Logic Programming and Deductive Databases

Chapter 11: Datalog with Built-In Predicates

Prof. Dr. Stefan Brass
Martin-Luther-Universität Halle-Wittenberg
Summer 2022

http://www.informatik.uni-halle.de/~brass/lp22/

Objectives

After completing this chapter, you should be able to:

- check the allowedness of a clause.
- explain how function symbols could be implemented with binding patterns.

Contents

1. Datalog
2. Formal Treatment of Built-In Predicates
3. Range-Restriction, Allowedness
4. Function Symbols and Built-In Predicates

Basic Datalog (1)

- Datalog is an adaption of Prolog for database applications.
- The main differences are:
  - No extra-logical constructs are used
    (e.g., no cut, no built-in predicates with side effects).
  - There is no prescribed execution sequence.

    It is the task of the query optimizer to find a good evaluation sequence.

    Of course, this excludes built-in predicates with side-effects.
  - No function symbols are allowed (e.g., no lists).

    I.e. terms are variables or constants.
  - Rules must be range-restricted, i.e. every variable in the rule head appears also in the body.

    The notion of range-restriction is discussed in much more detail below.

    It ensures that all rule applications yield ground atomic formulas (facts).

Basic Datalog (2)

- Advantages of Datalog:
  - It is a "more declarative" language than Prolog.
    The user does not have to think about execution algorithms.

    E.g., in Prolog, the programmer has often to order the rules about a predicate in a specific sequence. In Datalog, the rule order is meaningless. Also, the sequence of body literals is not important (for good optimizers).
  - Termination is guaranteed.

    E.g., the classic "ancestor" rules permit to compute the transitive closure also of cyclic graphs. In Prolog, the programmer has to do some bookkeeping which nodes were visited. That significantly complicates the program.
  - By not using structured terms, the language becomes extremely simple.

    In classical relational databases, there are no structured terms.

Basic Datalog (3)

- While both languages (Datalog and Prolog) use the same formalism of definite Horn clauses, the programmer thinks quite differently when writing rules:
  - The Prolog programmer thinks in predicate calls, and sees the rules executed from left (head) to right (body).
  - The Datalog programmer thinks in derivable facts, and envisions the rules executed from right (body) to left (head).

    The direction from right to left might seem strange at first, but it is the direction of the implication arrow.
- Prolog is executed with SLD-resolution.
- In Datalog, one can use any method to compute the minimal model, e.g. iterate the T_P-operator (one needs only the part of the minimal model that is relevant for the query).

Repetition of Definitions

- A Datalog term t is a constant or a variable.
- An atomic formula A has the form $p(t_1, \ldots, t_k)$ and consists of a predicate p (of arity k) and terms t_i, $i = 1, \ldots, k$.

  p is called the predicate of the atomic formula, we write $pred(A)$ for the predicate of A. t_i is called the i-th argument of the atomic formula.
- A Datalog rule has the form $A \leftarrow B_1, \ldots, B_n$ with atomic formulas A and B_j, $j = 1, \ldots, n$, which satisfies the condition of range-restriction (below).

  As long as there are no built-in or negated predicates, range-restriction simply means that every variable that appears in A must appear in at least one B_j.
  A is called the head of the rule and $B_1 \wedge \cdots \wedge B_n$ is its body.
  If p is the predicate of A, this rule is called a rule about p.
- A Datalog program is a finite set of Datalog rules.

Datalog Variants

- There are many extensions of this very basic version of Datalog.

  When you read a scientific paper, check what the author actually means by Datalog. Every Datalog version should include the above (definite Horn clauses without function symbols), but there are also more powerful versions.

- Datalog$^{\neg}$ contains also negation.

  Maybe only stratified negation, maybe more. See Chapter 15.

- There are also variants with aggregation.
- Datalog^{fun} contains also function symbols.
- Datalog$^{\pm}$ permits existential variables in the head, together with some restrictions that make it decidable.

  This is a whole family of languages. They permit to formalize ontologies, and can be seen as a syntactic variant of description logics.

Contents

1 Datalog

2 Formal Treatment of Built-In Predicates

3 Range-Restriction, Allowedness

4 Function Symbols and Built-In Predicates

Built-In Predicates in Datalog

- Built-in predicates with side-effects, e.g. for input/output, are not studied here. These do not exist in Datalog.

  There are other options for declarative output.

- Typical built-in predicates that are used in Datalog are the comparison operators $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, as well as some support for arithmetics, e.g. `X = Y + Z`.

  This can be seen as a user-friendly notation for the predicate `sum(Y,Z,X)` that was studied in Chapter 6. In Prolog, one would have to write `is`, in order to distinguish evaluation of arithmetic expressions from unification. However, Datalog has no unification, or only a very restricted form of unification called "matching": One can replace a variable from the rule by a constant from a given fact, or can check that a constant from the rule is equal to the constant in the corresponding argument of a fact. In this way, body literals are matched with facts.

Semantics of Programs (1)

Formal Treatment of Built-In Predicates:

- Let a fixed interpretation $\mathcal{I}_B$ be given that defines the extensions $\mathcal{I}_B[p]$ of all built-in predicates p.

  Of course, this interpretation must also define the domain. Typically, one considers only Herbrand interpretations. Then the base interpretation defines the domain, the meaning of the function symbols, and the meaning of the built-in predicates.

- Then one considers only interpretations $\mathcal{I}$ that are extensions of $\mathcal{I}_B$, i.e. all interpretations must agree with $\mathcal{I}_B$ on the signature for which $\mathcal{I}_B$ is defined.

Semantics of Programs (2)

- One identifies an interpretation $\mathcal{I}$ with the set of facts $p(t_1, \ldots, t_n)$ with $\mathcal{I} \models p(t_1, \ldots, t_n)$, where p is not a built-in predicate.

  This is an extension of the corresponding convention for Herbrand interpretations. Since built-in predicates nearly always have an infinite extension, excluding them increases the chances that the set of facts is finite (and thus can be explicitly written down or explicitly stored).
  Of course, also predicates defined by rules can have infinite extensions.

- As before, the semantics of a program P is the least fixed point of the T_P-operator (see next slide).

  This is the least model of P among the interpretations $\mathcal{I}$ that extend $\mathcal{I}_B$.

Semantics of Programs (3)

- The definition of the immediate consequence operator T_P does not have to be modified:

$$T_P(\mathcal{I}) := \{F \in \mathcal{B}_\Sigma \mid \text{There is a rule } A \leftarrow B_1 \wedge \cdots \wedge B_n \text{ in } P \text{ and a ground substitution } \theta, \text{ such that } \bullet\ \mathcal{I} \models B_i\theta \text{ for } i = 1, \ldots, n, \text{ and } \bullet\ F = A\theta\}.$$

- Since the head literal A does not contain a built-in predicate, one gets only facts about user-defined predicates.

Valid Binding Patterns (1)

- The designer of the logic programming system defines for each built-in predicate p a set $valid_B(p)$ of valid binding patterns such that for all built-in predicates p and all $\beta \in valid_B(p)$:
  - Suppose that p has arity n and let
    $$\{i_1, \ldots, i_k\} := \{i \mid 1 \leq i \leq n,\ \beta_i = \mathtt{b}\}.$$
  - Then for all domain elements $d_{i_1}, \ldots, d_{i_k}$ in $\mathcal{I}_B$, the set
    $$\{(d'_1, \ldots, d'_n) \in \mathcal{I}_B[p] \mid d'_{i_1} = d_{i_1}, \ldots, d'_{i_k} = d_{i_k}\}$$
    must be finite and computable.

    I.e. the requirement is that given any values for the bound arguments, it must be effectively possible to compute values for the other arguments, and to compute all such solutions.

Valid Binding Patterns (2)

- Together with the range-restriction defined below, this ensures that each single application of the T_P-operator is computable and has a finite result.
  - Of course, in the limit (minimal model), it is still possible that user-defined predicates have infinite extensions.

    Which is also bad, because it means that the computation does not terminate.
  - However, each approximation $T_P \uparrow n$ of the minimal model can be effectively computed.

Contents

1 Datalog

2 Formal Treatment of Built-In Predicates

3 Range-Restriction, Allowedness

4 Function Symbols and Built-In Predicates

Motivation (1)

- In deductive databases, query evaluation is done by applying the T_P-operator iteratively to compute the minimal model (with certain optimizations).

  Computing the entire minimal model would not be goal-directed.
  Of course, one should compute only facts that are important for the query.
  This problem is solved by the magic set transformation: Given a logic program and a query, it computes a new logic program that has the same answer, but implies only facts relevant to the query. See Chapter 12.

- The allowed rules must be restricted so that immediate consequences can effectively be computed.

Motivation (2)

- For example, computing immediate consequences for a rule like the following would be difficult:

$$p(X, Y) \leftarrow q(X).$$

- The possible values for Y depend on the domain: All data values can be inserted, often this set infinite, and maybe not even explicitly known.

  Normally, one works with the Herbrand universe that consists of all terms which can be constructed from the constants and function symbols appearing in the program. Then one can add a completely unrelated fact, in which a new constant appears, and thereby change the extension of p. That is a strange behaviour.

Motivation (3)

- Given e.g. $q(a)$, one could derive the "fact" $p(a, Y)$.
- One problem with this is that variables cannot be easily represented in database relations.

  As long as one does not use function symbols, derived predicates should correspond to views in relational databases. Furthermore, at least some prototypes did actually use a relational database system for query evaluation: Then storing an intermediate result in a temporary relation for p is at least difficult.

- In contrast to Prolog, deductive databases normally use only one-directional, restricted form of unification ("matching"): Variables appear only in rules, body literals are matched with variable-free facts.

Allowed Rules (1)

Definition (Allowed Rule, First Try):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is called allowed iff every variable that appears in the head literal A appears also in at least one body literal B_i.

Note:

- This definition works only if the body literals have no binding restrictions.
- E.g. one cannot compute all consequences of the following rule, although it statisfies the condition:

```
less(X, Y) :- X < Y.
```

Allowed Rules (2)

Definition (Allowed Rule with Built-In Predicates):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is called allowed iff every variable that appears in the rule appears also in at least one body literal B_i, the predicate of which is not a built-in predicate.

Example:

- E.g. the following rule satisfies this condition:

```
teenager(X)  :-  person(X, BirthYear),
                 BirthYear < 2012,
                 BirthYear > 2002.
```

Allowed Rules (3)

Note:

- If all rules are allowed in the above sense, one can effectively compute every approximation $T_P \uparrow i$ of the minimal model.

  Assuming that $valid_B(p)$ is not empty for every built-in predicate p. Built-in predicates without any valid (implemented) binding pattern obviously make no sense.

- If in addition, the rules do not contain function symbols, the minimal model itself can be computed.

  It contains then only constants that appear in the program. Assuming that programs are always finite, this means that the minimal model is finite, and is reached after finitely many iterations of the T_P-operator.

Range Restriction (1)

- The notion of "allowed rule" above assumes that we really want to compute the entire extension of all derived predicates.

  I.e. that all predicates should support the binding pattern `f...f` like stored relations. This is not always required.

- Predicates that have binding restrictions are typically defined by rules that are not allowed:

```
append([], L, L).
append([F|R], L, [F|RL]) :-
      append(R, L, RL).
```

  append is called only with binding patterns `bbf` and `ffb` (or the more specialized binding patterns `bfb`, `fbb` and `bbb`). However, we never compute the entire extension of append (corresponding to the binding pattern `fff`).

Range Restriction (2)

- The rule about `less` makes sense if it is called with binding pattern `bb`:

```
less(X, Y) :- X < Y.
```

- Occurrences of variables in literals with built-in predicates can act as binding. This rule defines a predicate without any binding restriction:

```
price_with_vat(Prod, X) :- product(Prod, Price),
                           X is Price * 1.19.
```

- All this shows that the allowedness requirement is too restrictive.

Range Restriction (3)

Definition (Input Variables):

- Given a literal $A = p(t_1, \ldots, t_n)$ and a binding pattern $\beta = \beta_1, \ldots, \beta_n$ for p, the set of input variables of A with respect to β is

$$input(A, \beta) := \bigcup \{vars(t_i) \mid 1 \leq i \leq n,\ \beta_i = \mathtt{b}\}$$

(i.e. all variables that appear in bound arguments).

Note:

- Input variables in body literals must be bound before the literal can be called. Input variables in head literals are bound when the rule is executed.

Range Restriction (4)

Definition (Valid Binding Patterns):

- Let *valid* be a function that maps every predicate p to a set $valid(p)$ of binding patterns for p.
- For built-in predicates p: $valid(p) = valid_B(p)$.

  I.e. *valid* extends $valid_B$ from built-in predicates to all predicates.
- *valid* is called a valid binding pattern specification.

Remark:

- We assume that the programmer defines valid binding patterns for every predicate.

  In practice, it might be possible to compute the possible binding patterns, but that would complicate the next definition.

Range Restriction (5)

Definition (Range-Restricted Rule):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is range-restricted for a binding-pattern β iff there is a sequence $i_1, \ldots, i_n$ of the body literals (i.e. $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$) such that
  - for every $j \in \{1, \ldots, n\}$ there is a binding pattern $\beta_j \in \mathit{valid}(\mathit{pred}(B_{i_j}))$ with
    $$\mathit{input}(B_{i_j}, \beta_j) \subseteq \mathit{input}(A, \beta) \cup \mathit{vars}(B_{i_1} \wedge \cdots \wedge B_{i_{j-1}})$$
  - and furthermore it holds that
    $$\mathit{vars}(A) \subseteq \mathit{vars}(B_1, \ldots, B_n) \cup \mathit{input}(A, \beta).$$

Range Restriction (6)

- The definition assumes the following evaluation:
  - First, variables in the bound argument positions of the head literal are assigned values (based on the input arguments of the predicate call).
  - Then the body literals are evaluated in some order. For each literal, variables in bound argument positions must already have a value. Variables in other argument positions get a value by this call.
  - In the end, a tuple is produced that corresponds to the head literal. Therefore, all variables in the head literal must now have a value.

Range Restriction (7)

- The evaluation sequence of body literals may depend on the binding pattern for the head literal.
- For instance, consider the following rule:

```
p(X,Y) :- sum(X,1,Z), prod(Z,2,Y).
```

- The given sequence of body literals is possible for the binding pattern `bf`.
- For the binding pattern `fb`, the system should automatically switch the sequence of body literals.

  The Datalog programmer does not necessarily know the binding pattern.

  Furthermore, it would be bad style to double the rule.

Range Restriction (8)

Definition (Range-Restricted Program):

- A program P is range-restricted with respect to a binding pattern specification *valid* iff for every rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ and every binding pattern $\beta \in \mathit{valid}\big(\mathit{pred}(A)\big)$, the rule is range-restricted for β.
- A program P is strictly range-restricted iff every rule in P is range-restricted for the binding pattern `f...f`.

  Strict range restriction is the requirement for the T_P-operator to be directly executable. As we will see, the magic set transformation turns a range-restricted program into a strictly range-restricted program.

Contents

1 Datalog

2 Formal Treatment of Built-In Predicates

3 Range-Restriction, Allowedness

4 Function Symbols and Built-In Predicates

Record Constructors (1)

- Function symbols in Prolog are record/structure constructors.
- Let `cons(`E`, `N`, `L`)` be a built-in predicate for managing nodes L in a linked list (records with two components: list element E and "next" pointer N).
- It can be called with two binding patterns:
  - `bbf`: For constructing a list node.
  - `ffb`: For selecting the components of a list node.
- Note that `cons(`E`, `N`, `L`)` actually means L=`[`E`|`N`]`.

Record Constructors (2)

- Consider again the definition of append:

```
append([], L, L).
append([F|R], L, [F|RL]) :-
        append(R, L, RL).
```

- Instead of using composed terms like `[F|R]`, one can also use the built-in predicate `cons`:

```
append([], L, L).
append(L1, L2, L3) :-
        cons(F, R, L1),    % Split L1 into F and R
        append(R, L2, RL),
        cons(F, RL, L3).   % Compose F and RL to L3
```

Record Constructors (3)

- The right sequence of body literals in the above rule depends on the binding pattern for the predicate.
- As written down, the rule works if `append` is called with binding pattern `bbf`.
- If it is called e.g. with binding pattern `ffb`, the body literals should be (automatically) reordered:

```
append(L1, L2, L3) :-
        cons(F, RL, L3),
        append(R, L2, RL),
        cons(F, R, L1).
```

Record Constructors (4)

- If for every function symbol f of arity n, one has a built-in predicate p_f of arity $n+1$ that constructs/splits records of type f, composed terms are not strictly necessary:
  - As shown above for `append`, one can always replace them by a new variable and a call to the built-in predicate.
- Of course, this assumes that there are no terms with "holes" (variables) in them.

  In deductive databases, this is normally the case.

Evaluable Functions (1)

- Conversely, one could use functional notation for certain built-in predicates.
- E.g. consider

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
        N > 1,
        N1 is N-1, N2 is N-2,
        fib(N1, F1), fib(N2, F2),
        F is F1+F2.
```

Evaluable Functions (2)

- One could now write the rule as:

```
fib(N, F)  :-  N > 1,
               fib(N-1, F1), fib(N-2, F2),
               F is F1+F2.
```

- Note: This is not correct in Prolog (typical beginner's error).
- Even the following would be possible:

```
fib(N, fib(N-1)+fib(N-2)) :- N > 1.
```

- Or better still:

```
fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1)+fib(N-2) :- N > 1.
```

A preprocessor could translate this back to the standard predicate notation.

Evaluable Functions (3)

- So, why has Prolog only non-evaluable functions (record constructors)?
- Record constructors are uniquely invertable.
- Consider e.g. the following rule:

  ```
  p(X+Y, X, Y).
  ```

  Compare it with this rule:

  ```
  q([X|Y], X, Y).
  ```

- The first rule does not support the binding pattern `bff`, the second does support it.

Evaluable Functions (4)

- As long as one has only record constructors, every occurrence of a variable in a bound argument position in the head defines a value for the variable.
- For evaluable functions, this is not necessarily the case. But e.g. the following rule supports `bf`:

```
p(X+1, X).
```

- However, new logic programming languages are still being proposed, and everybody is free to define (and implement) his/her own language.

There are many proposals for combined logic-functional languages.