Deductive Databases and Logic Programming

(Summer 2006)

Chapter 1: Introduction

- What is Prolog? What are Deductive Databases?
- First Example.
- Motivation.
- Integrated Systems: DB + PL.
- History, State, and Future.

Objectives

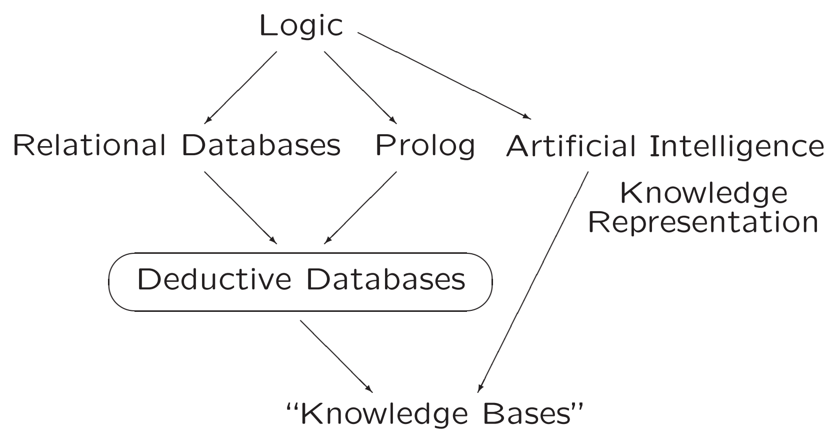
After completing this chapter, you should be able to:

- explain the difference between declarative (logic) programing and imperative programming.
- explain what a deductive database is.
- explain the main strengths/advantages of deductive databases compared with classical relational DBs.
- explain the current problems.
- develop simple Prolog/Datalog programs.

Overview

1. What are Prolog and Deductive DBs?
2. How to use a Prolog System
3. DDBs as Integrated Systems: DB + PL
4. Strengths of Deductive Databases
5. History, Current Problems

Area / Context



Logic Programming (1)

- Ideal of logic/declaractive programming:
  - ◇ The program is a specification of the problem,
  - ◇ and the system automatically computes a possible solution that satisfies the given conditions.
- I.e. the programmer specifies
  - ◇ what is the problem, but not
  - ◇ how to compute a solution.
- I.e. the program is a set of axioms, and computation is a proof of a goal statement from the program.

Logic Programming (2)

- SQL is a declarative query language. The user specifies only conditions for the requested data:

```
SELECT X.HOMEWORK, X.POINTS
FROM   SOLVED X
WHERE  X.STUDENT = 'Ann Smith'
```

- Advantages of declarative languages:
  - ◇ Often simpler formulations: The user does not have to think about efficient execution.
  - ◇ Easier adaptation to changing environments.
  - ◇ Better formalization, simpler verification.

Logic Programming (3)

- Can this work also for programming languages, not only query languages?
- Prolog ("Programming in Logic") is
  - ◇ a programming language based on these ideas.
  - ◇ used in industry for real problems.
  - ◇ not an ideal logic programming language.

    Sometimes one must know how it is executed and give some execution information. A Prolog program is not pure logic.
- Deductive databases are purer, but are still in the research state.

Logic Programming (4)

- Programming Inefficiency $*$ Runtime Inefficiency $\geq$ Constant (Robinson)
- In declarative languages,
  - ◇ the productivity of the programmer is often greater than in imperative languages, but

    E.g. Prolog program 10-fold shorter than similar C++ Program.
  - ◇ the runtime of the program is often longer.
- Programmers are expensive ("software crisis"), computers become faster and cheaper.

  I.e. the constant in the above inequation shrinks because of advances in hardware technology.

Logic Programming (5)

- Algorithm = Logic + Control (Kowalski)
  Imperative/Procedural Language: Explicit Control, Implicit Logic.
  Declarative/Descriptive Language: Explicit Logic, Implicit Control.
- Imperative languages (e.g. C) are coupled to the Von Neumann architecture of today's computers.
- Declarative languages have a larger independence of current hardware/software technology:
  - ◇ Simpler Parallelization
  - ◇ More powerful optimization

    This includes using new algorithms: If the next version of Oracle contains a new join algorithm, existing queries will profit from it. Already using a new index is a new algorithm.

Logic Programming (6)

- Naturally, logic programming languages are especially well-suited for knowledge-intensive tasks.

  However, e.g. compilers for Prolog are normally written in Prolog.
- E.g. expert systems and natural language processing are typical applications of Prolog.
- Also problems, where only constraints for a solution are given (development of time-tables, schedules) are well treated by logic programming.

  In Prolog or special "Constraint Logic Programming" languages. For NP-complete problems, there is anyway no good algorithm, so why bother to write one down?

Deductive Databases (1)

A Deductive Database is ...

- An integrated system consisting of a DB and a declarative programming language (Prolog-like).
- "Pure Prolog" with special support for managing large sets of facts.
- A relational DB with a new query language (Datalog) and the possibility to define recursive views.
- A restricted theorem prover: It can handle only quite simple formulas, but very many of them.

  "deduce" = "compute a logical consequence".

Deductive Databases (2)

A Deductive Database Consists of ...

- a relational database (EDB),
  which defines relations/predicates "extensionally",
  i.e. by enumerating all tuples, and
- a logic program (IDB),
  which defines relations/predicates "intensionally",
  i.e. by giving rules (formulas of a particular kind).

Example for extensional vs. intensional:
time-table/schedule for busses (very regular data).

Example Database (1)

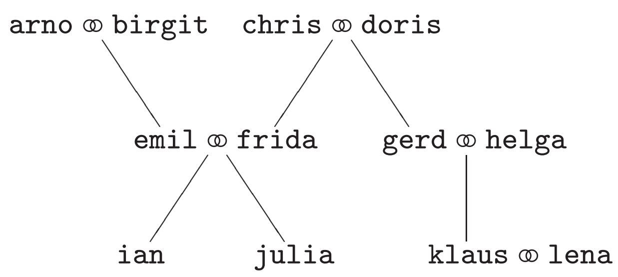
parents		
Child	Father	Mother
emil	arno	birgit
frida	chris	doris
gerd	chris	doris
ian	emil	frida
julia	emil	frida
klaus	gerd	helga

couple	
Man	Woman
arno	birgit
chris	doris
emil	frida
gerd	helga
klaus	lena

man
Name
arno
⋮

woman
Name
birgit
⋮

Example Database (2)



Identifiers in Prolog

- In Prolog, identifiers start with a lowercase letter.

  Otherwise they contain uppercase and lowercase letters, as well as digits and the underscore symbol "_". The reason for this restriction is that in Prolog, variables are distinguished from constants etc. by starting them with an uppercase letter.

- Alternatively, one can use any sequence of characters enclosed in single quotes ' (apostroph).
- If one wants that the names in the example start with an uppercase letter, quotes are needed: `'Arno'`.

  Of course, one must decide for one, `arno` and `'Arno'` are not the same in Prolog (it is case-sensitive). However, `arno` and `'arno'` are the same.

Predicates (1)

- Logic is the science of statements and their interrelationships, especially consequence.
- Consider a statement with placeholders, e.g. "C is child of F (father) and M (mother)".
- Let us abbreviate this to "$\texttt{parents}(C, F, M)$".
- The statement can be true or false if concrete values are given for the placeholders.
- E.g. in the situation described in the above DB:
  - ◇ $\texttt{parents}(\texttt{emil}, \texttt{arno}, \texttt{birgit})$ is true,
  - ◇ $\texttt{parents}(\texttt{emil}, \texttt{chris}, \texttt{doris})$ is false.

Predicates (2)

- "`parents`" is an example for a predicate symbol. It has three arguments: child, father, mother.
- Formally, a predicate is a function that assigns true or false to given values for the arguments.
- A predicate symbol is a name for such a function.

  One could also choose another name, such as `p` or `child_of`. One only has to use one name consistently. Logic and Prolog do not understand the meaning of the name, they only know the specified facts and rules.
- Since logic analyses statements, it carefully distinguishes between symbols and their interpretation.

  Relation names are defined in the DB schema, relations in the state.

Predicates (3)

- The extension of a predicate is the set of argument tuples for which the predicate is true.

  (`emil`, `arno`, `birgit`) belongs to the extension of `parents` (in the situation of the above DB state), while (`emil`, `chris`, `doris`) does not.
- Predicates (with finite extension) are really the same as (database) relations.

  E.g. given a relation, one can see it as prediate that is true for the tuples in the relation, and false for all other argments. In the opposite direction, one chooses the extension of the predicate as the relation.
- In Prolog, one can define predicates with infinite extension, e.g. `odd`(n) is true iff n is an odd number.

Predicates (4)

- In logic and Prolog, the arguments of a predicate are identified by position.

  I.e. one must know that the first argument is the child, the second the father, and the third the mother. The names of the placeholders in the original statement (C, F, M) are not important.
- In SQL, the columns of a table (attributes of a relation) are identified by name.
- However, one could also define a logic programming language that uses argument names.

  If there are few arguments, and one applies consistent style rules for ordering the arguments, then the Prolog notation is quicker (more concise). With many arguments, the SQL notation is safer.

Example DB as Facts (1)

```
parents(emil, arno, birgit).
parents(frida, chris, doris).
parents(gerd, chris, doris).
parents(ian, emil, frida).
parents(julia, emil, frida).
parents(klaus, gerd, helga).

couple(arno, birgit).
couple(chris, doris).
couple(emil, frida).
couple(gerd, helga).
couple(klaus, lena).
```

Example DB as Facts (2)

```
man(arno).
man(chris).
man(emil).
man(gerd).
man(ian).
man(klaus).

woman(birgit).
woman(doris).
woman(frida).
woman(helga).
woman(julia).
woman(lena).
```


Logical Formulas (1)

- If there were only such elementary statements, logic would not be very interesting.
- However, one can combine statements with logical connectives, e.g.:
  - ◇ $\wedge$: logical "and" (conjunction)
  - ◇ $\vee$: logical "or" (disjunction)
  - ◇ $\neg$: logical "not" (negation)
  - ◇ $\leftarrow$: logical "if"
  - ◇ $\leftrightarrow$: logical "iff" (if and only if)

Logical Formulas (2)

- One can also introduce variables:
  - ◇ $\forall X$: "for all X" (universal quantification)
  - ◇ $\exists X$: "there is an X" (existential quantification)
- In SQL, such formulas are used as query language.

  SQL has no universal quantifier, except in a specific context: `>= ALL`. However, one can simulate it with `EXISTS`-subqueries. Actually, it is a result of mathematical logic that one kind of quantifier suffices.
- Prolog is a restricted automated theorem prover: Knowledge can be specified not only as facts (as in RDBs), but also as rules (special kind of formulas).

Rules (1)

- One can define predicates not only by facts, but also by "if-then" rules:

  $$\mathtt{father}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{parents}(\mathtt{X},\mathtt{Y},\mathtt{Z}).$$

  "If `Y` and `Z` are parents of `X`, then `Y` is father of `X`".
- A rule has two parts:
  - ◇ Rule Head: The left hand side, the conclusion.
  - ◇ Rule Body: The right hand side, the condition.
- If the rule body is satisfied (for certain values of `X`, `Y`, `Z`), the rule head can be derived (with the same values of `X`, `Y`, `Z`).

Rules (2)

- The above rule defines the predicate "`father`" and uses the predicate "`parents`".

  I.e. it assumes that there is information about `parents` that can be used to derive information about `father`. Prolog does not require a specific sequence of declaration: One could also define "`parents`" below the rule for "`father`". This is also important because two predicates can reference each other with mutual recursion (see below). In predicate logic, there is no such distinction between definition and use.

- Derived predicates correspond to database views.
- A rule with the predicate p in the head is called a "rule about p".

  E.g. the above rule is a rule about `father`.

Rules (3)

- Names starting with a capital letter are variables: One can insert any value for a variable.

  I.e. the variables are universally quantified ("for all", $\forall$) in front of the rule. Of course, during a single rule application, one must replace different occurrences of the same variable by the same value.

- E.g. when one replaces `X` with `emil`, `Y` with `arno`, and `Z` with `birgit`, one gets:

  $$\texttt{father}(\texttt{emil}, \texttt{arno}) \leftarrow \texttt{parents}(\texttt{emil}, \texttt{arno}, \texttt{birgit}).$$

- The right hand side of the rule is true (it is given as a fact), thus the left hand side is derived.

Rules (4)

- Suppose one substitutes e.g. `X` with `emil`, `Y` with `chris`, and `Z` with `doris`:

  $$\texttt{father}(\texttt{emil}, \texttt{chris}) \leftarrow \texttt{parents}(\texttt{emil}, \texttt{chris}, \texttt{doris}).$$

- The right hand side cannot be proven, thus nothing can be derived with this rule instance (the condition is false, nothing follows about the head).

  This does not mean that the rule head must be false: There might be another rule / rule instance that permits to derive it (see below).

- Of course, Prolog and deductive databases do not simply try all possible values for the variables.

Rules (5)

- Of course, one can choose better variable names (they only have to start with an uppercase letter):

  $$\texttt{father}(\texttt{Child}, \texttt{Father}) \leftarrow \texttt{parent}(\texttt{Child}, \texttt{Father}, \texttt{Mother}).$$

- This renaming of variables does not change the meaning of the rule in any way.
- Variables are implicitly $\forall$-quantified in front of each rule. I.e. the scope of each variable is the rule.
- Two different rules can have variables with the same name, but there is no connection between them.

Multiple Rules (1)

- Of course, a predicate "mother" can be defined in the same way: $\texttt{mother}(\texttt{X},\texttt{Z}) \leftarrow \texttt{parents}(\texttt{X},\texttt{Y},\texttt{Z})$.
- One can define serveral rules about a predicate: ("parent" and "parents" are different predicate symbols.)

$$\begin{aligned}\texttt{parent}(\texttt{X},\texttt{Y}) &\leftarrow \texttt{father}(\texttt{X},\texttt{Y}).\\ \texttt{parent}(\texttt{X},\texttt{Y}) &\leftarrow \texttt{mother}(\texttt{X},\texttt{Y}).\end{aligned}$$

- Both rules can be used to derive facts about `parent`:
  - ◇ E.g. $\texttt{parent}(\texttt{emil},\texttt{arno})$ follows from the first rule

    Plus the rule about `father` and the fact $\texttt{parents}(\texttt{emil},\texttt{arno},\texttt{birgit})$.
  - ◇ and $\texttt{parent}(\texttt{emil},\texttt{birgit})$ from the second.

Multiple Rules (2)

- Suppose that in the first rule, `X` is replaced by `emil`, and `Y` by `birgit`:

$$\texttt{parent}(\texttt{emil},\texttt{birgit}) \leftarrow \texttt{father}(\texttt{emil},\texttt{birgit}).$$

- The condition (rule body) is false in the intended interpretation, and indeed, $\texttt{father}(\texttt{emil},\texttt{birgit})$ cannot be derived from the given facts and rules.
- However, the consequence (rule head) is true: $\texttt{parent}(\texttt{emil},\texttt{birgit})$ follows from the other rule.

  If the rule body is true, the head must be true, too. If the rule body is false, this rule alone does not say anything about the head (unless we know that there is no other rule about the predicate).

Multiple Body Literals

- A rule can have several conditions which are conjunctively connected (logical "and"):

$$\begin{aligned}\texttt{grandparent}(\texttt{X},\texttt{Z}) \;\leftarrow\; &\texttt{parent}(\texttt{X},\texttt{Y}) \wedge\\ &\texttt{parent}(\texttt{Y},\texttt{Z}).\end{aligned}$$

- E.g. one successful application of the rule is:

$$\begin{aligned}\texttt{grandparent}(\texttt{ian},\texttt{birgit}) \;\leftarrow\; &\texttt{parent}(\texttt{ian},\texttt{emil}) \wedge\\ &\texttt{parent}(\texttt{emil},\texttt{birgit}).\end{aligned}$$

- Both conditions in the body ("body literals") follow from the given facts and rules.
- Then this rule can be applied and permits to derive that `birgit` is grandparent of `ian`.

Constants in Rules

- Of course, one can use also constants in rules (not only variables):

$$\begin{aligned}\texttt{person}(\texttt{X},\texttt{m}) &\leftarrow \texttt{man}(\texttt{X}).\\ \texttt{person}(\texttt{X},\texttt{f}) &\leftarrow \texttt{woman}(\texttt{X}).\end{aligned}$$

- Here the second argument of the predicate indicates whether the person is male (`m`) or female (`f`).
- In this example, a constant appears in a body literal:

$$\begin{aligned}\texttt{grandmother}(\texttt{X},\texttt{Y}) \;\leftarrow\; &\texttt{grandparent}(\texttt{X},\texttt{Y}),\\ &\texttt{person}(\texttt{Y},\texttt{f}).\end{aligned}$$

Example: Summary (1)

Facts (Database):

`parents(emil, arno, birgit).`
⋮

`couple(arno, birgit).`
⋮

`man(arno).`
⋮

`woman(birgit).`
⋮

Example: Summary (1)

Rules (Derived Predicates, Views):

$$
\begin{array}{lcl}
\mathtt{father}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{parents}(\mathtt{X},\mathtt{Y},\mathtt{Z}). \\
\mathtt{mother}(\mathtt{X},\mathtt{Z}) & \leftarrow & \mathtt{parents}(\mathtt{X},\mathtt{Y},\mathtt{Z}). \\
\mathtt{parent}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{father}(\mathtt{X},\mathtt{Y}). \\
\mathtt{parent}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{mother}(\mathtt{X},\mathtt{Y}). \\
\mathtt{grandparent}(\mathtt{X},\mathtt{Z}) & \leftarrow & \mathtt{parent}(\mathtt{X},\mathtt{Y}) \land \mathtt{parent}(\mathtt{Y},\mathtt{Z}). \\
\mathtt{person}(\mathtt{X},\mathtt{m}) & \leftarrow & \mathtt{man}(\mathtt{X}). \\
\mathtt{person}(\mathtt{X},\mathtt{f}) & \leftarrow & \mathtt{woman}(\mathtt{X}).
\end{array}
$$

Queries (1)

- Given the above program ("knowledge base"), one can pose queries (goals for the theorem prover), e.g.
  - ◇ ? $\mathtt{grandparent}(\mathtt{ian},\mathtt{birgit})$.

    $\longrightarrow$ Yes.
  - ◇ ? $\mathtt{grandparent}(\mathtt{klaus},\mathtt{arno})$.

    $\longrightarrow$ No.
  - ◇ ? $\mathtt{mother}(\mathtt{frida},\mathtt{X})$.

    $\longrightarrow$ $\mathtt{X} = \mathtt{doris}$.

Queries (2)

- Example queries (proof goals), continued:
  - ◇ ? $\mathtt{mother}(\mathtt{X},\mathtt{doris})$.

    $\longrightarrow$ $\mathtt{X} = \mathtt{frida}$.
    $\mathtt{X} = \mathtt{gerd}$.
  - ◇ ? $\mathtt{mother}(\mathtt{X},\mathtt{Y})$.

    $\longrightarrow$ $\mathtt{X} = \mathtt{emil},\quad \mathtt{Y} = \mathtt{birgit}$.
    $\mathtt{X} = \mathtt{frida},\quad \mathtt{Y} = \mathtt{doris}$.
    $\mathtt{X} = \mathtt{gerd},\quad \mathtt{Y} = \mathtt{doris}$.
    ⋮ ⋮
  - ◇ ? $\mathtt{father}(\mathtt{emil},\mathtt{X}) \land \mathtt{mother}(\mathtt{emil},\mathtt{Y})$.

    $\longrightarrow$ $\mathtt{X} = \mathtt{arno},\quad \mathtt{Y} = \mathtt{birgit}$.

Queries (3)

- Syntactically, queries are the same as rule bodies (a conjunction of literals).
- Queries (Goals) are not very powerful, e.g. they normally do not permit disjunction.

  Actually, modern Prolog systems have disjunction in rule bodies and queries. However, this is not really necessary.
- However, one can extend the knowledge base with new rules that define temporary predicates. These new predicates can also be used in the query.

  SQL-99 permits to define temporary views in queries (WITH-clause).

Exercises

- Define a predicate "married_with".

  This takes the information from "couple", but should be symmetric: If X is married with Y, then Y is married with X.
- Define a predicate "siblings".

  The condition $X \neq Y$ can be used in the rule body.
- Define a predicate "uncle".
- Define a predicate for consistency checks: Is there a person which is male and female at the same time?

  Define a predicate "inconsistent" that is derivable if there is such a problem. Of course, additional predicates can be defined.

Recursive Rules (1)

- It is possible to use a predicate in its own definition:

  $\mathtt{ancestor(X, Y)} \leftarrow \mathtt{parent(X, Y)}.$
  $\mathtt{ancestor(X, Z)} \leftarrow \mathtt{parent(X, Y)} \land \mathtt{ancestor(Y, Z)}.$
- Initially, no facts about ancestor are known, thus only the first rule is applicable.
- Then, ancestor(X, Y) is known if Y is parent of X.
- This can be inserted in the second rule, and it is derived that grandparents are also ancestors.
- Another application of the second rule yields that great-grandparents are ancestors, too. And so on.

Recursive Rules (2)

- Finally, all ancestor relationships that hold in the database are derived.

  The example DB contains only three generations, so there are already no great-grandparents. But the recursion works with any number n of generations: After $n-2$ iterations, no new facts are derived.
- Of course, a recursive rule like

  $$\mathtt{p(X)} \leftarrow \mathtt{p(X)}.$$

  is useless: It never yields anything new.

  In Prolog, such a rule would actually create an infinite loop. This shows that Prolog is not an ideal logic programming language. In logic, the rule is a tautology: It is always trivially satisfied. Deductive databases can process such rules without problems.

Recursive Rules (3)

- The important point is that although one of the rules that defines "ancestor" uses "ancestor", it never refers to the same fact as it tries to prove.
- As in other programming languages, in Prolog one has to reduce the "problem size" in the recursive call, or the recursion will not come to an end.
- E.g. given the query "? ancestor(julia, birgit)", Prolog will first try the nonrecursive rule.

  Prolog tries the rules in the order they are written down. This dependence on the rule order again violates the ideal of logic programming. Deductive DBs are again better, but at the expense of performance.

Recursive Rules (4)

- Using the nonrecursive rule, Prolog has to prove parent(julia, birgit), but this fails.
- Now it uses the recursive rule. It inserts the data from the query and finds that it has to prove

$$\texttt{parent}(\texttt{julia}, \texttt{Y}) \wedge \texttt{ancestor}(\texttt{Y}, \texttt{birgit}).$$

- Thus, it first finds the parents of julia, and then processes the recursive calls:
  - ◇ ancestor(emil, birgit).
  - ◇ ancestor(frida, birgit).

Recursive Rules (5)

- The recursive call ancestor(emil, birgit) is proven with the nonrecursive rule: birgit is mother of emil.

  Thus, the answer "Yes" is printed.

- The recursive call ancestor(frida, birgit) fails.

  Prolog first tries to prove that birgit is parent of frida. This fails. Then it creates again two recursive calls by inserting frida's parents: ancestor(chris, birgit) and ancestor(doris, birgit). These immediately fail since there are no parents of chris and doris in the database.

- The problem size is reduced because every recursive call goes one generation up in the database and somewhere, there are no further data.

Recursion in SQL-99

- Ancestors cannot be computed in SQL-92 (one needs one more join for every generation).
- However, SQL-99 permits recursion:

```
WITH
    RECURSIVE ancestor(Child, Anc) AS
        SELECT Child, Par FROM parent
        UNION
        SELECT P.Child, A.Anc
        FROM   parent P, ancestor A
        WHERE  P.Par = A.Child
SELECT Anc FROM ancestor
WHERE  Child = 'julia'
```

Datalog vs. Prolog (1)

Deductive DBs Permit Less Control, More Logic:

- Order of rules not relevant.
- Order of conditions in the rule body often not relevant (depending on system/chosen optimizations).
- No cut (used in Prolog to prune the search tree).

  The cut will be explained further below. Sometimes, the cut only gives hints to the Prolog system for faster execution. But in practical Prolog programming it is used also in a way that modifies the logical meaning of the program. This violates the logic programming idea.

- Termination is guaranteed if no lists, term constructors or datatype functions are used.

Datalog vs. Prolog (2)

Reason (Database vs. Programming Language):

- The Prolog programmer knows how predicates are used (called): Which arguments are given (input), and which are variables (output).

  Closed system: There is only one main predicate, which calls all others.

- Databases allow very different queries.

  When a predicate (view) is defined, one normally does not know how it will be used in queries. Needs query optimizer.

- Query evaluation should be guaranteed to terminate, program execution cannot.

  But modern DDB systems allow the complete Pure Prolog.

Datalog vs. Prolog (3)

All Advantages of Databases:

- Presistence (Possibility to store data that live longer than a single program execution)
- Multi-User, Security, Access Control, Integrity.
- Transactions (Atomicity, Backup&Recovery, Concurrency Control).
- But current DDB prototypes have not necessarily all database functions.

Datalog vs. Prolog (4)

Deductive DBs are Better for Large Sets of Facts:

- Prolog implementations are very inefficient if the data does not fit into real main memory.
- If the data resides on disks, set-oriented evaluation techniques are better, since one anyway has to read whole blocks (e.g. Merge Join, B-Trees).

Efficiency is a Problem for Deductive DBs:

- If Prolog works for a program, the program is executed at least 10 times faster in Prolog systems than in current deductive database systems.

Overview

1. What are Prolog and Deductive DBs?
2. How to use a Prolog System
3. DDBs as Integrated Systems: DB + PL
4. Strengths of Deductive Databases
5. History, Current Problems

More about Prolog Syntax

- In Prolog, every fact (or rule) must be terminated with "." (full stop).

  It is required that the full stop is followed by white space (a space or a line break). The reason is that "." is also an operator in Prolog (list constructor). If it is used that way, it is not followed by white space.

- One should avoid spaces between the predicate and the opening parenthesis "(".

  This is used to distinguish operator syntax from the standard syntax. Operator syntax is treated in a later chapter.

- Comments in Prolog start with "%" and extend to the end of the line. (Alterative: `/* ... */` as in C.)

No Declarations in Prolog

- Prolog is a concise language: One does not have to declare predicates or constants.

  Predicates are automatically declared by writing facts or rules about them. This is a bit dangerous because typing errors might not be detected. However, most Prolog systems require at least that (1) facts and rules about one predicate are not interrupted by facts/rules about another predicate (2) at least one fact/rule exists for every predicate that is called. This gives already some protection.

- Prolog is untyped. However, many type systems have been proposed and implemented for Prolog.

  It is easy to write a type checker for Prolog in Prolog, because Prolog has good metaprogramming facilities (processing programs as data).

Rule Syntax

- In Prolog, one writes
  - ":-" instead of "$\leftarrow$",
  - a comma "," for conjunction (instead of "$\wedge$").
- E.g. the rule that defines `grandparent` is written as:

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

- Where a space is permitted, one can use newline, spaces, tabs (Prolog is free format):

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

Anonymous Variables (1)

- When a variable appears only once in a rule, its name is not important.
- Prolog then permits to use an underscore "_" instead of the variable name ("anonymous variable").

  I.e. each occurrence of the underscore stands for a new variable. Even if the underscore appears twice in a rule, it is not the same variable.
- E.g. the rule about `mother` can be written as:

```
mother(Child, Mother) :- parents(Child, _, Mother).
```


Anonymous Variables (2)

- I.e. the underscore can be used to fill in arguments that are not needed.

  This is necessary since arguments are identified by position. It corresponds to a projection in databases.
- Most Prolog systems give a warning ("singleton variables") if a non-anonymous variable appears only once in a rule.

  This is intended to catch typing errors in variables. Variables do not have to be declared, but a typing error will yield a variable that appears only once. If one wants to have a meaningful name, one can start that name with an underscore to switch off the warning.

Using a Prolog System (1)

- Write the logic program into a file, e.g. "`family.pl`".

  The extension "`.pl`" is usual for Prolog sources. Unfortunately, it is also used for Perl programs (Prolog was first!). Some Prolog systems permit to choose the extension "`.pro`" during installation. Of course, one can use any extension, but if it is not the standard extension, one later has to specify it explicitly.
- Start the Prolog system (e.g. "`pl`" under UNIX).
- It should display the prompt "`?-`".

  This means that it is in query mode. Above, "?" was used for queries.
- Read the file with the command "`[family].`".

  The brackets are an abbreviation for the built-in predicate "`consult`", e.g. "`consult(family).`". Commands are queries to special predicates.

Using a Prolog System (2)

- Do not forget the full stop "." at the end!

  Every Prolog fact, rule, query, or command must be terminated with a full stop. Otherwise, Prolog assumes that the command continues on the next line and either silently waits for more input or displays a prompt like "|". Of course, one can the still write the full stop.
- If one has to specify a path (or a filename that is not a Prolog identifier) one must put it in single quotes ' (to make it a Prolog identifier), e.g.

```
['C:/stefan/courses/lp03/examples/family.pl'].
```

- Note that the backslash "\" is usually interpreted as escape symbol, thus it must be doubled: "\\".

Using a Prolog System (3)

- If one wants to enter rules and facts interactively, one can read the special file "`user`", e.g. "`[user].`".

  The input usually ends with the UNIX end-of-file marker `Crtl+D`.

- Facts and rules can be distributed over several files, e.g. "`[myfacts,myrules1,myrules2].`"

  Most Prolog systems assume that rules about one predicate are stored consecutively in the file. If one loads another file that contain rules about the same predicate, the first rules are forgotten. Normally a warning is printed in this case. However, it is possible (depending on the system) that one reloads a file with the rules about a predicate removed, and the old rules still remain in memory (until one exists from the Prolog system). This is normally no problem, since one will not call the old predicate.

Using a Prolog System (4)

- Once facts and rules are defined, one can enter queries (from the "`?-`" prompt), e.g.

  `grandparent(julia,X).`

- Prolog prints only one solution at a time.
  - If one wants more solutions, one must press the "`;`" key (this stands in Prolog for "or").

    When there are no more solutions, Prolog will print "No". This "a tuple at a time" processing (which may also print duplicates) is also a difference to deductive databases.

  - If one does not want more solutions, one must press the "`Enter`" key.

Using a Prolog System (5)

- If a query should get into an infinite loop, one can press "`Crtl+C`".

  This normally will enter the Prolog debugger. Pressing "`a`" (for "abort") will stop the query and leave the debugger.

- One can leave the Prolog system with "`halt.`".

  "`quit`" and "`exit`" will not work in most systems. If one really wants, one can of course define them by a rule. Again: Don't forget the full stop "`.`" at the end.

- For predicates with 0 arguments (like `halt`), one does not write "`()`" in Prolog.

  "`halt().`" is a syntax error.

Using a Prolog System (6)

- Most systems have an online manual which documents at least all built-in predicates, e.g. try

  `help(consult/1).`

- Note that the number of arguments usually has to be specified.

  In Prolog, different predicates can have the same name if they have a different number of arguments. E.g. in SWI-Prolog, one can also call "`help.`" to bring up the online manual. This is documented in "`help(help/0).`".

Exercises (1)

- The Table DEPT has the columns
  - DEPTNO (Department Number),
  - DNAME (Department Name),
  - LOC (Location):

DEPT		
DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

- How would these data look as Prolog facts?

Exercises (2)

EMP					
EMPNO	ENAME	JOB	MGR	SAL	DEPTNO
7369	SMITH	CLERK	7902	800	20
7499	ALLEN	SALESMAN	7698	1600	30
7521	WARD	SALESMAN	7698	1250	30
7566	JONES	MANAGER	7839	2975	20
7654	MARTIN	SALESMAN	7698	1250	30
7698	BLAKE	MANAGER	7839	2850	30
7782	CLARK	MANAGER	7839	2450	10
7788	SCOTT	ANALYST	7566	3000	20
7839	KING	PRESIDENT		5000	10
7844	TURNER	SALESMAN	7698	1500	30
7876	ADAMS	CLERK	7788	1100	20
7900	JAMES	CLERK	7698	950	30
7902	FORD	ANALYST	7566	3000	20
7934	MILLER	CLERK	7782	1300	10

Exercises (3)

Formulate These Queries in Prolog and in SQL:

- Print number, name of the department in Boston.
- List number and name of all employees in the research department.
- List the names of all employees who are manager or president of the company.
- List all employees who earn more than their direct supervisor. One can use a condition like X > Y.
- List all employees who are directly or indirectly managed by “JONES”.

Overview

1. What are Prolog and Deductive DBs?
2. How to use a Prolog System
3. DDBs as Integrated Systems: DB + PL
4. Strengths of Deductive Databases
5. History, Current Problems

Integrated DBMS+PL (1)

General Trend:

- Persistent Programming Languages:
  Pascal/R, PJama.
- Object-Oriented Databases (C++ & Persistence)
- Stored Procedures: Oracle PL/SQL.
- Triggers in Databases (Active Databases)
- Fourth Generation Languages (4GL)
- Extensible DBMS, Universal Server.

Integrated DBMS+PL (2)

Why?

- Any serious DB-application needs programs.
- SQL not computationally complete,
  e.g. the following must be done outside SQL:
  - ◇ complex calculations,
  - ◇ complex business rules,
  - ◇ user interface,
  - ◇ interface with other systems.

Integrated DBMS+PL (3)

Problems with Embedded SQL:

- Programmer needs to work with two different languages and compilers. (Plus e.g. HTML).
- Different type system in C and SQL.
- "Impedence Mismatch": Set-oriented evaluation in SQL, tuple-oriented evaluation in C.
- The DBMS optimizer can only optimize single SQL statements in isolation (locally).
- Evaluation plans for queries in programs (external to DBMS) must be kept for the next execution.

Integrated DBMS+PL (4)

Integration is a Natural Development:

- Database management systems were used to put all data of a company under one central control:
  - ◇ to avoid redundancies and inconsistencies,
  - ◇ to share the data, software, hardware, support,
  - ◇ to simplify the development of new applications.
- Now is the time to do the same with procedures/programs!

Integrated DBMS+PL (5)

A Subprogram-Library is not the Same:

- If a procedure in the library is changed, all application programs must be linked with the new version.
- If the DB schema is changed, possible problems (e.g. missing attributes or relations) will only be detected when the critical program is executed.
- Application programs and the relations that they access are not contained in the data dictionary.
- Such procedures cannot be used in queries or triggers. They exist only outside the DBMS.

Deductive Databases

- In contrast to other DBMS+PL-systems, deductive databases use a declarative programming language.
- Declarativity is nothing new in databases: SQL is a declarative query language.
- All other competitors give up declarativity and set-orientation for queries, or have a gap between database language and programming language.
- In deductive databases the integration is so tight that it is usually not possible to say where the query ends and the program begins.

Overview

Deductive DBs: Features (1)

Recursion:

- Important for processing hierarchical data, and data that has the form of a graph.
- E.g. the WWW is a directed graph of documents.
- Documents are hierarchically structured, e.g. XML defines tree-structure (plus `IDREF`).

  "Object Exchange Model", Models for semistructured data: graphs.
- CAD: Parts are often hierarchically composed out of smaller parts.
- CASE: Call structure of procedures is a graph.

Deductive DBs: Features (2)

Rules:

- Expert Knowledge.
- For regularly structure data (e.g. bus schedules).
- Syntax rules.
- Integrity constraints.
- Knowledge Independence.

  Data independence means that data is independent of application programs. Now one wants to centrally collect more knowledge about the application domain. Currently, this knowledge is only buried in application program code.

Deductive DBs: Features (3)

Views:

- Views/derived predicates are *the* basic concept of deductive databases. In relational databases they are only an addition.
- Deductive DBs allow also views which can only be called with certain input arguments.

  Important for computations, interfaces to external programs.
- Views are important for integrating several heterogeneous data sources (DDBs as "middleware").

Deductive DBs: Features (4)

Views, Continued:

- Analysis of large data sets with more abstract concepts ("Data Mining", "Decision Support").
- Storing repeating queries.

  → collection of knowledge.
- Plus all standard advantages of views:
  - ◇ Adaption of the schema to the wishes of different users,
  - ◇ logical data inpedendence,
  - ◇ security.

Deductive DBs: Features (5)

Complex Objects:

- Lists, variant records, trees can be expressed as Prolog terms and easily be processed in Prolog.

Pattern Matching:

- Prolog has terms with variables (data structures with holes).

Several Solutions / Backtracking:

- Patterns can appear in texts several times.
- Hypothetical reasoning: Exploring alternatives.

Overview

1. What are Prolog and Deductive DBs?
2. How to use a Prolog System
3. DDBs as Integrated Systems: DB + PL
4. Strengths of Deductive Databases
5. History, Current Problems

History of the Field (1)

~322 BC	Syllogisms [Aristoteles]
~300 BC	Axioms of Geometry [Euklid]
~1700	Plan of Mathematical Logic [Leibniz]
1847	"Algebra of Logic" [Boole]
1879	"Begriffsschrift" (Early Logical Formulas) [Frege]
~1900	More natural formula syntax [Peano]
1910/13	Principia Mathematica (Collection of formal proofs) [Whitehead/Russel]
1930	Completeness Theorem [Gödel/Herbrand]
1936	Undecidability [Church/Turing]

History of the Field (2)

1960	First Theorem Prover [Gilmore/Davis/Putnam]
1963	Resolution-Method for Theorem proving [Robinson]
~1969	Question Answering Systems [Green et.al.]
1970	Linear Resolution [Loveland/Luckham]
1970	Relational Data Model [Codd]
~1973	Prolog [Colmerauer, Roussel, et.al.] (Started as Theorem Prover for Natural Language Understanding) (Compare with: Fortran 1954, Lisp 1962, Pascal 1970, Ada 1979)
~1973	Algorithm = Logic + Control [Kowalski]

History of the Field (3)

1976	Minimal Model Semantics [van Emden, Kowalski]
1977	Conference "Logic and Databases" [Gallaire, Minker]
1977	First Compiler for Prolog [Warren]
1982	Start of the "Fifth Generation Project" in Japan (ended 1994)
1986	"Magic Sets"
1986	Perfect Model Semantics
1986	First Deductive DB Systems
1987	CLP(R): Arithmetic Constraints [Jaffar]

History of the Field (4)

1988	CHIP: Finite Domain Constraints [Van Hentenryck]
1988	Well-Founded and Stable Model Semantics
~1989	First Textbooks on Deductive DBs
~1992	Second DDB Prototype System Generation
1996	ISO Standard for Prolog
1996	smodels: Answer Set Programming System [Nimelä/Simons]

Deductive DBs: Problems (1)

- There are very many research papers.
- There are some prototype implementations. However, they are used more or less only in academia (with some exceptions).
- There is no commericial deductive DBMS at the moment (that is sold for general use).
- One company producing a DDBMS had to close.

  Furthermore, at least one project aiming at a commerical DDBMS was significantly modified in its goals.

Deductive DBs: Problems (2)

- In a panel discussion on ICDE'93 ("Are we polishing a round ball?") the area of recursive query evaluation was declared as no longer interesting.
- Ramakrishnan wrote 1995 that better implementations will solve the current problems.

  Thus, in his view, there is no need for fundamentally new techniques, i.e. more research.
- Ullman told me on PODS'99 that the area has no future.

Deductive DBs: Problems (3)

- There are not so many grants in this area anymore.

  It is difficult to find professorships. More and more researchers move to other areas.
- Prolog-Implementations are still much faster that deductive databases (more than one order of magnitude).
- Whereas Prolog-compiler are typically written in Prolog, Datalog is not used in implementing deductive databases.

Reasons Not to Give Up (1)

- Prolog is being successfully used in industry.

  Boeing, British Airways, Kodak, Swiss Life, IBM (Machine Translation), Philips Research, SRI (natural language).
  Microsoft Windows NT uses a small Prolog interpreter in order to generate optimal configurations for networks. Knowledgeware's Application Development Workbench (a CASE tool, 1995: 60.000 Licenses sold) contains around 250.000 lines of Prolog code. The company estimates that it would have been around 10 times larger if written in C. [Frühwirth/Abdennadher 1997]

- At Ericsson a special deductive DBMS for their own use has been developed.

Reasons Not to Give Up (2)

- Constraint Logic Programming is very successful in industry.

  It is estimated that 1996 constraint technology for 100 million $ was sold. (1996, data mining tools for 120 million $ were sold. Microsoft's turnover was 10.000 million $.) [Frühwirth/Abdennadher 1997]

- The logic programming community is still alive.

  Many well-known database researchers have worked in the area of deductive databases.

- Many ideas of deductive databases are now being used in extensions of standard RBMS (recursion in SQL-99, magic sets in DB2).

Reasons Not to Give Up (3)

- Obviously, more research is necessary to ensure that Datalog can really be used as a programming language, not only as a query language.

  The goals and promises of deductive databases are not yet reached, because the problems turned out to be more complex than expected. But when they will be reached, deductive DBs are very attractive.

- Object-oriented databases turned out to be a step backwards in some respects. This lead to object-relational systems, but DOOD-systems might be even better.
- Deductive DBs and SQL do not exclude each other.

Reasons Not to Give Up (4)

- Query languages for the Web and for XML data, as well as intelligent agents in the Web open new and fruitful application areas.
- The "semantic web" needs artificial intelligence.
- Answer set programming is a new logic programming formalismn that was successfully used in a number of applications.
- Once more students who have learnt about deductive databases work in industry, the acceptance will be greater.