

⇒ Non-recursive clauses for a pred. p should usually

Come before recursive clauses.

5. The programming language Prolog

Prolog: most popular logic prog. language, developed in the 1970s by Kowalski + Colmerauer.

Essentially, Prolog uses the same syntax as the logic programs in Sect 4.

- function + pred. symbols: strings starting with lower-case letter, strings consisting of special symbols (e.g., <-->, +, <, ...), ...

  Variables: strings starting with upper-case letter or with _ (e.g, _G192). Special anonymous variable _ : its instantiation is not included in answer substitutions and several occurrences of _ can be instantiated differently.

  Ex: Prog p(a,b,c).

  ?- p(_,_,X).

  Answer: X = c

- Prolog allows overloading of fct. and pred. symbols:

One may have different symbols with the same name, but different arity:

$p(a,b,c).$

$p(a,c).$

$\leftarrow$ 2 different p-symbols that have no connection

To distinguish such symbols, we often write

$$p/3 \quad \text{and} \quad p/2.$$

- To gain efficiency, Prolog does not implement proper unification, but it does not perform the occur check:

  To unify X with a term t, one has to check whether X occurs in t. In that case, X and t are not unifiable (unless $X = t$).

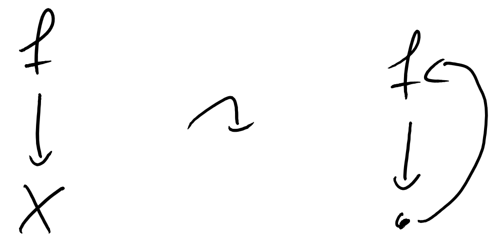
Prolog omits this check.

So for Prolog, X and $f(X)$ are unifiable

The mgu instantiates X by $\underbrace{f(f(\ldots\ldots))}_{\text{infinitely many}}$.

More precisely: X is replaced by a pointer to $f(X)$.

$\Rightarrow$ we obtain



Such terms are called rational terms
(can be represented by finite graphs).

Good prog. style: avoid this problem, do not write programs where the occur check would fail, do not write programs that construct such infinite terms.

Prolog has many pre-defined predicates, including a predicate for proper unification:

unify_with_occurs_check

```
?- unify_with_occurs_check(X, f(X)).
false

?- unify_with_occurs_check(X, f(Y)).
X = f(Y)
```

5.1. Arithmetic
5.2. Lists
5.3. Operators
5.4. Cut + Negation
5.5. Input + Output
5.6. Meta-Programming

FRI: Lectures now in AH 1 (Same room as exercise course)

5.7. Parsing with Prolog

5.1. Arithmetic

All data objects have to be represented as terms.
Suitable for data structures like lists, trees, graphs, ...
Unsuitable for numbers:

One can represent $\mathbb{N}$ by the fct. symbols

$$0 \in \Sigma_0 \quad \text{and} \quad s \in \Sigma_1.$$

Then

$$
\begin{aligned}
0 &\mathrel{\hat{=}} 0\\
s(0) &\mathrel{\hat{=}} 1\\
s(s(0)) &\mathrel{\hat{=}} 2\\
&\vdots\\
s^{1000}(0) &\mathrel{\hat{=}} 1000
\end{aligned}
$$

Drawbacks:
- one can't use efficient arithmetic operations of the operating system/processor
- unsuitable for large numbers

$\Rightarrow$ Prolog has built-in numbers.

Addition algorithm on user-defined numbers:

To implement a function of arity n, one needs a predicate of arity $n+1$.

add/3 : $add(t_1, t_2, t_3)$ iff $t_1 + t_2 = t_3$

add(X, 0, X).
add(X, s(Y), s(Z)) :- add(X, Y, Z).

?- add(s(0), s(s(0)), X). ← computes 1+2
X = s(s(s(0)))

?- add(X, s(s(0)), s(s(s(0)))). ← computes 3-2
X = s(0)

The same alg. can be used for addition and subtraction
⇒ Bidirectionality

?- add(s(0), Y, Z).
← infinitely many answer substitutions

⇒ Even simple (and reasonable) programs may have an infinite SLD-tree if one uses an unfortunate query
⇒ termination depends on query

Built-in numbers in Prolog:

arithmetic expression: term built from
- numbers (0, 1, 2, ...)
- variables (X, Y, ...)

- binary infix function symbols for arithmetic:

  $+, -, *, //, **, \ldots$

  $//$: integer division; $**$: exponentiation

- unary negation symbol $-$

In principle, arithmetic expressions are ordinary terms. Most Prolog predicate symbols treat them as ordinary terms

```
equal(X,X).

?- equal(X,Y).
X=Y.

?- equal(3, 1+2).
false.
```

← Reason: 3 and 1+2 cannot be unified.

There are some pre-defined Prolog predicates that evaluate arithmetic expressions. This is in contrast to ordinary logic programming where fct. symbols are never evaluated.

Pre-defined infix predicates for comparison of arithmetic expressions: $<$, $>$, $=<$, $>=$, $=:=$, $=\backslash=$

for $\leq$ for $\geq$ for $=$ for $\neq$

For such a predicate op:

?- t_1 op t_2.

succeeds iff at the point of evaluation, t_1 and t_2 are fully instantiated arithmetic expressions and the result of evaluating t_1 is in relation "op" to the result of evaluating t_2.

- ?- 1 < 2.
  true
- ?- 1 * 1 < 1 + 1.
  true
- ?- 6 // 3 < 5-4.
  false
- p(1).
  q(2).
  ?- p(X), q(Y), X < Y.
  true

  this is fully instantiated when this literal is evaluated
- ?- X < 1.

Program stops with error (X is not fully instant.)

- ?- a < 1.

  Program stops with error (a is no arithmetic expr.)

Problem: these predicates can't be used to instantiate variables

?- X =:= 1.

Will not result in answer subst X = 1, but in a prog. error.

Thus: another pre-defined predicate "is".

?- t_1 is t_2.

succeeds iff t_2 is a fully instantiated arithmetic expression, t_2 evaluates to some number z, and t_1 unifies with z.

?- X is 2.

X = 2

?- X is 1+1.

X = 2

?- 2 is 1+1.

true

?- 1+1 is 2.

false

?- X is 3+4, Y is X+1.

X=7, Y=8

?- Y is X+1, X is 3+4.

prog. error

Equality predicates:

=:=	arithmetic equality, both left- and right-hand side are evaluated
is	arithmetic equality, only right-hand side is evaluated
=	syntactic unification (without occur check) (i.e., is defined by the only fact X=X.)
==	syntactic identity (no unification)

?- a=a.

true

?- 2=1+1.

false

?- 1+X = Y+1.

X=1, Y=1

?- f(1,X) = f(Y,1).

X=1, Y=1

?- X = 3+4, Y is X+1.

X=3+4, Y = 8

?- X == X.

true

?- X == Y.

false

Addition with built-in numbers:

```
add(X,0,X).
add(X, s(Y), s(Z)) :- add(X,Y,Z).
```

} with user-def. numbers

```
add(X,0,X).
add(X,Y,Z) :- Y>0, Y' is Y-1, add(X,Y',Z'),
                                Z is Z'+1.
```

?- add(1,2,X)

X=3

?- add(X,2,3).

prog. error, because one reaches an is-literal where the right-hand side is not fully instantiated.

⇒ bidirectionality is lost, because the built-in arithmetic predicates are not bidirectional.

Why don't we use the following alg:

```
add(X,0,X).
add(X, Y+1, Z+1) :- add(X,Y,Z).
```

```
?- add(1,2,X).
false

?- add(1, 0+1, X).
X = 1+1
```

More arithmetic algorithms:

fact(t_1,t_2) holds iff $t_2 = t_1!$.

```
fact(0,1).
```

e.g. fact(4,24), since $4 \cdot 3 \cdot 2 \cdot 1 = 24$

```
fact(X, Y) :- X>0, X' is X-1, fact(X', Y'),
              Y is X * Y'.
```

```
?- fact(4, Y).
Y = 24
```

gcd (greatest common divisor):

gcd(t_1, t_2, t_3) iff t_3 is the gcd of t_1 and t_2 (for natural numbers)

```
?- gcd(28, 36, Z).
Z = 4
```

```
gcd(X, 0, X).
gcd(0, X, X).
gcd(X, Y, Z) :- X =< Y, X > 0, Y' is Y-X,
                gcd(X, Y', Z).
gcd(X, Y, Z) :- X > Y, Y > 0, X' is X-Y,
                gcd(X', Y, Z).
```

There is a pre-defined predicate number/1 to check whether the argument is a number.

```
?- number(2).
true

?- number(1+1).
false

?- X is 1+1, number(X).
X=2
```

For numbers, the built-in numbers lead to more readable and more efficient algorithms.

arithmetic functions are evaluated using efficient arithmetic operations of the operating system.

For lists, there is also a pre-defined data structure to increase readability.

5.2 Lists

Montag, 8. Juni 2015 08:30

Representation of lists as terms:

names from Lisp

$nil \in \Sigma_0$ to represent the empty list

$cons \in \Sigma_2$ to represent list insertion

i.e., cons(7, cons(3, nil)) stands for a list with the elements 7 and 3. – [7,3]

length-algorithm for user-defined lists:

len(l, z) iff the length of the list l is z

```
len(nil, 0).
len(cons(X, XS), Z) :- len(XS, Z'),
                       Z is Z'+1.
```

```
?- len(cons(7, cons(3, nil)), Z).
Z = 2
```

Prolog has a pre-defined data structure for lists:

[ ] $\in \Sigma_0$ for empty list

. $\in \Sigma_2$ for list insertion

$.(7, .(3, [\,]))$ stands for $[7,3]$.

$len([\,], 0).$

$len(.(X, XS), Z) \text{ :- } len(XS, Z'),$
$Z \text{ is } Z' + 1.$

For lists built with [ ] and ., Prolog offers alternative notations to improve readability:

- $.(t_1, t_2) = [t_1 \mid t_2]$ $\quad [7,3] = [7 \mid [3]]$
- $.(t_1, [\,]) = [t_1]$ $\quad .(7, [\,]) = [7]$
- $.(t_1, .(t_2, .(t_3, t))) = [t_1, t_2, t_3 \mid t]$
- $.(t_1, .(t_2, .(t_3, [\,]))) = [t_1, t_2, t_3]$

$$= [t_1, t_2 \mid [t_3 \mid [\,]]]$$

$$= [t_1 \mid [t_2, t_3 \mid [\,]]]$$

?- [1,2]==[1 | [2]].
true

← shortnotations are converted to notation with . and [ ]
=> these terms are considered

to be syntactically equal

?- .(1, X) = [1,2,3].

X = [2,3]

?- [X, [1 | X]] = [[2], Y].

X = [2], Y = [1,2]

Algorithms on lists:

app should append/concatenate lists

app(t_1, t_2, t_3) should hold iff

t_3 is the concatenation of t_1 and t_2

e.g. app([1,2], [3,4,5], [1,2,3,4,5])

(append/3 is pre-defined in Prolog).

```
app([ ], XS, XS).
app([X|XS], YS, [X|ZS]) :- app(XS, YS, ZS).
```

5.3 Operators

Freitag, 12. Juni 2015 08:30

Usually, Prolog uses Prefix-notation for function and predicate symbols:

$$p(X, f(a))$$

Such symbols are also called functors.

Sometimes, one wants to use binary symbols in infix-notation: $2+3$ instead of

$$+(2,3)$$

Similarly, one might want to use unary symbols in prefix- or postfix notation (without brackets).

$-X$ instead of $-(X)$

$X-$ —— " ————

Some pre-defined symbols are already declared as operators (e.g., +). Then $+(2,3)$ and $2+3$ are considered to be syntactically equal.

To define operators, one uses a directive of the following form:

queries contained in the program. When loading the program, Prolog tries to prove these queries

$:- \; op(Precedence, Type, Name(s)).$

Directive: clause with empty head.

determines how strong the binding of an operator is

determines whether it is an infix, prefix or postfix operator and the association of the operator

names of the symbols that are declared as operators

Pre-defined op-directives for +, -, *:

:- op(500, yfx, [+, -]).

:- op(400, yfx, *).

Precedence is needed to state how strong an operator binds its arguments. Smaller precedence means that the operator has a stronger binding.

e.g., $1 + 2 * 3$ stands for $1 + (2 * 3)$

because precedence of * is smaller than precedence of +.

Type (e.g., yfx) determines the order of operator and arguments. Here, "f" stands for the operator and y, x stand for arguments.

Types for infix-operators:

- yfx
- xfy
- xfx

x = argument with smaller precedence than f

y = argument with smaller or equal precedence than f

Precedence of an argument:

- if outermost symbol is an operator, then the precedence of the argument is the prec. of this operator:

  2+3 has precedence of +, i.e., 500

  (6*7)+3 —— " ——

- if argument is in brackets or a variable or built with a functor, then its precedence is 0.

  (2+3) has precedence 0

  2 —— " ——

Type is needed to determine whether the operator associates to the left or right:

```
:- op(500, yfx, [+,-]).
```

5 - 4 - 3 stands for (5-4)-3

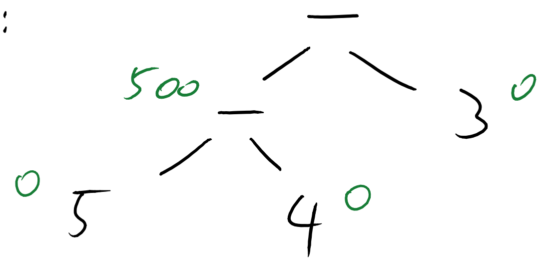
Reason:

500 —

500 0

yfx means

Reason:



yfx means that the right argument must have smaller precedence (stronger binding)

$\Rightarrow$ yfx means association to the left

xfy — " —————— right

xfx means: no association

:- op(500, xfx, +++).

Then 3 * 2 +++ 4 * 5 stands for

(3 * 2) +++ (4 * 5)

But 1 +++ 2 +++ 3 is not allowed.

$\underbrace{(1 +++ 2)}_{\text{precedence } 0}$ +++ 3 is ok.

Types for prefix operators: fx , fy

Types for postfix operators: xf , yf

Pre-defined prefix operator for negation of numbers: - is overloaded, i.e., there is a binary and a unary -.

:- op(200, fy, -).

Therefore -2-3 stands for (-2)-3

and --2 stands for -(-2)

```
- 200
|
- 200
|
2 0
```

Goal of operators: increase readability
Allows a simple form of natural language processing.

Verb: "was" should be used in infix notation

laura was beautiful instead of
was(laura, beautiful)

"was" should not associate to left or right:

laura was beautiful was young
makes no sense

:- op(300, xfx, was).

"of" should also be used in infix-notation, associates to the right:

secretary of son of john stands for

secretary of (son of john)

"of" should bind stronger than "was":

laura was secretary of john stands for

laura was (secretary of john).

```
:- op(250, xfy, of).
```

"the" unary prefix operator, type fx

(the the son makes no sense)

the secretary of the son stands for

(the secretary) of (the son)

=> "the" binds stronger than 'of'

```
:- op(200, fx, the).
```

laura was the secretary of the head of the department.

↑ Prolog fact which looks like natural language.

=> goal: programming in (almost) natural language

This fact stands for

was(laura, of(the(secretary),
of(the(head), the(department))))

Prolog-Prog:

laura was the secretary of the head of the department.

?- Who was the secretary of the head of the department.

Who = laura.

?- laura was What.

What = the secretary of the head of the department.

?- Who was the secretary of the head of What.

Who = laura
What = the department

5.4 The Cut Predicate and Negation

Montag, 15. Juni 2015 08:30

5.4.1: Built-in Cut-Predicate

Goal: do not traverse certain parts of the SLD-tree when backtracking
(in order to increase efficiency or to avoid non-termination)

5.4.2: implement Meta-Predicates like negation

female (X) :- not(male (X)).

↑ such a negation was not available yet

Reason: { female (X), male (X) } is no Horn Clause

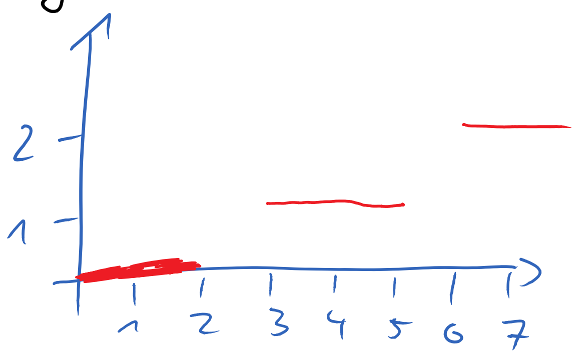
5.4.1 The Cut Predicate

Backtracking: if one reaches finite failure or if user enters ";" after reaching □.

Cut: avoid certain backtracking

Ex:

$$f(x) = \begin{cases} 0, & \text{if } x < 3 \\ 1, & \text{if } 3 \leq x < 6 \\ 2, & \text{if } 6 \leq x \end{cases}$$



f(X 0) :- X<3.

| ?- f(1,Y) 0<Y

f(X, 0) :- X<3.
f(X, 1) :- 3=<X, X<6.
f(X, 2) :- 6=<X.

?- f(1,Y), 0<Y.

Observation: The conditions

X<3

3=<X, X<6

6=<X

exclude each other.

If proving one of these conditions succeeds, one should not backtrack to try the other f-clauses.

Solution: Cut predicate !

- Predicate of arity 0
- Proof of ! always succeeds
- Side effect: as soon as ! has been proved, certain alternative paths of the SLD tree are not explored anymore.

The cuts in this program are "green cuts": only influence efficiency, but if one removes the cuts, one still gets the same results.

Efficiency of example program can be improved further: $?- f(7, Y).$

If $X<3$ succeeds in the first clause, the we will not read clause 2+3 (because of !).
If $X<3$ fails in the first clause, then there is no need to check $3 =< X$ in clause 2, because $\neg X<3$ implies $3 =< X$.
$\Rightarrow$ remove $3=<X$ from clause 2
remove $6=<X$ from clause 3.

Now the cuts are "red cuts". Removing the cuts would yield different new answer substitutions: $?- f(0,2).$ $\leftarrow$ if cuts are removed
true

In general: What is the effect of a cut?

If a query $?- A_1, \dots, A_k$
is resolved with a prog. clause $B :- C_1, \dots, C_i, !, C_{i+1}, \dots, C_n$
using mgu σ of A_1 and B,

then one obtains the SLD-tree on the slide.

Cut means that no alternatives are considered anymore for those nodes between

$A_1, \ldots, A_k$ and

$\sigma'(!, C_{i+1}, \ldots, C_n, A_2, \ldots, A_k)$.

But for the nodes above and below those two nodes, backtracking works as usual.

Example to illustrate the full effect of cut:

Version without cuts.

?- a(X).

X=0 ; X=1 ; X=2 ; X=3 ; X=4 ; X=5

Now replace the second b-clause by

b(X) :- c(Y), !, d(X,Y).

?- a(X).

X=0 ; X=1 ; X=5

Examples for using the cut in natural programs:

- gcd (greatest common divisor)

?- gcd(12, 9, Z).
Z = 3

- remove

remove(X, Xs, Ys) iff Ys results from Xs by removing all occurrences of the element X from the list Xs.

?- remove(1, [0, 1, 2, 1], Ys).
Ys = [0, 2]

The cut in clause 2 is needed to ensure that clause 3 is only reached if $X \neq Y$ (i.e., if the element to be removed is not at the beginning of the list).

If this cut were deleted, we would get:

?- remove(1, [0, 1, 2, 1], Ys).
Ys = [0, 2]; Ys = [0, 2, 1]; Ys = [0, 1, 2];
Ys = [0, 1, 2, 1]

5.4.2. Meta-Variables and Negation

Prolog allows the use of meta-variables:

variables : can be instantiated by terms
meta-variables: —— " ———— formulas

(terms: monika, date(15, 6, 2015), ...
formulas: male(gerd), married(gerd, renate), ...)

Prolog also allows meta-predicates:
predicate: applied to terms
meta-predicate: applied to formulas

Ex:

p(a). ← p is a meta-predicate (applied to formula a)

a. ← a is a predicate symbol

?- p(X), X. ← X is a meta-variable
X = a

?- X. ← no resolution with completely uninstantiated meta-variables
program error

```
or(X, Y) :- X.
or(X, Y) :- Y.
```

is pre-defined under the name ";"
using the directive

```
:- op(1100, xfy, ;)
```

Thus, one can ask query:

```
?- X = 4 ; X = 5.
```

```
X = 4 ; X = 5
```

One can also implement a meta-predicate for "if-then-else":

if(A,B,C) should implement "if A then B else C"

```
if(A,B,C) :- A, !, B.
if(A,B,C) :- C.
```

cut is needed to ensure that one does not reach clause 2 if A holds

if(A,B,C) is pre-defined in Prolog under the name "A -> B ; C"

<u>Negation</u> is implemented as "finite failure"
("Negation as failure"):

("Negation as failure"):

Goal: prove $\neg A$

```
not(A) :- A, !, fail.
not(A).
```

← pre-defined predicate whose proof always fails

is pre-defined in Prolog, can also be used as prefix-operator \+

```
not_equal(X,Y) :- not(X=Y).

?- not_equal(1,2).
true

?- not_equal(1,X).
false
```

← Negation turns $\exists$ into $\forall$.

More precisely: one has to prove not(1=X). To this end, prove 1=X, succeeds, thus not(1=X) fails.

Negation in Prolog uses two assumptions:

1. If a query doesn't hold, then this is determined in finite time.

But: ?- not(even(1)).

does not return "true", because even(1) doesn't terminate. ↳ Although "even(1)" doesn't hold, we can't detect it in finite time.

2. Closed World Assumption: If something can't be proved with our program, then it must be false.

?- not(even(-2)).

true

Since proof of even(-2) fails.

5.5 Input and Output

Freitag, 19. Juni 2015 08:30

Up to now:

Input: only via queries

Output: only via answer substitutions

Now: extra-logical predicates for "real" input + output

<u>Write/1</u>

- write(t)
  - proof always succeeds
  - side-effect: t is printed on the current output-stream (by default: screen)

```
?- X is 2+3, write(X).
5                         ← printed on the screen
X=5
```

fct. symbol of arity 0

```
?- write('Hello World').
Hello World               ← write omits quotes in the output
true
```

Prog:

```
mult(X,Y) :- Result is X*Y, write(X*Y),
             write(' = '), write(Result).
```

?- mult(3, 4).

3 * 4 = 12

true

mult(3,4)

|

Res is 3*4, write(3*4), write('='), write(Result),

|

write(3*4), write('='), write(12)

⋮

Side-Effects cannot be undone when backtracking:

```
q(a).
q(b).
p :- q(X), write(X), X = b.
```

?- p.

ab

true

p

|

q(X), write(X), X=b

X=a: write(a), a=b

| a = b

↯

X=b: write(b), b=b

| b=b

| □

nl/0

new line predicate

- always succeeds
- creates a new line in the output stream

?- write(a), nl, write(b), nl, write(c).

a
b
c
true

read/1

- read(t)
- reads a term s from the standard input stream (by default: Keyboard)
  End of term must be marked by .
- succeeds iff t and s unify
  (can be used to check which input was given by the user)

Example: sqr

Input + Output can also be done with files.
→ change input/output stream.

see/1 and tell/1, seen/0 and told/0

see(t) sets the input stream to the file t

tell(t) sets output stream to file t

tell(t) sets output stream to file t

seen
told
} close the current i/o stream and set it back to the default

Sqr-example:

Inputfile should contain 3. -4.

If the end of file is reached, then read(X) returns the answer X= end_of_file.

Afterwards, Outputfile contains

The square of 3 is 9

The square of -4 is 16

5.6 Meta-Programming

Freitag, 19. Juni 2015 08:30

Prolog can manipulate terms (Sect 5.6.1) and programs (Sect 5.6.2). In particular, a program can manipulate itself while it is running.

5.6.1. Manipulation of terms and formulas

pre-defined predicates to manipulate / access / recognize certain forms of terms:

- number/1 : checks whether arg. is a number
- var/1 : var(t) is true iff t is an (uninstantiated) variable

```
?- var(X).
true
?- X=2, var(X).
false
```

- nonvar/1 : nonvar(t) is true iff t is no variable

```
?- nonvar(a).
true
?- X=2, nonvar(X).
X=2
?- nonvar(X).
```

— although there is an instantia-

?- nonvar(X).
false

— although there is an instantiation of X such that nonvar is true

- atomic/1: atomic(t) is true iff t is a fct/pred symbol of arity 0 or a number

?- atomic(a).
true

?- atomic(-2).
true

?- atomic(a(a)).
false

?- atomic(X).
false

- compound/1: compound(t) is true iff t is a term/formula which does not just consist of a symbol of arity 0 or a number or a variable

?- compound(a).
false

?- compound(X).
false

?- compound(1+2).
true

?- compound(a(a)).
true

These predicates can be used to recognize certain forms of terms. But we also want to extract certain parts of terms (decomposition) and to construct new terms.

Solution: transform terms to lists or vice versa

e.g. $f(a,b)$ can be transformed to

$f/2$

$[f, a, b]$

$f/0$

Pre-defined predicate: $=.. \; / 2$

(infix notation)

$t \;=..\; l$ is true iff

l is the list-representation of the term t

?- f(a,b) =.. L.

L = [f,a,b]

?- $\overbrace{1+2}^{+(1,2)}$ =.. L.

L = [+,1,2]

?- f(g(a),b) =.. L.

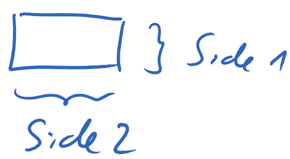
L = [f, g(a), b]

```
?- T =.. [f, a, b].
T = f(a, b)

?- T =.. [f].
T = f.

?- X =.. Y.
error
?- X =.. [Y, a, b].
error
?- X =.. [f | L].
error
```

error if the leading fct. symbol and its arity are indetermined

<u>Example for using =..</u> : Represent and enlarge different geometrical figures

square(Side) ← term to represent 

rectangle($Side_1$, $Side_2$) ← 

triangle($Side_1$, $Side_2$, $Side_3$)

circle(Radius)

Prog: enlarge(square(Side), Factor, square(NSide)) :- NSide is Factor * Side.

enlarge(rect(S_1, S_2), F, rect(NS_1, NS_2)) :- NS_1 is F * S_1, NS_2 is F * S_2.

```
enlarge(triangle(S1,S2,S3), ...) :- ...
enlarge(circle(R), ...) :- ...
```

Disadvantage: new enlarge-clause for each geometrical figure, although all these clauses essentially do the same.

Better solution:

```
enlarge(Fig, Factor, NFig) :- Fig =.. [Type | Param],
                              multlist(Param, Factor, NParam),
                              NFig =.. [Type | NParam].

multlist([], _, []).
multlist([X|L], Factor, [NX|NL]) :- NX is Factor * X,
                                    multlist(L, Factor, NL).
```

There are additional predicates to access/manipulate parts of terms:

- functor/3: functor(t, f, n) is true iff f/n is the leading fct/pred symbol of t

```
?- functor(g(f(X), X, g), F, N).
F = g, N = 3

?- functor(T, g, 3).
```

T = g(X, Y, Z).

- <u>arg/3</u>: arg(n, t, a) is true iff a is the n-th argument of t

```
?- arg(3, g(f(X), X, g), A).
A = g

?- functor(D, date, 3),
   arg(1, D, 19),
   arg(2, D, 6),
   arg(3, D, 2015).

D = date(19, 6, 2015).
```

Ex: Predicate to check whether a term is variable-free:

```
ground(T) :- nonvar(T),
             T =.. [Functor | Argumentlist],
             groundlist(Argumentlist).

groundlist([]).
groundlist([T | Ts]) :- ground(T), groundlist(Ts).
```

5.6.2 Manipulation of Programs

Prolog-prog $\hat{=}$ data base of clauses which can be read and modified

?- clause(t_1, t_2).

is true iff there is a program clause

B :- $C_1, \ldots, C_k$ ← $k=0$ is possible needed for facts: B :- true.

such that

clause(t_1,t_2) unifies with

clause$(B, (C_1, \ldots, C_k))$.

Ex: times(_, 0, 0).

times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1),
Z is Z1 + X.

?- clause(times(X,Y,Z), Body).

Y=0, Z=0, Body=true ;

Body = (Y>0, Y1 is Y-1, ..., Z is Z1 + X).

While "clause" can be used to read the code of the running program, there also exist predicates that can modify the text of the running program:

assert/1 and retract/1

?- assert(t).

Proof always succeeds, but as a side-effect, the clause t is added at the <u>end</u> of the program.

(The predicate asserta(t) adds the clause t at the <u>beginning</u> of the program. The pred. assertz is like assert.)

<u>Ex</u>:

```
?- assert(p(0)).
true
?- p(X).
X = 0
?- clause(p(X), B).
X = 0, B = true
?- assert(square(X,Y) :- times(X,X,Y)).
true
```

Clauses built with "clause" cannot be asserted. "clause" is <u>static</u>

Predicates can be static or dynamic.
By default, all predicates in the prog. are static.

Clauses for static predicates cannot be added or removed by assert + retract.

Predicates introduced by "assert" are dynamic.

Moreover, predicates in the program can be declared to be dynamic by a corresponding directive:

Ex:
```
:- dynamic times/3.
times(_, 0, 0).
times(X,Y,Z) :- Y>0, ... .
```

```
?- times(2,3,Z).
Z=6
?- asserta(times(X,1,X)).
true
?- clause(times(X,Y,Z), B).
X=Z, Y=1, B=true
```

?- retract(t)

proof succeeds iff there is a prog. clause that unifies with t. As a side effect, this prog. clause is removed.

Ex:
```
?- retract(times(X,Y,X) :- Body).
Y=1, Body=true;    ← removes times(X,1,X)
```

X = 0, Y = 0, Body = true ; ← removes times(_,0,0)
Body = ... ← removes times(X,Y,Z) :- Y>0, ...

assert + retract can lead to completely non-understandable programs ⇒ use them only for certain purposes.

Sensible use of assert + retract: compute results and store them for later use.

Ex: Store results of computations in a table to re-use these results later on and avoid their repeated re-computation.

	0	1	...	9
0	0	0		0
1	0	1		9
:	0	...	...	
9	0	9		81

```
member(X, [X|L]).
member(X, [_|L]) :-
        member(X, L).
```

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],
             member(X,L),
             member(Y,L),
             Z is X*Y,
             assert(mult(X,Y,Z)),
             fail.        ← enforces backtracking
```

↝ X and Y will range over all numbers from 0,...,9
↝ 100 new facts are added to the program.

?- maketable.
false
?- mult(X,Y,8).
X=1, Y=8 ;
X=2, Y=4 ;
X=4, Y=2 ;
X=8, Y=1.

There exists a pre-defined predicate
findall/3
which finds <u>all</u> solutions to a query (i.e., without needing the user to press ;).

findall(t, g, l) is true iff the following holds:
- Prolog tries to prove the query g and builds up the full SLD tree.
- Then it collects all answer substitutions

$\sigma_1, \ldots, \sigma_n$ (in left-to-right depth-first search).

- Then findall(t, g, l) is true iff l is the list $[\sigma_1(t), \sigma_2(t), \ldots, \sigma_n(t)]$.

Ex: family program including the rule

fatherOf(F, C) :- married(F, W), motherOf(W, C).

?- findall(C , fatherOf(gerd, C), L).

$$L = [\underbrace{\text{susanne}}_{\sigma_1(C)}, \underbrace{\text{peter}}_{\sigma_2(C)}]$$

?- findall(fatherOf(gerd, C), fatherOf(gerd, C), L).

$$L = [\underbrace{\text{fatherOf(gerd, susanne)}}_{\sigma_1(\text{fatherOf(gerd, C)})}, \underbrace{\text{fatherOf(gerd, peter)}}_{\sigma_2(\text{fatherOf(gerd, C)})}]$$

findall could be programmed ourselves using assert and retract:

findall(X , Query, Xlist) :- Query,
    assert(answer(X)),

```
collectAnswers([X|Rest]) :- retract(answer(X)),
                            !,
                            collectAnswers(Rest).
                            ;
                            fail
                            ;
                            collectAnswers(Xlist).
collectAnswers([ ]).
```

Since Prolog-programs can also be regarded as terms, one can use Prolog to write

<u>meta-programs</u> (programs that operate on programs, e.g., compilers and interpreters)

and

<u>meta-interpreters</u> (interpreter for a prog. language that is written in this prog. language).

In particular, one can also easily write interpreters for variants of Prolog.

<u>Simplest meta-interpreter (: Meta-Interpreter 0)</u>

```
prove(Goal) :- Goal.
```

If prog. contains p(0)

```
?- prove(p(X)).
X=0
```

Meta-Interpreter 1 (for pure logic programs)

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Variant of this meta-interpreter where composed goals are handled from right to left:

Meta-Interpreter 2

Variant of meta-interpreter 1 which also returns the length of the proof:

Meta-Interpreter 3

```
?- prove(fatherOf(gerd, C), N).
C=susanne, N=3
```

5.7 Difference Lists and Definite Clause Grammars

Freitag, 26. Juni 2015 08:30

Goal: • Parsing (i.e., solving the word problem for context-free languages).

Solution: • Prolog offers special support for context-free grammars
• Efficient because of the use of <u>difference lists</u>.

5.7.1. Difference Lists

Goal: more efficient implementation of list operations.

Ex: app/3 for list concatenation

?- app([1,2,3], [4,5], Zs).
Zs = [1,2,3,4,5]

Complexity: $O(n)$ where n is the length of the list in the first argument.

Goal: find an alternative append-implementation with complexity $O(1)$.

Idea: use a different representation of lists: <u>Difference Lists</u>

Difference Lists

$[1,2,3]$ can be represented as $[1,2,3,4,5]-[4,5]$

Representation is not unique.

$[1,2,3]$ could also be represented as

$$[1,2,3,4,5 \mid Ys]-[4,5 \mid Ys] \quad \text{or}$$

$$[1,2,3 \mid Ys]-Ys \quad \text{etc.}$$

most general difference list representing $[1,2,3]$

Alternative implementation of app:

$$\text{app}(Xs-Ys,\ Ys,\ Xs).$$

$$?\text{- app}([1,2,3 \mid Ys]-Ys,\ [4,5],\ Zs).$$

is not related to pre-defined subtraction. One could also any other fct. symbol.

$Zs=[1,2,3,4,5]$

Reason: in 1 resolution step we obtain $\square$

using mgu: $Ys=[4,5]$,

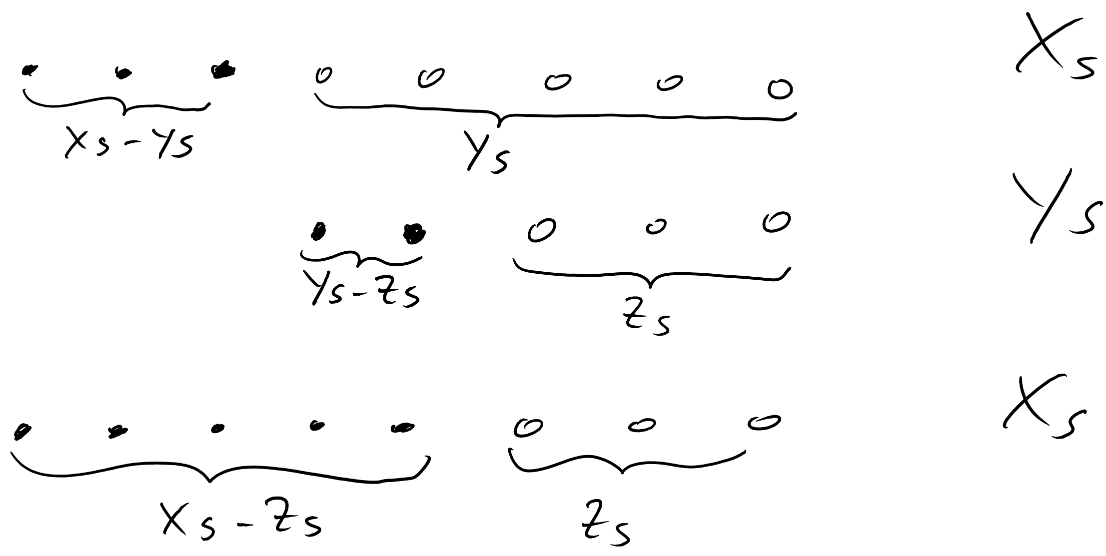
$Xs=[1,2,3,4,5]$

$Zs=$ —— " ——

Disadvantage: only arg 1 is in difference list-representation. $\Rightarrow$ app cannot be used repeatedly.

Better version, where all arguments of app are difference lists:

$$app(Xs - Ys,\ Ys - Zs,\ Xs - Zs).$$



$$app(\underline{Xs}-Ys,\ \underline{Ys}-Zs,\ \underline{Xs-Zs}).$$

$$?-\ app(\underline{[1,2,3\,|\,Ys]}-Ys,\ \underline{[4,5\,|\,Zs]}-Zs,\ \underline{Res}).$$

$$Ys = [4,5\,|\,Zs]$$
$$Xs = [1,2,3,4,5\,|\,Zs]$$
$$Res = [1,2,3,4,5\,|\,Zs] - Zs$$

Now we obtain the result in difference list-repres.
Computation only needs 1 resolution step ($O(1)$).

$$app(Xs - \underline{Ys},\ \underline{Ys} - Zs,\ Xs - Zs)$$

only works if the first 2 arguments are represented in a "compatible" way.

e.g. :?- app([1,2,3,6]-[6], [4,5]-[], Res).

do not unify

false

Better: use the most general difference list representation (e.g. [1,2,3|Ys]-Ys).

5.7.2. Definite Clause Grammars

Prolog allows representation of context-free grammars and it directly contains an efficient algorithm for parsing, based on difference lists.

→ Parsers for different languages can be easily implemented in Prolog.

Context-free grammar:

$G = (N, T, S, P)$ where

N : set of non-terminals

T : set of terminals

S : $S \in N$ start symbol

P : Set of productions (rules) of the form:

$$A \to \alpha \quad \text{with } A \in N, \alpha \in (N \cup T)^*$$

G defines a derivation relation $\Rightarrow_G$ between words:

$$\beta \Rightarrow_G \gamma \quad \text{iff}$$

there is a $A \to \alpha \in P$ such that

$\beta = \beta_1 \, A \, \beta_2$ and

$\gamma = \beta_1 \, \alpha \, \beta_2$

Grammar G defines the language

$L(G) = \{ w \in T^* \mid S \Rightarrow_G^* w \}$.

Ex: Sentence $\Rightarrow_G$

Nominalphr Verbalphr $\Rightarrow_G$

Article Noun Verbalphr $\Rightarrow_G$

a Noun Verbalphr $\Rightarrow_G$ ...

a cat scares the mouse

Representation of context-free grammars in Prolog:

- Non-terminals of N are written as constants (i.e., as predicate symbols of arity 0).
- Terminals of T are written singleton lists with a constant (e.g., [cat]).
- Words of T^* are written as lists of constants (e.g. [a, mouse, hates]). The empty word ε is written as [ ].
- Words of $(N \cup T)^*$ are written as sequences of constants and lists of constants. So "a mouse Verb Nominalphrase" is written as "[a, mouse], verb, nominalphrase".
- Instead of "$\to$", one writes -->.

Prolog translates rules built with --> into ordinary clauses.

First idea for such a translation:

- Every non-terminal could correspond to a unary predicate which checks whether its argument can be derived from this non-terminal.

- a --> [a_1, a_2, a_3] would be translated to the clause:

  non-terminal terminals

  a([a_1, a_2, a_3]). ← states that the word $a_1 a_2 a_3$ can be derived from a.

  Ex: verb --> [scares] would be translated to

  verb([scares]).

- a --> a_1 would be translated to

  a(A) :- a_1(A).

  Ex: verbalphrase --> verb would be transl. to

  verbalphrase(A) :- verb(A).

- a --> a_1, a_2 would be translated into

  a(A) :- append(A_1, A_2, A),
  a_1(A_1).

$a_2(A_2)$.

Ex: sentence --> nominalphr, verbalphr. is translated to

sentence(S) :- append(NP, VP, S),
nominalphr(NP), verbalphr(VP).

Drawback: inefficient, because append is called repeatedly (due to backtracking).

Solution: use difference lists instead.

Then: $a(A-B)$ would hold iff

from the non-terminal a one can derive the word A without its suffix B.

Prolog uses a representation of difference lists with 2 arguments: $a(A,B)$ instead of $a(A-B)$.

$\Rightarrow$ For every non-terminal a, Prolog creates a predicate symbol $a/2$.

$a(A,B)$ holds iff from a one can derive the word/list A without its end B.

- $a \dashrightarrow a_1$ is translated to

$a(A,B) \text{ :- } a_1(A,B).$

- a --> a1, a2 is translated to

```
a(A,B) :- app(Xs-Ys, Vs-Ws, A-B),
          a1(Xs,Ys),
          a2(Vs,Ws).
```

Alternative more elegant formulation:

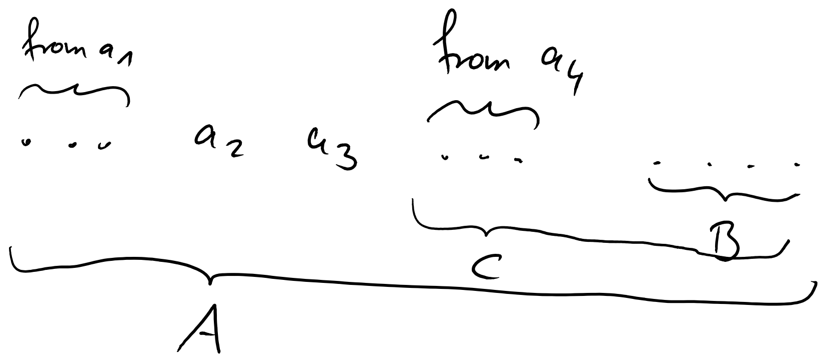
```
a(A,B) :- a1(A, C), a2(C, B).
```

- a --> [a1, a2, a3] is translated into

```
a([a1, a2, a3 | Xs], Xs).
```

- a --> a1, [a2, a3], a4 is translated into

```
a(A,B) :- a1(A,[a2, a3 | C]), a4(C, B).
```



Use of this prog. for parsing:

```
?- sentence([the, cat, scares, a, mouse], []).
true
?- sentence([the, cat, scares, a, mouse, trash], [trash]).
true
```

```
?- sentence(S, []).
S = [a, cat, scares] ;
S = [a, cat, hates] ;
⋮
```

Goal: extend logic programming by constraints

$\Rightarrow$ for the signature (Σ, Δ) introduce a sub-signature $\Sigma' \subseteq \Sigma$, $\Delta' \subseteq \Delta$ for constraints.

<u>Def. 6.1.1</u> (Constraint-Signature, Constraints)

see slide.

Constraints: • atomic formulas over sub-signature (Σ', Δ'),

• $s = t$, where s and t are arbitrary terms,

• true, fail

the special predicates in Δ' may only be applied to the special fct. symbols in Σ'

= can be applied to all fct. symbols

<u>Ex 6.1.2.</u> Constraint-Signature for integer numbers.

predicates #>= etc. are different from

$>= \in \Delta_2$ $\in \Delta_2' \subseteq \Delta_2$.

This Constraint-Signature is pre-defined in Prolog and called FD (finite domain).

Constraints: $X+Y$ #> $Z*3$

$\max(X,Y)$ #= $X \bmod 2$

$f(X) + 2 = Y + Z$

$\notin \Sigma'$

Idea: There should be a constraint solver to handle constraints which has to be combined with the ordinary mechanism to evaluate logic programs.

To determine whether a constraint is true, one needs a constraint theory CT.

Def. 6.13 (Constraint Theory)

Let $(\Sigma, \Delta, \Sigma', \Delta')$ be a constraint signature.

CT is constraint theory iff $CT \subseteq \mathcal{F}(\Sigma', \Delta', \mathcal{V})$ is satisfiable and only contains closed formulas.

no free variables, e.g. $\forall X \; X+0 \#= X$

Idea: We assume that we have a constraint solver to decide $\varphi \in CT$ for all closed formulas $\varphi \in \mathcal{F}(\Sigma', \Delta', \mathcal{V})$.

Ex 6.14 For FD, CT_{FD} should contain all true closed formulas over integers.

(CT_{FD} is not decidable, not even semi-decidable.

→ see Sect. 6.2)

Def 6.15 (Syntax of LP with Constraints)

A non-empty finite set $\mathcal{P}$ of definite Horn clauses over a constraint signature $(\Sigma, \Delta, \Sigma', \Delta')$ is a logic program with constraints iff $\{true\} \in \mathcal{P}$, $\{X = X\} \in \mathcal{P}$, and for all constraints iff $\{true\} \in \mathcal{P}$, $\{X = X\} \in \mathcal{P}$, and for all other clauses $\{B, \neg C_1, \dots, \neg C_n\} \in \mathcal{P}$ we have:

(a) if $B = p(t_1, \dots, t_m)$, then $p \notin \Delta' \cup \{true, fail, =\}$

(b) if $C_i = p(t_1, \dots, t_m)$ and $p \in \Delta'$, then

$$t_1, \dots, t_m \in \mathcal{T}(\Sigma', \mathcal{V}).$$

Condition (b) also has to hold for all queries $\{\neg C_1, \dots, \neg C_n\}$.

Ex 6.16 factorial as a CLP

Semantics of CLP: declarative + procedural semantics

Declarative Semantics: entailment from
- clauses of the program $\mathcal{P}$
- constraint theory CT

Def 6.17 (Declarative Semantics of CLP)

Let $\mathcal{P}$ be a LP with constraints, let CT be the corresponding constraint theory. Let $G = \{\neg A_1, \dots, \neg A_k\}$ be a query. Then the declarative semantics of $\mathcal{P}$ and CT wrt G is defined as:

$$D[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \dots \wedge A_k) \mid \mathcal{P} \cup CT \models \sigma(A_1 \wedge \dots \wedge A_k), \sigma \text{ ground subst.}\}.$$

Ex 6.18 $\mathcal{P}$ from Ex 6.16.

$G = \{\neg fact(1, Z)\}$

$G' = \{\neg fact(X, 1)\}$

$D[\![\mathcal{P}, CT_{FD}, G]\!] = \{ fact(1,1) \}$.

$D[\![\mathcal{P}, CT_{FD}, G']\!] = \{ fact(0,1), fact(1,1) \}$

?- fact(X,1).

X = 0 ;

X = 1

?- fac(X,1).

X = 0 ;

prog. error

Main advantages of CLP:

- efficiency
- bi-directionality

<u>Corollary 6.19</u> Let $\Sigma' = \emptyset$, $\Delta' = \emptyset$.

Then $D[\![\mathcal{P}, \emptyset, G]\!] = D[\![\mathcal{P}, G]\!]$.

(i.e.: CLP is a proper extension of LP).

Now we have to define the procedural semantics, i.e., how to evaluate CLP.

Pure LP: binary SLD-resolution with prog. clauses of $\mathcal{P}$

Problem: CT can contain arbitrary formulas (not just definite Horn clauses). Constraint solver should be used to handle CT.

Idea: also represent the SLD-resolution steps as "constraints" (to have a uniform representation of

(Evaluation steps with prog. clauses and with constraints)

these constraints are unification problems of the form:

"does the goal unify with the head of a clause?"

Ex 6.1.10. Illustrate how SLD-resolution steps can be represented as constraints.

add-program

Query: $?- add(s(0), s(0), U)$.

Idea: Do not perform the required unifications directly, but only collect the unification problems that have to be solved.

Configurations now have the form $(G, \underbrace{CO}_{\text{conjunction of unification problems } A = B})$

Start with initial configuration $(G, true)$.

In each step, check whether CO remains satisfiable (otherwise, one can't perform the desired resolution step).

Final configuration of successful computation:

$(\Box, CO)$

Now CO can be simplified to obtain the answer subst.:

$X' = s(0) \wedge Z = s(0) \wedge X = s(0) \wedge Y = 0 \wedge U = s(s(0))$

In pure LP, "=" can only be applied to terms, not to formulas. Therefore, if A and B are atomic formulas, we write $\overline{A=B}$ as an abbreviation for a corresponding conjunction of equalities ~~between~~ terms:

<u>Def 6.1.11</u> Let A, B be atomic formulas. Then we define the formula $\overline{A=B}$ as follows:

- $\overline{A=B}$ is fail, if $A=p(\ldots)$, $B=q(\ldots)$, $p\neq q$.
- $\overline{A=B}$ is true, if $A=p$, $B=p$
- $\overline{A=B}$ is the formula $s_1=t_1 \wedge s_2=t_2 \wedge \ldots \wedge s_n=t_n$ if $A=p(s_1,\ldots,s_n)$, $B=p(t_1,\ldots,t_n)$

<u>Ex 6.1.12</u> add-example using definition of $\overline{A=B}$

A configuration (G_1, CO_1) should only be evaluated to (G_2, CO_2) if CO_2 is still satisfiable (under the axioms for $=$ and true).

Thus, we check:

$$\{\forall X\ X=X,\ true\} \models \underbrace{\exists\ CO_2}_{\text{existential closure of } CO_2,\ \text{i.e., all variables of } CO_2 \text{ are existentially quantified}}$$

This variant of the procedural semantics of LP can easily be extended in order to handle constraints

- Now one can add both unification constraints (with $=$) and constraints built with Δ' (e.g. $X \#> 0$)
- When checking satisfiability of constraints, one also has to regard CT: (G_1, CO_1) can be evaluated to (G_2, CO_2) only if:

$$\{\forall X = X, true\} \cup CT \models \exists\, CO_2$$

← this needs the constraint solver

- After each evaluation step, one can <u>simplify</u> the constraints (here one has to take CT into account again)

<u>Def 6.1.14</u> (Procedural Semantics of CLP)

Let $\mathcal{P}$ be a CLP and CT be the corresponding constraint theory.

A <u>configuration</u> is a pair (G, CO) where G is a query or $\Box$ and CO is a conjunction of constraints.

<u>Computation step</u>: $(G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2)$

See slide

$P[\![\mathcal{P}, CT, G]\!]$: Here, the atoms of G are instantiated by all those ground subst. σ where $\sigma(CO)$ is true.

<u>Ex 6.1.15</u> Procedural semantics of fact.

Here: "$\neg$" omitted in the queries

Procedural semantics of fact.

Here: · "¬" omitted in the queries
 · Simplified constraints after each eval. step } for readability

A computation for query G is a (finite or infinite) sequence of configurations:

$$(G, true) \vdash_{\mathcal{P}} (G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2) \vdash \dots$$

A computation is successful iff it ends in

$$(\square, CO).$$

The answer constraints are $simplify(CO)$

where: $CT \cup \{\forall X\ X = X, true\} \models \forall (CO \leftrightarrow simplify(CO))$

Simplification can also be used after each computation step.

Thm 6.1.16 (Equivalence of declarative + procedural semantics for CLP)

Let $\mathcal{P}$ be a CLP and let CT be the corresp. constraint theory. Let G be a query.

Then: $D[\![\mathcal{P}, CT, G]\!] = P[\![\mathcal{P}, CT, G]\!]$.

CLP has the same indeterminisms as LP, and they are resolved in the same way:

Indet. 1: Which prog. clause is used for the next step?
 ⇒ top to bottom

Indet 2: Which literal of the goal is used for the next step?
 ⇒ left to right

⇒ Construct SLD trees by depth-first search from left to right.

Ex 6.1.17 ?- fac(X, 1).

$\mathrm{fac}(X,1)$

$\{X/0\}$ / $\quad$ \\

$\Box$ $\qquad$ $X > 0,\ X1 \text{ is } X-1,\ \mathrm{fac}(X_1, Y_1),\ 1 \text{ is } X * Y_1$

1. Answer Subst: $X = 0$;

Then: prog. error, because X is not instantiated in X > 0.

⇒ fac is not bidirectional

?- fact(X, 1)

Instead of labeling edges by unifiers,
we now label them by the constraints:

if $(G_1, CO_1) \vdash_g (G_2, CO_2)$,

then this results in the edge:

$$G_1 \xrightarrow{CO_2 \text{ or } simplify(CO_2)} G_2$$

CLP is bidirectional:

?- fact(X,1)

finds both solutions for X (but runs into non-termination afterwards).

If one exchanged the last 2 literals in the recursive fact-rule, it would terminate.

6.2 CLP in Prolog

Freitag, 3. Juli 2015 08:30

In Prolog, one first has to say which constraint theory CT should be used.
CLP-libraries come in modules (some of them are included in Prolog-distributions).

use_module/1 is a predicate to import modules

To import the library with the constraint theory CT_{FD}, the Prolog program must contain the directive:

```
:- use_module(library(clpfd)).
```

Problem: Constraint solver for CT is needed to check satisfiability of constraints in each computation step (and to simplify constraints).
$\Rightarrow$ should be automatic + efficient

But: most constraint theories are undecidable or only have very time-consuming decision procedures (CT_{FD} is undecidable).

Solution: Instead of checking

$$CT \cup \{\forall X = X, \mathit{true}\} \models \exists\, CO$$

this question is only "approximated".
This is efficient, but not always correct
(i.e., there might be conjunctions of constraints CO where Prolog falsely detects their satisfiability).

To approximate satisfiability in CT_{FD}, one typically uses <u>path-consistency</u>.

<u>Def. 6.2.1</u> (Path Consistency)
Let $CO = \varphi_1 \wedge \ldots \wedge \varphi_m$ be a conjunction of constraints with $\varphi_i \in At(\Sigma_{FD}, \Delta_{FD}, \mathcal{V})$. Let $X_1, \ldots, X_n$ be the variables in CO and let $D_1, \ldots, D_n$ be subsets of $\mathbb{Z}$. $D_1, \ldots, D_n$ are <u>admissible domains</u> for $X_1, \ldots, X_n$ w.r.t. CO iff for all φ_i and all variables X_j the following holds:

for all $a_j \in D_j$ there exist $a_1 \in D_1, \ldots, a_{j-1} \in D_{j-1}, a_{j+1} \in D_{j+1}, \ldots, a_n \in D_n$

such that $CT_{FD} \models \varphi_i[X_1/a_1, \ldots, X_n/a_n]$.

CO is <u>path-consistent</u> iff there are admissible domains

$D_1, \dots, D_n$ that are all not empty.

Problem: Satisfiability of the constraints separately:
if φ_1 and φ_2 are both satisfiable,
then $\varphi_1 \wedge \varphi_2$ still does not need to be satisfiable.

Automated checking of Path-Consistency:

1. Let $D_1 = \mathbb{Z}, \dots, D_n = \mathbb{Z}$.
2. Process the constraints φ after each other. For each φ:
3. Process the variables after each other. For X_j:
   Reduce its domain D_j to those values where φ can be made true if the other variables can only take values from their domains.
4. The whole process is repeated until the constraints do not change any more.

Ex 6.22 Let CO be

$$X_1 \ \#> 5 \ \wedge \ X_1 \ \#< X_2 \ \wedge \ X_2 \ \#< 9$$

- Beginning: $D_1 = \mathbb{Z}$, $D_2 = \mathbb{Z}$
- Consider $X_1 \ \#> 5 \ \Rightarrow \ D_1 = \{6, 7, 8, \dots\}$, $D_2 = \mathbb{Z}$
- Consider $X_1 \ \#< X_2 \ \Rightarrow$

$D_1 = \{6,7,8,\dots\}$, $D_2 = \mathbb{Z}$ — For every $a_1 \in D_1$ there exists $a_2 \in D_2$ such that $a_1 \#< a_2$.

$D_1 = \{6,7,\dots\}$, $D_2 = \{7,8,\dots\}$ — For every $a_2 \in D_2$ there exists $a_1 \in D_1$ such that $a_1 \#< a_2$.

- Consider $X_2 \#< 9 \Rightarrow D_1 = \{6,7,\dots\}$, $D_2 = \{7,8\}$
- Consider $X_1 \#> 5 \Rightarrow D_1 = \{6,7,\dots\}$, $D_2 = \{7,8\}$
- Consider $X_1 \#< X_2 \Rightarrow D_1 = \{6,7\}$, $D_2 = \{7,8\}$

$\vdots$

Now nothing changes anymore $\Rightarrow$
CO is path-consistent ($D_1 \neq \emptyset$, $D_2 \neq \emptyset$).

Simplify(CO) = X1 in 6..7, X1 #< X2, X2 in 7..8

The constraint signature contains more symbols:
in, ...

CT_{FD} "should" be used for finite domains, but this is not enforced:

?- X1 in 6..sup, X1 #< X2, X2 in inf..8

$\hat{=}\, \infty$ (sup) $\qquad \hat{=}\, -\infty$ (inf)

CT_{FD} has a predicate "label" which enforces that all solutions are enumerated:

?- X1 #> 5, X1 #< X2, X2 #< 9, label([X1,X2]).

list of variables for which answer substitutions should be computed

X1 = 6, X2 = 7 ;
X1 = 6, X2 = 8 ;
X1 = 7, X2 = 8 ;
false

label can only be used if all the variables have finite domains:

?- X1 #> 5, X1 #< X2, label([X1,X2]).

prog. error

Ex 6.2.3 Incorrectness of Path Consistency

?- X1 #> X2, X1 #=< X2.

$X_1 \#> X_2 \wedge X_1 \#=< X_2$ is path-consistent, but unsatisfiable.

$D_1 = \mathbb{Z}$ $D_2 = \mathbb{Z}$ are admissible domains.

- for every $a_1 \in D_1$, there exists $a_2 \in D_2$ such that
  a_1 #> a_2
- for every $a_1 \in D_1$, there exists $a_2 \in D_2$ such that
  a_1 #=< a_2
- for every $a_2 \in D_2$, ...

Ex 6.2.4 n-queens problem

- chess board of size $n \times n$
- place n queens on the board that cannot beat each other

	1	2	3	4
1			X	
2	X			
3				X
4		X		

- Represent the positions of queens by a list $[X_1, \dots, X_n]$ where X_i is the row for the queen of column i. (e.g. [2, 4, 1, 3]).
- ?- queens(4, L) will compute solution L for chess-board of size 4x4.
- "L ins 1..N" means "X in 1..N" for

every element X of L

- all_different is pre-defined
- first three literals: L is a permutation of {1,.., N}