Disclaimer

This is an attempt on a summary of the keywords[1] given for the course TDDD08. For your own sake, don't assume that anything is correct. The explanations have been taken from the course book[2] and the lectures[3].

Contents

[1]`https://www.ida.liu.se/~TDDD08/misc/keywords.txt`
[2]`http://www.ida.liu.se/~ulfni/lpp/`
[3]`https://www.ida.liu.se/~TDDD08/lectures.shtml`

1 Logic

1.1 Language

In the formal language of predicate logic objects are represented by terms.

In natural language only certain combinations of words are meaningful sentences. The counterpart of sentences in predicate logic are special constructs built from terms.

1.1.1 Term

A term is an element of the language,

- All variables are terms
- All constants are terms
- If f is an n-ary functor and $t_1, \ldots, t_n$ are terms is $f(t_1, \ldots, t_n)$ a term

1.1.2 Functor

- A functor takes n terms and evaluates to a new term
- A constant can be seen as a functor without arguments

1.1.3 Predicate

A predicate takes n terms and evaluates to a truth-value

- A 1-ary predicate can be seen as a property
- A n-ary predicate can be seen as a relation

1.1.4 Free/bound variables

Let F be a formula and X a variable. An occurance of X in F is said to be bound to F if

- The occurence follows a quantifier, or
- The occurence is part of a subformula that follows $\forall X$ or $\exists X$.

If the occurence of X is not bound it is said to be free.

1.1.5 Ground term/formula

A term or formula is said to be ground if it does not contain any variables.

1.1.6 Closed formula

A closed formula is a formula that does not contain any free variables.

1.1.7 Atomic formula (atom)

An atomic formula is a logical statement (a predicate applied to a tuple of terms without logical connectives).

- If P is an n-ary predicate symbol and $t_1, t_2, \ldots, t_n$ are terms, then $P(t_1, t_2, \ldots, t_n)$ is an atomic formula.

1.2 Interpretation

The interpretation of constants, functors and predicate symbols provide a basis for assigning truth values to the formulas.

An interpretation $\mathcal{J}$ of an alphabet $\mathcal{A}$ is a non-empty domain $\mathcal{D}$ (or $|\mathcal{J}|$) and a mapping that associates

- Each constanct $c \in \mathcal{A}$ with an element $c_{\mathcal{J}} \in \mathcal{D}$
- Each n-ary functor $f \in \mathcal{A}$ with a function $f_{\mathcal{J}} : \mathcal{D}^n \rightarrow D$
- Each n-ary predicate symbol $p \in \mathcal{A}$ with a relation $p_{\mathcal{J}} \subseteq \mathcal{D} \times \ldots \times \mathcal{D}$ (n times)

1.2.1 Valuation

A valuation φ is a mapping from variables of the alphabet to the domain of an interpretation. It is a function that assigns objects of an interpretation to the variables of the language.

Let $\mathcal{J}$ be an interpretation, φ a valuation and t a term. Then $\varphi_{\mathcal{J}}(t)$ is an element in $\mathcal{D}$ defined as

- If t is a constant c then $\varphi_{\mathcal{J}}(t) := c_{\mathcal{J}}$
- If t is a variable X then $\varphi_{\mathcal{J}}(t) := \varphi(X)$
- If t is of the form $f(t_1, \ldots, t_n)$ then $\varphi_{\mathcal{J}}(t) := f_{\mathcal{J}}(\varphi_{\mathcal{J}}(t_1), \ldots, \varphi_{\mathcal{J}}(t_n))$

$\varphi[X \mapsto t]$ denotes the valuation which is identical to φ except that X maps to t.

1.3 Model

An interpretation $\mathcal{J}$ is said to be a model of a set of closed formulas P iff every formula $F \in P$ is true in $\mathcal{J}$.

1.3.1 Satisfiability

A satisfiable set of closed formulas has atleast one model.

1.4 Logical consequence

A closed formula F is called a logical consequence of a set of closed formulas P iff F is true in every model of P ($P \models F$).

A way to prove $P \models F$ is to show that $\neg F$ is false in every model of P. That $P \cup \{\neg F\}$ is unsatisfiable.

1.5 Soundness, completeness

- Inference rules are said to be sound if what is possible to derive with them ($P \vdash F$) are also logical consequences ($P \models F$).
- Inference rules are said to be complete if they make every logical consequences ($P \models F$) derivable ($P \vdash F$) using the rules.

2 Definite programs

A definitie program is a finite set of definite clauses.

2.1 Syntax (rules, facts, queries/goals)

2.1.1 Definite clause

A definite clause is a clause that contains exactly one positive literal.

- $\forall(A_0 \lor \neg A_1 \lor \ldots \lor \neg A_n)$ which is equivalent to
- $A_0 \leftarrow A_1, \ldots, A_n$

2.1.2 Rule

A rule is a definite clause where $n \geq 1$.

- $A_0 \leftarrow A_1, \ldots, A_n$

2.1.3 Fact

A fact is a definite clause where $n = 0$.

- A_0

2.1.4 Queries/Goals

Queries and goals are definite clauses with no positive literals.

- $\leftarrow A_1, \ldots, A_n$

The difference between a query and a goal is that a query asks a question (contains variables) and a goal verifies a statement. A query returns solutions with respect to variables, a goal answers yes or no.

2.2 Proof trees

A proof tree (implication tree) for a definitie program P is a finite tree in which

- $B_1, \ldots, B_n$ are children of a node $B \Rightarrow$
  $B \leftarrow B_1, \ldots, B_n$ is an instance of a fact of P.

For any definite program P,atoms $A_1, \ldots, A_n$ and A

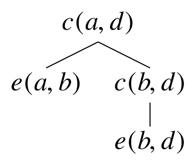
- $P \models A_1, \ldots, A_n$ iff $P \models A_1$, and . . ., and $P \models A_n$
- $P \models A$ iff A is the root of some proof tree for P.

Example Let the program be

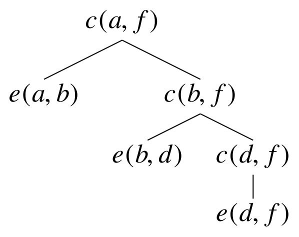
$$\begin{array}{ll} c(X,Z) \leftarrow e(X,Z) & e(a,b).e(d,f) \\ c(X,Z) \leftarrow e(X,Y), c(Y,Z) & e(b,d) \end{array}$$

Two examples of proof trees of the program are

Tree 1



Tree 2



2.3 Atomic logical consequences of a program (proof tree roots)

The atomic logical consequences of P are all the atomic formulae which may be obtained from the facts of P by applying modus ponens.

For sets J of possible non-ground atoms

- $T_P^c(J) = \{A \mid A \leftarrow A_1, \ldots, A_n \text{ is an instance of a } C \in P, \{A_1, \ldots, A_n\} \subseteq J\}$
  - $(T_P^c)^i(\emptyset)$ denotes the roots of proof trees of height less than i.

Let $PTR(P)$ be the **proof tree roots** of P. Then

- $PTR(P) = \bigcup_{i=1}^{\infty} (T_P^c)^i(\emptyset)$

2.4 Correctness / completeness (of a program) w.r.t. a specification

2.5 Incorrectness diagnosis, incompleteness diagnosis

2.6 Herbrand universe/base/interpretation

Let $\mathcal{A}$ be an alphabet with at least one constant symbol

- The **Herbrand universe of** $\mathcal{A}$ ($U_{\mathcal{A}}$) is the set of all ground terms constructed from the alphabets functors and constants.
- The **Herbrand base of** $\mathcal{A}$ ($B_{\mathcal{A}}$) is the set of all ground, atomic formulas over the alphabet [all the predicates and every combination of the elements in the universe as input].

The Herbrand universe and base are (for our purposes) defined for a given program P. The alphabet is assumed to contain the symbols that appear in P.

- The **Herbrand interpretation of** P is an interpretation $\mathcal{J}$ that
  - The domain of $\mathcal{J}$ is U_p
  - For every constant c, $c_{\mathcal{J}}$ is defined as c
  - For every n-ary functor f the function $f_{\mathcal{J}}$ is defined as $f_{\mathcal{J}}(x_1, \dots, x_n) := f(x_1, \dots, x_n)$
  - For every n-ary predicate symbol p, $p_{\mathcal{J}}$ is a subset of U_P^n

The Herbrand interpretation has predefined meanings of functors and constants. In order to specify a Herbrand interpretation it suffices to list the relations associated with the predicate symbol.

Example For an n-ary predicate symbol p and a Herbrand interpretation $\mathcal{J}$ the meaning of $p_{\mathcal{J}}$ consists of the set $\{(t_1, \dots, t_n) \in U_P^n \mid \mathcal{J} \models p(t_1, \dots, t_n)\}$

2.7 Least Herbrand model

- A **Herbrand model of** P is a Herbrand interpretation $\mathcal{J}$ which is a model of every formula $F \in P$ ($\mathcal{J} \models F$).
- The **least Herbrand Model of** P (M_P) is the set of all ground atomic logical consequences of the program ($M_P = \{A \in B_P \mid P \models A\}$).

2.8 Immediate consequence operator

3 SLD-resolution

3.1 Substitutions

A substitution is a finite set of pairs of terms $\{X_1/t_1, \dots, X_n/t_n\}$ where

- t_i is a term
- X_i is a variable
- $X_i \neq t_i$ and $X_i \neq X_j$

The **empty substitution** is denoted ϵ.

3.1.1 Application

$X\theta$ denotes the application of a substitution θ to a variable X.

$$X\theta = \begin{cases} t; X/t \in \theta \\ X; \text{else} \end{cases}$$

3.1.2 Composition

Let θ and σ be two substitutions

- $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$
- $\sigma = \{Y_1/t_1, \ldots, Y_n/t_n\}$

The composition $\theta\sigma$ is obtained from $\{X_1/s_1\sigma, \ldots, X_m/s_m\sigma, Y_1/t_1, \ldots, Y_n/t_n\}$ by

- Removing all $X_i/s_i\sigma$ where $X_i = s_i\sigma$
- Removing all Y_j/t_j for which $Y_j \in \{X_1, \ldots, X_m\}$

3.2 Unifier

A unifier is a substitution θ such that applying θ to terms s, t makes them identical ($\theta s = \theta t$). θ is a unifier of s, t. The search for a unifier is the process of solving $s \doteq t$.

3.2.1 Generality of substitutions

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

3.2.2 Most general unifier

A unifier θ is said to be a mgu of two terms iff θ is more general than any other unifier of the terms.

3.2.3 Solved form

A set of equations $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ is said to be in solved form iff $X_1, \ldots, X_n$ are distinct variables none of which appear in $t_1, \ldots, t_n$.

If $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ is in solved form then $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ is an mgu of $(X_1, \ldots, X_n)$ and $(t_1, \ldots, t_n)$.

3.3 Unification algorithm

- **Input** A set of ε equations
- **Output** An equivalent set of equations in solved form or failure
- **Repeat until no action is possible on any equation in** ε
  - Select an arbitrary $s \doteq t \in \varepsilon$
  - **(Case 1)** $f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n), n \geq 0$
    - Replace equation by $s_1 \doteq t_1, \ldots, s_n \doteq t_n$
  - **(Case 2)** $f(s_1, \ldots, s_n) \doteq g(t_1, \ldots, t_n), f/m \neq g/n$
    - Halt with failure
  - **(Case 3)** $X \doteq X$
    - Remove the equation
  - **(Case 4)** $t \doteq X$, t is not a variable
    - Replace the equation by $X \doteq t$
  - **(Case 5)** $X \doteq t$, $X \neq t$, X occurs more than once in ε
    - **(Case 5a)** X is a proper subterm of t
      - Halt with failure
    - **(Case 5b)**
      - Replace all other occurences of X by t

Due to inefficiency of occurs check (Case 5a) it is often removed.

In Prolog this check is omitted which means that Case 5b is always executed when $X \doteq f(X)$. An infinite tree is constructed, $X \doteq f(f(...))$ (actually a graph with a cycle is built).

3.3.1 Renaming

A substitution $\{X_1/Y_1, \dots, X_n/Y_n\}$ is called a renaming substitution iff $Y_1, \dots, Y_n$ is a permutation of $X_1, \dots, X_n$

If $\theta = \{X_1/Y_1, \dots, X_n/Y_n\}$ is a renaming then $\theta^{-1} = \{Y_1/X_1, \dots, Y_n/X_n\}$ is its reverse. $\theta\theta^{-1} = \epsilon$

3.4 SLD-resolution

Given a program P, compute answers for an initial query Q_0. The computation step is done by

- $Q = A_1, \dots, \boldsymbol{A_i}, \dots, A_m$ - a query
- $C = \boldsymbol{H} \leftarrow B_i, \dots, B_n$ - a variant of a clause of P
- θ - mgu of $\boldsymbol{A_i}$ and $\boldsymbol{H}$

- $Q' = (A_i, \dots, A_{i-1}, \boldsymbol{B_1, \dots, B_n}, A_{i+1}, \dots, A_m)\theta$

Note that $P \models Q' \rightarrow Q\theta$

3.5 SLD-derivation

An SLD-derivation is a sequence of queries, clauses and mgu:s

- $Q_0 \overset{C_1}{\underset{\theta_1}{-}} \rightarrow Q_1 \overset{C_2}{\underset{\theta_2}{-}} \rightarrow Q_2 \dots Q_{n-1} \overset{C_n}{\underset{\theta_n}{-}} \rightarrow Q_n$

Where

- Each C_i is a variant of a clause of P
- Each Q_i is a resolvent of Q_{i-1} and C_i with mgu θ_i
- No variables of C_i occurs in $Q_0, \dots, Q_{i-1}$, $\theta_1, \dots, \theta_{i-1}$ and $C_1, \dots, C_{i-1}$
  (the variables in C_i are new)

Obtaining a clause variant satisfying the last requirement requires renaming.

Note that $P \models Q_n \rightarrow Q_0\theta_1 \dots \theta_n$. A derivation is successful if Q_n is true, then $P \models Q_0\theta_1 \dots \theta_n$. $P \models Q_0\theta_1 \dots \theta_n$ is the computated answer for P and Q_0.

An SLD-resolution is sound since each computed answer is a correct answer. It can be seen as an attempt to construct a proof tree for some instance of A_i top-down.

- Q_j leaves - a partially constructed tree
- Other leaves - instances of facts of the program
- $A\theta_1 \dots \theta_j$ - the root of the tree

- $Q_0 \xrightarrow[\theta_1]{C_1} Q_1 \xrightarrow[\theta_2]{C_2} Q_2 \ldots \xrightarrow[\theta_n]{C_n} \blacksquare$ - a successful derivaton

Example

Let the program P be

$c(X, Z) \leftarrow e(X, Z)$	$e(a, b).e(d, f)$
$c(X, Z) \leftarrow e(X, Y), c(Y, Z)$	$e(b, d)$

Then

$Q_0 = c(X, Y)$	$C_1 = c(X', Z') \leftarrow e(X', Y'), c(Y', Z')$ $\theta_1 = \{X'/X, Z'/Y\}$
$Q_1 = e(X, Y'), c(Y', Y)$	$C_2 = e(a, b)$ $\theta_2 = \{X/a, Y'/b\}$
$Q_2 = c(b, Y)$	$C_3 = c(X'', Z'') \leftarrow e(X'', Y'')$ $\theta_3 = \{X''/b, Z''/Y\}$
$Q_3 = c(X, Y)$	$C_4 = e(b, d)$ $\theta_4 = \{Y/d\}$
$Q_5 = \blacksquare$	

3.6 SLD-tree

For an SLD-tree for program P, query Q via (computed with) rule $\mathcal{R}$ the following is true

- Branches are SLD-derivations for P, Q via $\mathcal{R}$
- A node Q' with A selected has exactly one child for each applicable clause C of P. The child is a resolvent of Q' and a variant of C with respect to A

3.7 Selection rule (computation rule)

A function selecting atom in a query Q_i to be resolved is called a selection rule or a computation rule. The Prolog selection rule selects the first atom of the query to be resolved.

3.8 Soundness and completeness

4 Negation

4.1 Closed world assumption

4.2 Negation as (finite) failure

4.3 Program completion

4.4 General logic programs

4.5 SLDNF-resolution, SLDNF-forest

4.5.1 successful derivation, failed tree, floundering

5 Definite clause grammars

5.1 Translation to Prolog

6 Prolog

6.1 Terms as the data structures of logic programming

6.2 Recursive data structures

6.3 Lists and difference lists

6.4 Constraints (especially FD-constraints)

6.5 The cut - don’t use it.