**UNIVERSITÄT DORTMUND**
Fachbereich Informatik

Lehrstuhl VIII
Künstliche Intelligenz

# Optimizing Chain Datalog Programs and their Inference Procedures

LS–8 Report 20

**Anke Rieger**

Dortmund, February 27, 1996

Universität Dortmund
Fachbereich Informatik

University of Dortmund
Computer Science Department

Forschungsberichte des Lehrstuhls VIII (KI)
Fachbereich Informatik
der Universität Dortmund

ISSN 0943-4135

Anforderungen an:

Universität Dortmund
Fachbereich Informatik
Lehrstuhl VIII
D-44221 Dortmund

Research Reports of the unit no. VIII (AI)
Computer Science Department
of the University of Dortmund

ISSN 0943-4135

Requests to:

University of Dortmund
Fachbereich Informatik
Lehrstuhl VIII
D-44221 Dortmund

e-mail: reports@ls8.informatik.uni-dortmund.de
ftp: ftp-ai.informatik.uni-dortmund.de:pub/Reports
www: http://www-ai.informatik.uni-dortmund.de/ls8-reports.html

# Optimizing Chain Datalog Programs and their Inference Procedures

LS–8 Report 20

**Anke Rieger**

Dortmund, February 27, 1996

Universität Dortmund
Fachbereich Informatik

**Abstract**

We present methods for optimizing chain Datalog programs by restructuring and post-processing. The rules of the programs define intensionally a set of target concepts, which are to be derived via forward chaining. The restructuring methods transform the rules, such that redundancies and ambiguities, which prevent efficient evaluations, are removed without changing the coverage of the target concepts. The post-processing method increases the coverage by introducing recursive rules in the chain Datalog program. Based on the correspondence between chain Datalog programs and context-free languages, which in our case reduce to regular ones, we present a method to map restructured and/or post-processed programs to prefix acceptors, which are deterministic finite state automata, whose input/output alphabets consist of predicates. We present an efficient marker passing method which is applied to a prefix acceptor, and which optimizes inferences. We proof that this method is sound and complete, i.e., it calculates the minimum Herbrand model of the chain Datalog program which has been mapped to the respective prefix acceptor. As the developments, presented in this paper, have been motivated by an ILP application to robotics, we have applied the methods to this real-world domain. The experimental results at the end of the paper reflect the improvements, we have gained.

# Contents


**1 Introduction** **3**

**2 The Robotics Domain** **4**

**3 Logic Programming Concepts** **6**
- 3.1 Definitions . . . . . 6
- 3.2 Semantics of logic programs . . . . . 8
- 3.3 Correspondence between chain Datalog programs and CFG's . . . . . 9
- 3.4 Non-elementary chain Datalog rules . . . . . 10
- 3.5 Constraints . . . . . 11

**4 Structuring Chain Datalog Rules in Prefix Acceptors** **12**
- 4.1 Sorting the premise literals of chain Datalog rules . . . . . 13
  - 4.1.1 The `sort`-method . . . . . 14
  - 4.1.2 The `sort_dc`-method . . . . . 16
  - 4.1.3 Related work . . . . . 18
- 4.2 The `prefix tree`-method . . . . . 18
- 4.3 Restructuring chain Datalog programs . . . . . 22
  - 4.3.1 The `restruct`-method . . . . . 22
  - 4.3.2 The `restruct_dc`-method . . . . . 25
  - 4.3.3 Equivalence of the restructured program . . . . . 28
  - 4.3.4 Mapping the restructured program to a prefix acceptor . . . . . 29
  - 4.3.5 Related work . . . . . 30

**5 MP: An Efficient Forward Inference Method** **31**
- 5.1 The marker passing method . . . . . 31
- 5.2 Soundness and completeness . . . . . 35

**6 Post-Processing Chain Datalog Programs** **41**
- 6.1 Disadvantages of the rules learned for the robotics domain . . . . . 42
- 6.2 Post-processing the prefix acceptor: Step 1 . . . . . 43
- 6.3 The post-processed chain Datalog program . . . . . 46
- 6.4 Experiments . . . . . 48

**7 Restructuring, Marker Passing and Decompositions** **50**

**8 Conclusions** **58**
- 8.1 Summary . . . . . 58
- 8.2 Current and future work . . . . . 59

**A Appendix** **61**
- A.1 Algorithm `prefix_tree` . . . . . 61
- A.2 Algorithm `restruct_dc` . . . . . 62
- A.3 Auxiliary functions . . . . . 63
- A.4 Marker Passing: Example run on $PA'$ . . . . . 64

A.5 Post-Processing: Experimental Results . . . . . . . . . . . . . . . . . . . . . . 66
A.5.1 Complexity Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
A.5.2 Testing Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

# 1 Introduction

In this paper, we present methods for optimizing chain Datalog programs by restructuring and post-processing. The rules of these programs define intensionally a set of (learned) target concepts. They contain many redundancies, which are not superfluous in the sense that they can simply removed, but which cause (forward) inference procedures to become rather inefficient. Improvements of both, the programs and the inference procedures, are extremely important as the rules are used in a robot application to derive higher-level concepts from sensor observations in real-time.

Our restructuring methods transform a program without changing the coverage of the original target concepts. They use inverse resolution (see, e.g., [14], [21], [27]), i.e., they implement the W-operator (see [14]) as inter-construction for chain Datalog rules. Thus, our approach is closely related to the one proposed by Sommer [23]. However, his method `FENDER` does not yield the result we need. During the restructuring process new predicates are invented. We combine pairs of existing terms into a new combined term. As our main goal for introducing new concepts is to speed up inferences, our approach to concept formation differs from the demand-driven one proposed by Wrobel [28].

During the post-processing phase, some new concepts are merged according to criteria, which have to be specified by the user. The post-processing method performs a generalization step, which increases the coverage of the original target concepts.

In order to optimize the inference procedure, we use prefix acceptors, which are deterministic finite state automata whose input/output alphabets consist of predicates, and to which we apply a marker passing method. Given a chain Datalog program (original, restructured, or post-processed), we present two methods, which map it to a prefix acceptor. The first one structures the rules of the original, non-recursive program in a tree, which is then mapped to an acceptor. The second one maps any linear chain Datalog program to a prefix acceptor. The marker passing method is an efficient inference procedure, which derives all possible instances of the target concepts via forward inferences. We have proven, that this method is sound and complete, i.e., it calculates (part of) the minimum Herbrand model of the program, which has been mapped to the prefix acceptor. We show the relation between mapping chain Datalog rules in a prefix acceptor and marker passing, on one hand, and decompositions of chain Datalog programs for query optimization, on the other hand [7]. In principle, our approach to optimizing chain Datalog programs and their inference procedures can also be considered as an efficient implementation of the theoretical concepts introduced by Dong and Ginsburg [7]. The practical relevance of the methods is shown by their successful application to the robotics domain, which was developed in the BLearn-project.

In Section 2, we give a short overview of the robotics domain, which motivated most of the developments presented in this paper. We use examples from this domain throughout the paper in order to illustrate the methods. In Section 3, we define the logic programming concepts, which we need to characterize the syntax and semantics of chain Datalog programs. We also show the correspondence between chain Datalog programs and CFGs ([24]), as we make extensive use of CFGs, in order to illustrate the basic ideas of our methods. In Section 4, we present the restructuring methods as well as the methods, which map a chain Datalog program to a prefix acceptor. The marker passing method is explained in Section 5. Section 6 describes the post-processing method and results of the application of the methods to the robotics domain. In Section 7, we elaborate the relation between our methods and program decompositions. We conclude with a summary and comments on ongoing and future work in Section 8.

# 2 The Robotics Domain

Starting point for the work presented in this paper are operational concepts, which have been introduced in [10]. On one hand, operational concepts can be used to specify high-level plans for robot navigation. On the other hand, they are symbolically grounded in robot perceptions and actions, i.e., they can be derived from sensor measurements and elementary actions. This derivation is accomplished in several inference steps, which are reflected by the abstraction hierarchy in Figure 1. Operational concepts can be used to specify the domain knowledge about a specific type of environment (e.g., office buildings), in which the robot is to navigate. Given this domain knowledge, plan recognition systems [18] can be used to reason about what kinds of actions might be supported by an observation, and about what kinds of actions might be performed in order to achieve a goal. This process involves chaining forward from the observations and backwards from the goal, and terminating when the two chains intersect. We first consider the forward chaining part,

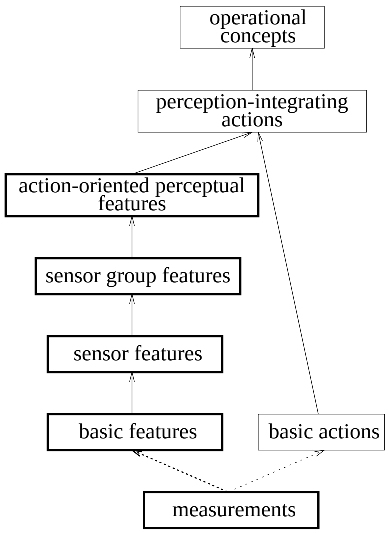


Figure 1: Abstraction hierarchy

i.e., the left side of the abstraction hierarchy, which accounts for the bottom-up derivation of perceptual features. The (forward) inference steps are indicated by the non-dashed arcs. Each arc connects two levels of the abstraction hierarchy. For each inference step, rules have been learned, such that concepts represented at the level, from which an arc emanates, appear in the premise of a rule, and concepts, which are represented at the level at the end of the arc, appear in the conclusion of a rule. An example of a rule[1], which derives *action-oriented perceptual features* from *sensor group features* is the following:

[1]We use a Prolog-like notation, i.e., variables begin with capital letters, constants with small letters.

```
through_door(Trace,Start,End,parallel) <-
  sg_jump(Trace,left,T1,T2,parallel) & sg_jump(Trace,right,T1,T2,parallel)
  & Start < T1 & T2 < End.
```

It states, that the robot moved `parallel`y through a doorway in a `Trace` during the interval from time point `Start` to `End`, if, during a subinterval, the sensors on the robot's `right` and `left` side perceived the edge grouping `jump`. *Sensor group features* are derived, if sufficiently many sensors, which are adjacent and belong to the same class, have perceived the same edge grouping:

```
sg_jump(Trace,right,TS,TE,parallel) <-
  s_jump(Trace,Sensor1,TS,TE,parallel) &
  s_jump(Trace,Sensor2,TS,TE,parallel) & adjacent(Sensor1,Sensor2) &
  sclass(Trace,Sensor1,T1,T2,right) & sclass(Trace,Sensor2,T1,T2,right) &
  T1 < TS & End < TE.
```

This rule states, that the sensors at the robot's `right` side perceive a `jump` during the time interval form `TS` to `TE` during which the robot moves `parallel`y along it, if at least two sensors, which belong to the class `right` perceived this grouping. An example of a rule, which derives sensor features from basic features is

```
s_jump(Trace,Sensor,X,Y,parallel) <-
    stable(Trace,Or,Sensor,X,X1) & incr_peak(Trace,Or,Sensor,X1,X2) &
    stable(Trace,Or,Sensor,X2,Y).
```

It states, that a sensor `Sensor` has perceived a `jump` in trace `Trace`, if it first perceived `stable` measurements during the time interval `X` to `X1`, an `incr_peak` between the successive time points `X1` and `X2`, and finally `stable` measurements during the interval from `X2` to `Y`, while moving `parallel`y along it. We rewrite these rules, in such a way, that we get rules, which are free of constants. For our example, we get

```
s_jump_parallel(Trace,Sensor,X,Y) <-
    stable(Trace,Or,Sensor,X,X1) & incr_peak(Trace,Or,Sensor,X1,X2) &
    stable(Trace,Or,Sensor,X2,Y).
```

The predicates, which appear in the head of these rules, are sensor feature predicates, which can be characterized as

$$sf(\underline{tr}, \underline{s}, \underline{from}, \underline{to}),$$

where $sf$ denotes a predicate symbol, which describes an object, which has been perceived during a trace, represented by the first argument of sort $\underline{tr}$, by a sensor, represented by the second argument of sort $\underline{s}$, during the time interval, whose start point is represented by the third argument of sort $\underline{from}$, and whose end point is represented by the fourth argument of sort $\underline{to}$. The predicates, which appear in the premise literals of the rules, are basic feature predicates, which can be characterized as

$$bf(\underline{tr}, \underline{o}, \underline{s}, \underline{from}, \underline{to}),$$

where $bf$ denotes a predicate symbol, which describes the tendency of change of the measurements, which have been perceived during a trace, represented by the first argument of sort $\underline{tr}$, by a sensor, represented by the third argument of sort $\underline{s}$, which has a certain orientation, represented by the second argument of sort $\underline{o}$, during the time interval, whose start point is represented by the fourth argument of sort $\underline{from}$, and whose end point is represented by the fifth argument of sort $\underline{to}$. We use **SF**, to denote the finite set containing the sensor feature predicates (here: 16 predicates), i.e.,

$$\begin{aligned}\mathbf{SF} = \{ & \texttt{s\_jump\_parallel}(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), \texttt{s\_jump\_diagonal}(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}),\\ & \texttt{s\_convex\_straight\_to}(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), \ldots\}.\end{aligned}$$

**BF** denotes the finite set of 13 basic feature predicates, i.e.,

$$\begin{aligned}\mathbf{BF} = \{ & \texttt{stable}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), \texttt{decreasing}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}),\\ & \texttt{incr\_peak}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), \texttt{no\_movement}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}),\\ & \texttt{something\_happened}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), \ldots\}.\end{aligned}$$

The rules are *chain Datalog rules*, which are the topic of this paper. In the next Section, we will introduce the logic programming concepts, which we need to characterize their syntax and semantics.

# 3 Logic Programming Concepts

## 3.1 Definitions

We use the notation and definitions given in [11] and [3]. We assume the existence of four finite, pairwise disjoint sets, $SO$, $CS$, $PS$, $VS$, containing *sort*, *constant*, *predicate*, and *variable* symbols. Sort symbols start with small letters and are underlined (e.g., $\underline{tr}, \underline{o}, \underline{s}, \underline{from}, \underline{to}, \ldots$). Constant symbols start with small letters (e.g., $x, y, z, \ldots$), predicate symbols with small letters (e.g., $a, b, c, \ldots, p, q, r, \ldots$), and variable symbols with capital letters (e.g., $Tr, O, S, X, Y, Z, \ldots$)[2]. $V_{\underline{s}}$ denotes, that variable $V$ refers to sort $\underline{s}$. A signature is defined by the tuple $(SO, CS, PS, \alpha)$, where $\alpha$ is a function, which maps a predicate symbol to a sequence of $n$ sort symbols, which denote the sorts of the respective arguments. We restrict a *term* to be either a constant or a variable. An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate symbol and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom or its negation. A *clause* is a closed formula[3] of the form $\forall X_1 \ldots \forall X_v (L_1 \vee \ldots \vee L_s)$, where $X_1, \ldots, X_v$ are variables and $L_1, \ldots, L_s$ are literals. Let $B_1, \ldots, B_n$ and $A_1, \ldots, A_m$ be atoms. Then, the clause $\forall X_1 \ldots \forall X_r (B_1 \vee \ldots \vee B_n \vee \neg A_1 \vee \ldots \vee \neg A_m)$ is denoted by $B_1, \ldots, B_n \leftarrow A_1, \ldots A_m$. A *program clause* or *definite clause* is a clause with $n = 1$. A *unit clause* or *fact* is a clause with $n = 1$ and $m = 0$. A *program rule* or simply a *rule* is a clause with $n = 1$ and $m > 0$, i.e., $B \leftarrow A_1, \ldots, A_m$. $B$ is called the *head* of the rule, the conjunction $A_1, \ldots, A_m$ is called the *body* of the rule. A rule is *safe* (*generative*), if all variables, which occur in the head of the rule, also occur in the body of the rule. A *logic program* is a finite set of definite clauses. A *Datalog program* is a function-free logic program, such that each rule of the program is safe. The safety condition together with the requirement that each fact belonging to a Datalog program be a ground fact

[2] Subscripts and superscripts can be applied to the symbols used for constants, variables, and predicates.

[3] A formula is closed if every variable occurring in it is bound by a quantifier.

ensures, that only a finite number of facts can be deduced from a Datalog program (see [3]). Ullman et. al. (see [24]) distinguish between basic and extended logic programs. A *basic logic program*, which is denoted by $\mathbf{P}_{\mathrm{I}}$, is a finite set of rules containing two types of predicates:

- IDB (Intentional Database) predicates, which appear in rule heads and, possibly, in rule bodies; $p, q, \ldots$ denote IDB predicates.
- EDB (Extensional Database) predicates, which appear in rule bodies only; $a, b, c, d, \ldots$ denote EDB predicates.
- $r_1, r_2, \ldots$ denote predicates, which may either be IDB or EDB predicates.

IDB($\mathbf{P}_{\mathrm{I}}$) and EDB($\mathbf{P}_{\mathrm{I}}$) denote the intensional and extensional database predicates, respectively, of the basic logic program $\mathbf{P}_{\mathrm{I}}$. An EDB *fact* is a ground fact over an EDB predicate, i.e., a fact with constants as arguments. If $A_i, i = 1, 2, \ldots$ and $B$ denote atoms, then $\breve{A}_i, i = 1, 2, \ldots$ and $\breve{B}$ denote ground facts over the respective predicates. An EDB *instance*, denoted by $\mathbf{P}_{\mathrm{E}}$, is a finite set of EDB facts. An *extended logic program*, denoted by $\mathbf{P}$, is the union of a basic logic program and an EDB instance, i.e., $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$. In the following, we assume that the rules of a basic program are Datalog rules.

Furthermore, we assume in certain contexts that the programs are *linear*.

**Definition 1** *([24]) A program is* linear*, if it contains rules, each of which has at most one recursive subgoal and at most one IDB subgoal.*

We consider basic logic programs with rules of a special form. We use the definitions given in [24] for elementary chain rules, elementary chains, left and right blocks. An *elementary chain rule* is a rule containing only binary predicates of the form

$$p(X, Y) \leftarrow r_1(X, X_1), r_2(X_1, X_2), \ldots, r_{k+1}(X_k, Y), \tag{1}$$

where $k > 0$, $p$ and $r_i, i = 1, \ldots, k+1$ denote predicates, and $X, Y, X_j, j = 1, \ldots, k$ are variables. Let $C$ be an (elementary) chain rule and $A$ an atom occurring in $C_{body}$, e.g., $r(X, Y)$. Then, we say that $A$ starts from variable $X$ and leads to variable $Y$. Let $from(A)$ denote the function, which maps an arbitrary predicate to its starting variable, and $to(A)$ the function, which maps a predicate to its ending variable. Let $X_i, i = 1, \ldots, k$ be the variables occurring in $C_{body}$ and not in $C_{head}$. $X_i$ is called a *chaining variable*, if $C_{body}$ contains two atoms, $A_1$ and $A_2$, such that $to(A_1)$ is equal to $from(A_2)$, e.g., given $A_1 = r_{i-1}(X_{i-1}, X_i)$ and $A_2 = r_i(X_i, X_{i+1})$, $X_i$ is a chaining variable. In principle, clauses can be considered as sets of literals, whose order of appearance does not matter. In the special case of chain rules, the atoms in the body of the rule can be sorted according to the relation $\ll$, which we define with the help of chaining variables as follows: Let $A_1, A_2$ be two atoms occurring in $C_{body}$. Then, $A_1$ *precedes* $A_2$, $A_1 \ll A_2$, if $to(A_1)$ is equal to $from(A_2)$. Given chain rule r1, we have $r_1(X, X_1) \ll r_2(X_1, X_2) \ll \ldots \ll r_{k+1}(X_k, Y)$.

Although this relation has not been stated explicitly in [24], it leads to their definition of an *elementary chain*, which is an ordered list of binary atoms, e.g., the ordered sequence of premise atoms of Rule 1, i.e.,

$$r_1(X, X_1), r_2(X_1, X_2), \ldots, r_{k+1}(X_k, Y). \tag{2}$$

The variables $X$ and $Y$ are called the *left block* and *right block* of the chain.

Correspondingly, we can define a relation on the chaining variables. Let $X_i$ and $X_j$ be two variables of the set of variables occurring in the chain of Rule 1, i.e., $\{X, Y, X_1, \ldots, X_k\}$. We say that $X_i$ *leads to* $X_j$, $X_i \leadsto X_j$, if there exists an atom $A \in C_{body}$, such that $from(A) = X_i$ and $to(A) = X_j$. Given the elementary Chain 2, we have $X \leadsto X_1 \leadsto X_2 \leadsto \ldots \leadsto X_k \leadsto Y$. Note, that both relations, $\ll$ and $\leadsto$, are intransitive, irreflexive, and asymmetric for (elementary) chain rules. Thus, they are neither a weak nor a strict order. A further restriction is, that the chaining variables have to be unique in the sense, that in a chain, there do not exist two atoms with the same starting and ending variable. The chain $a(X, X_1), b(X_1, X_2), c(X_1, X_2), d(X_2, X_3)$, for example, does not satisfy this requirement.

## 3.2 Semantics of logic programs

Given a function-free extended logic program $\mathbf{P}$ , the *Herbrand universe* of $\mathbf{P}$ , $\mathcal{U}_H(\mathbf{P})$, is the set of all constants appearing in $\mathbf{P}$ [4]. The *Herbrand base* of a program $\mathbf{P}$ , $\mathcal{B}_H(\mathbf{P})$, is the set of all ground atoms, which can be formed from the predicates in $\mathbf{P}$ and the terms in $\mathcal{U}_H(\mathbf{P})$ and which obey the sort conditions. An *interpretation* is a subset of $\mathcal{U}_H(\mathbf{P})$.

Given a function-free logic program $\mathbf{P}$ , there is a mapping $\mathrm{T}_\mathbf{P}$ from interpretations to interpretations. Let $I$ be an interpretation. Then, $\mathrm{T}_\mathbf{P}$ is defined as follows:

$$\begin{aligned} \mathrm{T}_\mathbf{P}(I) = \{\breve{B} \in \mathcal{B}_H(\mathbf{P}) \quad | \quad & C\sigma = (\breve{B} \leftarrow \breve{A}_1, \ldots, \breve{A}_m), m \geq 0, \\ & \text{is a ground instance of a clause } C \in \mathbf{P} \text{ and } \breve{A}_1, \ldots, \breve{A}_m \in I\} \end{aligned} \tag{3}$$

Van Emden and Kowalski have shown in [25], that the least fixpoint of $\mathrm{T}_\mathbf{P}$ is the minimum Herbrand model of $\mathbf{P}$[5](see also [2]). In the context of computing the minimum model, we mean the IDB-portion of the minimum Herbrand model. $\mathrm{T}^i_\mathbf{P}(\emptyset)$ denotes the i-th application of the $\mathrm{T}_\mathbf{P}$-mapping, with $\mathrm{T}^0_\mathbf{P}(\emptyset) = \emptyset$ and $\mathrm{T}^{i+1}_\mathbf{P}(I) = \mathrm{T}_\mathbf{P}(\mathrm{T}^i_\mathbf{P}(I))$. The fixpoint of the $\mathrm{T}_\mathbf{P}$-mapping is denoted by $\bigcup_{i=0}^{\infty} \mathrm{T}^i_\mathbf{P}(\emptyset)$. In the function-free case, there exists a natural number $\omega$, such that $\mathrm{T}^\omega_\mathbf{P}(\emptyset) = \bigcup_{i=0}^{\infty} \mathrm{T}^i_\mathbf{P}(\emptyset)$, i.e., the fixpoint, and thus the minimum Herbrand model, is determined after $\omega$ applications of the $\mathrm{T}_\mathbf{P}$-mapping. As we deal with basic logic programs $\mathbf{P}_\mathrm{I}$, we use $\mathrm{T}^\omega_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})$ to denote the fixpoint of $\mathrm{T}_{\mathbf{P}_\mathrm{I}}$ (and, thus, the minimum Herbrand model) of the program $\mathbf{P} = \mathbf{P}_\mathrm{I} \cup \mathbf{P}_\mathrm{E}$ with $\mathrm{T}^0_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E}) = \mathbf{P}_\mathrm{E}$.

**Definition 2** *([24]) Two basic logic programs are* equivalent with respect to a set of IDB-predicates $\mathcal{I}$*, if the minimum models of both programs, extended with the same EDB, restricted to the predicates in* $\mathcal{I}$*, are the same.*

**Definition 3** *Given an extended logic program* $\mathbf{P} = \mathbf{P}_\mathrm{I} \cup \mathbf{P}_\mathrm{E}$ *and a set of target predicates* $\mathcal{I} \subseteq IDB(\mathbf{P}_\mathrm{I})$*, the* coverage *for* $\mathcal{I}$ *is the subset of the minimum Herbrand model*

$$Cov_\mathbf{P}(\mathcal{I}) = \{p_i(t_1, \ldots, t_s) \mid p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in T^\omega_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})\}.$$

[4]Note, that in the context of general logic programs, a term can be a complex structure built from function symbols, variables, and constants. In that case, the Herbrand universe does not coincide with the set of constants (see, e.g., [25]).

[5]The Herbrand universe has to contain at least one constant to guarantee the existence of a minimal model.

So, two basic logic programs, extended with the same EDB instance $\mathbf{P}_{\mathrm{E}}$, have the same coverage for $\mathcal{I}$, if they are equivalent with respect to $\mathcal{I}$.

Given a function-free extended logic program $\mathbf{P}$, a *derivation tree* for a ground fact/atom $\breve{B}^0$ is a tree with atoms as nodes and edges between parents and children, such that:

1. $\breve{B}^0$ is the root.
2. For every internal node $\breve{B}^l$, whose children are $\breve{A}_1^l, \ldots, \breve{A}_k^l$, there is some ground rule instance $C\sigma$ of $C \in \mathbf{P}$, such that $C\sigma$ is $\breve{B}^l \leftarrow \breve{A}_1^l, \ldots, \breve{A}_k^l$.
3. Every node is in the minimum model of $\mathbf{P}$ ; leaves are not necessarily in the EDB.

A *complete derivation tree* is one in which all leaves are EDB facts. A *path* in the derivation tree is a directed path away from the root. The *fringe* of the tree is the set of its leaves.

## 3.3 Correspondence between chain Datalog programs and CFG's

In order to illustrate the correspondence between chain Datalog rules and CFG's, we use the examples and the lemma given in [24]. Elementary chain rules can be represented by nodes, which represent their arguments, and by directed arcs between the nodes, labeled by predicate symbols. Given the elementary chain

$$q_0(U,V), p(V,W), q_1(W,X), q_0(X,Y), q2(Y,Z),$$

we get the graph

$$U \overset{q_0}{\leadsto} V \overset{p}{\leadsto} W \overset{q_1}{\leadsto} X \overset{q_0}{\leadsto} Y \overset{q2}{\leadsto} Z,$$

which reflects the relation $\leadsto$ between variables. The elementary chain rule

$$p(X,Y) \leftarrow q_1(X,X_1), p(X_1,X_2), q_2(X_2,Y)$$

can be represented as

$$X \overset{p}{\leadsto} Y \leftarrow X \overset{q_1}{\leadsto} X_1 \overset{p}{\leadsto} X_2 \overset{q_2}{\leadsto} Y.$$

By ignoring the variables, by treating IDB predicates as grammar non-terminals, EDB-predicates as grammar terminals, and by inverting the implication arrow, we can rewrite the above mentioned elementary chain rule as grammar production $p \rightarrow q_1 p q_2$.

**Context-free grammars** A *context-free grammar* (CFG) (see, e.g., [9]) is a 4-tuple $G = (V, \Sigma, P, s)$, where $V$ and $\Sigma$ are disjoint, finite sets of *variables* and *terminals*, respectively. The special variable $s \in V$ is called the *start symbol*. $P$ is a finite set of productions; each production is of the form $p \rightarrow \alpha$, where $p$ is a variable and $\alpha$ is a string from $(V \cup \Sigma)$. Given a production $p \rightarrow \alpha$, $p$ is called its *head* and $\alpha$ its *body*. Let $\Rightarrow_G$ be the relation defined on strings in $(V \cup \Sigma)^*$ as follows: Let $p$ be a variable and $\alpha, \beta, \gamma$ be strings in $(V \cup \Sigma)^*$. If $p \rightarrow \alpha$ is a production in $P$, then $\beta p \gamma \Rightarrow \beta \alpha \gamma$. Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow_G$. The set $\hat{L}(G) = \{w \in \Sigma^* | p_1 \Rightarrow^* w\}$ is called the *language generated* by $G$. A set $\hat{L}$ is a *context-free language* (CFL) if $\hat{L} = \hat{L}(G)$ for some context-free grammar $G$. We define grammars, $G_1$ and $G_2$, to be equivalent, if $\hat{L}(G_1) = \hat{L}(G_2)$. We restrict ourselves to $\epsilon$-free grammars[6] and languages ($\epsilon$ denotes the empty word).

Analogous to Ullman and van Gelder [24], we define for each basic program $\mathbf{P}_\mathrm{I}$ a context-free grammar.

**Definition 4** *Let $\mathbf{P}_\mathrm{I}$ be a basic chain Datalog program. The grammar, which corresponds to $\mathbf{P}_\mathrm{I}$ is $G_{\mathbf{P}_\mathrm{I}} = (V, \Sigma, P, s)$, where $V = IDB(\mathbf{P}_\mathrm{I}) \cup \{s\}$, where $s$ is the starting symbol not occurring in $IDB(\mathbf{P}_\mathrm{I})$ and $EDB(\mathbf{P}_\mathrm{I})$. $\Sigma$ is defined as $\Sigma = EDB(\mathbf{P}_\mathrm{I})$ and*

$$\begin{aligned} P = \{p \rightarrow r_1, r_2, \ldots, r_n \quad &| \quad \mathbf{P}_\mathrm{I} \textit{ contains a rule of the form} \\ & \quad\; p(X,Y) \leftarrow r_1(X,X_1), r_2(X_1,X_2), \ldots, r_n(X_{n-1},Y)\} \\ &\cup \quad \{s \rightarrow p \mid p \in IDB(\mathbf{P}_\mathrm{I})\}. \end{aligned}$$

Ullman and Van Gelder have proven in [24] the following Lemma, which allows us to characterize chain programs with the help of their associated context-free grammars:

**Lemma 1** *Let $\mathbf{P}_\mathrm{I}$ be an elementary chain program, and let $G$ be the associated CFG in which each production corresponds to an elementary chain rule of $\mathbf{P}_\mathrm{I}$ as described above (or is of the form $s \rightarrow p, p \in IDB(\mathbf{P}_\mathrm{I})$). Let predicate $p$ in $\mathbf{P}_\mathrm{I}$ correspond to nonterminal $p$ in $G$, and let $s \rightarrow p$ be a production of $G$. Let $\mathbf{P}_\mathrm{E}$ be an EDB instance for $\mathbf{P}_\mathrm{I}$ and let $\mathbf{P} = \mathbf{P}_\mathrm{I} \cup \mathbf{P}_\mathrm{E}$. Let $\check{F}$ be a ground elementary chain all of whose atoms are in $\mathbf{P}$, and whose left and right blocks are constants, say $x$ and $y$, respectively. Let $\hat{F}$ be the string of terminal symbols of $G$ that corresponds to $\check{F}$, i.e., the string of EDB predicate symbols that occur in $\check{F}$. Then,*

$$\begin{gathered} \check{F} \textit{ is the fringe of the complete derivation tree of } p \\ \Leftrightarrow \\ \hat{F} \textit{ is in the language generated by } G. \end{gathered}$$

In the following sections, we shall extensively make use of the correspondence between chain Datalog programs and context-free grammars, in order to characterize the programs in terms of properties of the associated context-free languages.

## 3.4 Non-elementary chain Datalog rules

We now consider the rules, which have been learned, in order to derive sensor features from basic features. Given a basic logic program consisting of these rules, the basic feature predicates in the set **BF** are EDB predicates, the sensor feature predicates in the set **SF** IDB predicates. An example rule from Section 1 is

$$\begin{aligned} \texttt{s\_jump\_parallel}(Tr, S, \mathbf{X}, \mathbf{Y}) \quad \leftarrow \quad & \texttt{stable}(Tr, O, S, \mathbf{X}, \mathbf{X}_1), \texttt{incr\_peak}(Tr, O, S, \mathbf{X}_1, \mathbf{X}_2), \\ & \texttt{stable}(Tr, O, S, \mathbf{X}_2, \mathbf{Y}). \end{aligned} \tag{4}$$

[6] An $\epsilon$-free grammar is a grammar with no productions of the form $p \rightarrow \epsilon$. An $\epsilon$-free grammar corresponds to the requirement, that a basic logic program does not contain facts/unit clauses.

Rules like this one are non-elementary chain rules[7]. The boldly printed variables are the variables of the corresponding elementary chain rule. Each basic feature predicate starts from the variable at its fourth argument position and leads to the variable at its fifth position. In this domain, these variables denote the starting and end point of the time interval during which the basic feature is perceived. Thus, we have, e.g., $from(stable(Tr,O,S,X,X_1)) = X$ and $to(stable(Tr,O,S,X,X_1)) = X_1$. The sequence of premise atoms

$$\mathtt{stable}(Tr,O,S,X,X_1), \mathtt{incr\_peak}(Tr,O,S,X_1,X_2), \mathtt{stable}(Tr,O,S,X_2,Y). \quad (5)$$

is a non-elementary chain, where the $X_i, i = 1,2$ are chaining variables, and $X$ and $Y$ are the left and right block, respectively. Here, the relation $\ll$ coincides with the chronological order, in which the basic features are observed. The other variables guarantee, that the sequence of basic feature atoms refers to the same trace, $Tr$, and to the same sensor, $S$, which does not change its orientation, $O$, during the time interval from $X$ to $Y$.

In the following, whenever we talk about chain Datalog programs for deriving sensor features from basic features, we assume to be given the signature $(SO, CS, PS, \alpha)$, where $SO = \{\underline{tr}, \underline{o}, \underline{s}, \underline{from}, \underline{to}, \ldots\}$, and $PS = \{a,b,c,\ldots,q,\ldots,p,r,\ldots\}$. We divide $PS$ into two disjoint sets $PS_{\mathbf{BF}} = \{a,b,c,\ldots,q\}$ and $PS_{\mathbf{SF}} = \{p_1,p_2,p_3,\ldots,p_n\}$, i.e., $a,b,c,\ldots$ denote some of the predicate symbols of the predicates in **BF** and the $p_i$ denote some predicate symbols occurring in **SF**. Then $\alpha$ is defined as follows

$$\begin{array}{lll} \forall a \in PS_{\mathbf{BF}} & \alpha(a) = & \underline{tr}\ \underline{o}\ \underline{s}\ \underline{from}\ \underline{to} \\ \forall p_i \in PS_{\mathbf{SF}} & \alpha(p_i) = & \underline{tr}\ \underline{s}\ \underline{from}\ \underline{to}. \end{array}$$

If we introduce the sort $\underline{bool}$, we can rewrite these statements as

$$\begin{array}{ll} \forall a \in PS_{\mathbf{BF}} & a : \underline{tr}, \underline{o}, \underline{s}, \underline{from}, \underline{to} \to \underline{bool} \\ \forall p_i \in PS_{\mathbf{SF}} & p_i : \underline{tr}, \underline{s}, \underline{from}, \underline{to} \to \underline{bool}. \end{array}$$

Note, that for a non-elementary chain rule, e.g., Rule 4, we get the corresponding elementary chain rule, by omitting the variables $Tr$, $S$, and $O$

$$\mathtt{s\_jump\_parallel}(\mathbf{X},\mathbf{Y}) \leftarrow \mathtt{stable}(\mathbf{X},\mathbf{X_1}), \mathtt{incr\_peak}(\mathbf{X_1},\mathbf{X_2}), \mathtt{stable}(\mathbf{X_2},\mathbf{Y}).$$

Vice versa, we can extend an elementary chain rule by introducing the variables $Tr_{\underline{tr}}, O_{\underline{o}}$, and $S_{\underline{s}}$ at the appropriate positions (according to $\alpha$ of the signature) of the sensor and basic feature predicates in $PS_{\mathbf{BF}}$ and $PS_{\mathbf{SF}}$. Given that, we can use CFGs to characterize also these non-elementary chain rules.

## 3.5 Constraints

Given our domain of application, EDB predicates are basic feature predicates and IDB predicates are sensor feature predicates. From an "object-oriented" point of view, a sensor features represent a class of objects with 5 properties: its type (predicate name), the trace,

[7] Note, that the rules are also safe, as every variable in the head of the rule also appears in the body of the rule.

$\mathtt{tr}_{\mathbf{SF}}$, during which it has been perceived, the sensor, $\mathtt{s}_{\mathbf{SF}}$, which has perceived the object, the start point of a time interval, $\mathtt{from}_{\mathbf{SF}}$, and its end point, $\mathtt{to}_{\mathbf{SF}}$. Correspondingly, a basic feature has as properties its type, a trace, $\mathtt{tr}_{\mathbf{BF}}$, a sensor, $\mathtt{s}_{\mathbf{BF}}$, and its orientation, $\mathtt{o}_{\mathbf{BF}}$, a start point, $\mathtt{from}_{\mathbf{BF}}$, and an end point, $\mathtt{to}_{\mathbf{BF}}$, of the time interval during which it was perceived. In the following, we define the functions, which determine for a given predicate/object its respective property. The functions $tr_{\mathbf{SF}}$, $s_{\mathbf{SF}}$, $from_{\mathbf{SF}}$, and $to_{\mathbf{SF}}$, on one hand, and $tr_{\mathbf{BF}}$, $o_{\mathbf{BF}}$, $s_{\mathbf{BF}}$, $from_{\mathbf{BF}}$, and $to_{\mathbf{BF}}$, on the other hand, map an atom over a sensor (basic feature) predicate to the arguments, representing their property values. For example, $tr(a(t1,90,s5,1,8)) = t1$ and $to(p_1(t1,s5,1,15)) = 15$.

We define a *constraint* to be an equation of the form $P_{OC} = V$, where $P_{OC}$ denotes a property of an instance of (predicate) class $OC$, and $V$ denotes its value(s). A set of constraints is denoted by $\kappa_{OC} = \{P_{OC,1} = V_1, \ldots, P_{OC,n} = V_n\}$. If we apply a set of constraints to a predicate $A$ of a specific class, the result, denoted $A\kappa_{OC}$, is an atom, ground or non-ground, over the respective predicate, whose arguments representing the properties are set to the respective property values. For example, if we apply the constraints $\kappa_{\mathbf{BF}} = \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5, \mathtt{from}_{\mathbf{BF}} = 8\}$ to the basic feature predicate $A = b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}})$, the result $A\kappa_{\mathbf{BF}}$ is $b(t1, 90, s5, 8, \_{\underline{to}})$, where $\_{\underline{to}}$ denotes an arbitrary variable of sort $\underline{to}$. If we apply the constraints $\kappa_{\mathbf{SF}} = \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8, \mathtt{to}_{\mathbf{SF}} = 15\}$ to the sensor feature predicate $B = p1(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}})$, we get $B\kappa_{\mathbf{SF}} = p_1(t1, s5, 1, 1, 15)$. We use constraints in the marker passing method presented in Section 5.

# 4 Structuring Chain Datalog Rules in Prefix Acceptors

There are several characteristics of basic logic programs consisting of non-elementary chain Datalog rules, e.g., those, which derive sensor features from basic features. Consider the non-recursive example program $\mathbf{P}_{\mathrm{I}}$

$$\begin{aligned}
p_1(Tr,S,X,Y) &\leftarrow a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). && (6)\\
p_2(Tr,S,X,Y) &\leftarrow a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). && (7)\\
p_3(Tr,S,X,Y) &\leftarrow a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3), && (8)\\
&\qquad d(Tr,O,S,X_3,Y).\\
p_4(Tr,S,X,Y) &\leftarrow b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,Y). && (9)\\
p_5(Tr,S,X,Y) &\leftarrow b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,X_3), && (10)\\
&\qquad a(Tr,O,S,X_3,X_4), b(Tr,O,S,X_4,Y).
\end{aligned}$$

where $p_1, p_2, p_3, p_4, p_5$ denote sensor feature predicate symbols in $PS_{\mathbf{SF}}$ and $a, b, c, d$ denote basic features predicate symbols in $PS_{\mathbf{BF}}$. We have EDB($\mathbf{P}_{\mathrm{I}}$)= $\{a,b,c,d\}$ and IDB($\mathbf{P}_{\mathrm{I}}$)= $\{p_1,p_2,p_3,p_4,p_5\}$.

The first characteristic is, that the IDB predicates, i.e., the sensor feature predicates occur only in rule heads. Thus, the program has inference depth 1. In the example program, the premise literals are sorted according to the relation $\ll$. In our domain, this reflects the chronological order of the perceived observations. Given that, there exist a lot of rules, whose premise chains are prefixes of premise chains of other rules.

**Definition 5** *A chain $Ch_1$ is a* prefix (chain) *of chain $Ch_2$, if there exists a substitution $\sigma$ and a chain $Ch_3$, such that $Ch_2 = Ch_1 \sigma Ch_3$.*

The chain $Ch_1 = a(U, U_1), b(U_1, U_2)$, for example, is a prefix of chain $Ch_3 = a(X, X_1), b(X_1, X_2), c(X_2, Y)$ with $\sigma = \{U/X, U_1/X_1, U_2/X_2\}$ and $Ch_3 = c(X_2, Y)$. Furthermore, there exist ambiguous rules, i.e., rules with the same premise but different conclusions. Program $\mathbf{P}_\mathrm{I}$ is used to derive via forward inferences higher-level concepts from a sequence of observations.

Both characteristics, prefix chains and ambiguous rules, cause during evaluations via forward inferences, that the same input fact may have to be matched redundantly against premise literals of several rules. Assume, for example, that the robot perceives the ground chain of basic feature predicates, i.e., that the basic logic program $\mathbf{P}_\mathrm{I}$ gets as "input" the EDB instance (ground chain), which is an example of a sequence of basic features, which the robot perceives, while it is moving around

$$\mathbf{P}_\mathrm{E} = \{a(t1, 90, s5, 1, 8), b(t1, 90, s5, 8, 10), c(t1, 90, s5, 10, 15), d(t1, 90, s5, 15, 17)\}.$$

Then, the first EDB fact, $a(t1, 90, s5, 1, 8)$, matches the first premise atom of rules r6, r7, and r8. Although it cannot possibly lead to a successful derivation, the fact can, in principle, also be matched to the fourth literal of rule r10. For the second EDB fact, $b(t1, 90, s5, 8, 10)$, there exists a matching premise atom for every rule of the program $\mathbf{P}_\mathrm{I}$. In this case, it also makes no sense to match the fact with the fifth premise literal of rule r10. The minimum Herbrand model of the extended logic program $\mathbf{P} = \mathbf{P}_\mathrm{I} \cup \mathbf{P}_\mathrm{E}$ is equal to the fixpoint of the $\mathrm{T}_{\mathbf{P}_\mathrm{I}}$-mapping

$$\mathrm{T}^{\omega}_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E}) = \{p_1(t1, s5, 1, 15), p_2(t1, s5, 1, 15), p3(t1, s5, 1, 17), p_4(t1, s5, 8, 17)\}.$$

Our goal is, to structure the rules in such a way, that the multiple and superfluous matches, mentioned above, are avoided during the calculation of the minimum Herbrand model.

In this section, we present methods, which map a chain Datalog program to a prefix acceptor. We apply a marker passing method (see Section 5) to this acceptor, in order to calculate via forward inferences the minimum Herbrand model of the original program $\mathbf{P}_\mathrm{I}$. The `prefix_tree` method generates from a set of chain Datalog rules a prefix tree, which is then mapped to a prefix acceptor. We can achieve the same result, by restructuring the original program $\mathbf{P}_\mathrm{I}$, such that the resulting program $\mathbf{P}'_\mathrm{I}$ can be mapped directly to a prefix acceptor. Both methods take as input a chain Datalog program, which satisfies the following requirements: The program is non-recursive, the IDB predicates occur only in rule heads, and the premise atoms of each rule are sorted according to the relation $\ll$. Each non-recursive program, in which IDB predicates occur also in rule bodies, can be transformed to one with no IDB predicates in rule bodies by unfolding each rule with IDB subgoals in all possible ways. In general, clauses can be considered as sets of literals, whose order does not matter. In the next subsections, we present methods, which sort the premise literals of a chain Datalog rule according the relation $\ll$. In the sequel, we present the prefix tree and restructuring methods.

## 4.1 Sorting the premise literals of chain Datalog rules

We present two methods for sorting the premise literals of a chain Datalog rule, which exploit its syntactical features. The first one, `sort`, assumes, that for the given rule two requirements are satisfied

1. The relations $\leadsto$ and $\ll$ have to be intransitive, irreflexive, and asymmetric.

2. There are no premise literals, which have the same starting and ending variable.

The second method, `sort_dc`, does not require assumption 1 at the price of background knowledge about the data classes, to which the predicates of the rule belong. To be more specific, the user has to specify the functions $from$ and $to$ for each data class. In our domain of application, the starting variable is the one, which denotes the start point of the time interval, during which a sensor (basic) feature is perceived. The ending variable is the one, which represents the end point of the time interval. In this case, we can use the functions $from(A)$ and $to(A)$, which first determine the data class of literal $A$ and then call the function for the respective property of the data class, e.g., $from_{\mathbf{BF}}(A)$ ($from_{\mathbf{SF}}(A)$) and $to_{\mathbf{BF}}(A)$ ($to_{\mathbf{SF}}(A)$).

### 4.1.1 The `sort`-method

If assumptions 1 and 2 are satisfied, each chaining variable $X_i$ occurs in exactly one literal as starting variable and in one other literal as ending variable. Let $vars(L_i)$ and $vars(\{L_1, \ldots, L_n\})$ denote the variables occurring in literal $L_i$ and in the set of literals $\{L_1, \ldots, L_n\}$, respectively. Given a rule $C = C_{head} \leftarrow L_1, \ldots, L_n$, we determine for each pair of literals, $L_i, L_j, i \neq j, i, j \in \{1, \ldots, n\}$ the shared variables $vars(L_i) \cap vars(L_j)$. For the example rule

$$\begin{aligned} p_3(Tr, S, X, Y) \;\; \leftarrow \;\; & d(Tr, O, S, X_3, Y), a(Tr, O, S, X, X_1), c(Tr, O, S, X_2, X_3), \\ & b(Tr, O, S, X_1, X_2) \end{aligned}$$

we get

$$\begin{array}{rll} (d(Tr, O, S, X_3, Y), & a(Tr, O, S, X, X_1)): & \{Tr, O, S\} \\ (d(Tr, O, S, X_3, Y), & c(Tr, O, S, X_2, X_3)): & \{Tr, O, S, X_3\} \\ (d(Tr, O, S, X_3, Y), & b(Tr, O, S, X_1, X_2)): & \{Tr, O, S\} \\ (a(Tr, O, S, X, X_1), & c(Tr, O, S, X_2, X_3)): & \{Tr, O, S\} \\ (a(Tr, O, S, X, X_1), & b(Tr, O, S, X_1, X_2)): & \{Tr, O, S, X_1\} \\ (c(Tr, O, S, X_2, X_3), & b(Tr, O, S, X_1, X_2)): & \{Tr, O, S, X_2\} \end{array}$$

If we remove from each set the head variables, $vars(C_{head})$, we get

$$\begin{array}{rll} (d(Tr, O, S, X_3, Y), & a(Tr, O, S, X, X_1)): & \{O\} \\ (d(Tr, O, S, X_3, Y), & c(Tr, O, S, X_2, X_3)): & \{O, X_3\} \\ (d(Tr, O, S, X_3, Y), & b(Tr, O, S, X_1, X_2)): & \{O\} \\ (a(Tr, O, S, X, X_1), & c(Tr, O, S, X_2, X_3)): & \{O\} \\ (a(Tr, O, S, X, X_1), & b(Tr, O, S, X_1, X_2)): & \{O, X_1\} \\ (c(Tr, O, S, X_2, X_3), & b(Tr, O, S, X_1, X_2)): & \{O, X_2\}. \end{array}$$

Furthermore, we remove from each set the variables occurring in any other set:

$$\begin{array}{rll}
(d(Tr,O,S,X_3,Y), & a(Tr,O,S,X,X_1)): & \emptyset \\
(d(Tr,O,S,X_3,Y), & c(Tr,O,S,X_2,X_3)): & \{X_3\} \\
(d(Tr,O,S,X_3,Y), & b(Tr,O,S,X_1,X_2)): & \emptyset \\
(a(Tr,O,S,X,X_1), & c(Tr,O,S,X_2,X_3)): & \emptyset \\
(a(Tr,O,S,X,X_1), & b(Tr,O,S,X_1,X_2)): & \{X_1\} \\
(c(Tr,O,S,X_2,X_3), & b(Tr,O,S,X_1,X_2)): & \{X_2\}.
\end{array}$$

We consider only those pairs, which are associated with non empty variable sets. Given these partial chains of length 2, we try to extend them by merging, until we are left with a chain of length $n$. For our example, we get in the first iteration the extended chains

$$a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3)$$

and

$$b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3), d(Tr,O,S,X_3,Y).$$

In the second iteration, we get

$$a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3), d(Tr,O,S,X_3,Y).$$

We still have to check the left and right block. The first premise $L_{s_1}$ of the sorted chain $L_{s_1},\ldots,L_{s_n}$ has to share at least one variable (left block) with $C_{head}$, which does not occur in any other literal of the chain. Analogously, the last premise $L_{s_n}$ has to share at least one variable (right block) with $C_{head}$, which does not occur in any other literal of the sorted chain. This test succeeds for our example and the sorted chain rule $C_{head} \leftarrow L_{s_1},\ldots,L_{s_n}$ is returned. The pseudo-code of the method is given as Algorithm 1 below.

As we do not know, whether $X$ ($Y$) is the left or right block, two sorted premise chains are possible. If $X$ is the left block, we have

$$\begin{array}{rcl}
p_3(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3), \\
& & d(Tr,O,S,X_3,Y)
\end{array}$$

with

$$X \overset{p_3}{\rightsquigarrow} Y \leftarrow X \overset{a}{\rightsquigarrow} X_1 \overset{b}{\rightsquigarrow} X_2 \overset{c}{\rightsquigarrow} X_3 \overset{d}{\rightsquigarrow} Y.$$

If $Y$ is the left block, we have

$$\begin{array}{rcl}
p_3(Tr,S,X,Y) & \leftarrow & d(Tr,O,S,X_3,Y), c(Tr,O,S,X_2,X_3), b(Tr,O,S,X_1,X_2), \\
& & a(Tr,O,S,X,X_1)
\end{array}$$

with

$$Y \overset{p_3}{\rightsquigarrow} X \leftarrow Y \overset{d}{\rightsquigarrow} X_3 \overset{c}{\rightsquigarrow} X_2 \overset{b}{\rightsquigarrow} X_1 \overset{a}{\rightsquigarrow} X.$$

If assumptions 1 and 2 are not satisfied, the method will not find a sorted premise. Take for example the rule

$$p(X,Y) \leftarrow a(X,X_1), b(X_1,X_2), c(X_2,X_1), d(X_1,X_1), e(X_1,Y)$$

---

`sort`$(C_{head} \leftarrow C_{body})$
**begin**

1. $Pairs := \{(L_i, L_j) | L_i, L_j \in C_{body}, i \neq j\}$;
2. **for** each pair $(L_i, L_j) \in Pairs$
   **begin**
   (a) $L_iL_j\_Vars := vars(L_i) \cap vars(L_j)$;
   (b) $L_iL_j\_Vars := L_iL_j\_Vars - vars(C_{head})$;
   (c) $L_iL_j\_Vars := L_iL_j\_Vars - \bigcup_{k,l \neq i,j} vars(L_k, L_l)$;

   **end**
3. $Pairs := Pairs - \{(L_i, L_j) | L_iL_j\_Vars = \emptyset\}$;
4. **if** for each $(L_i, L_j)$, $L_iL_j\_Vars$ contains at least one variable, which does not occur in any $L_lL_k\_Vars$, with $i, j \neq k, l$,
   **then**
   (a) $SortedChain :=$ `extend_chains`$(Pairs, |C_{body}|)$;
   % let $SortedChain = L_1 \ldots L_n$;
   (b) $Left := (vars(L_1) \cap vars(C_{head})) - vars(\{L_2, \ldots, L_n\})$;
   (c) $Right := (vars(L_n) \cap vars(C_{head})) - vars(\{L_1, \ldots, L_{n-1}\})$;
   (d) **if** $Left \neq Right$, **then return** $SortedChain$;

   **else return failure**;

**end**

Algorithm 1: `sort`

---

with

$$X \overset{p}{\leadsto} Y \leftarrow X \overset{a}{\leadsto} X_1 \overset{b}{\leadsto} X_2 \overset{c}{\leadsto} X_1 \overset{d}{\leadsto} X_1 \overset{e}{\leadsto} Y$$

and

$$R_{\leadsto} = \{(X, X_1), (X_1, X_2), (X_2, X_1), (X_1, X_1), (X_1, Y)\},$$

which is not asymmetric and not irreflexive. If we apply the method for sorting the literals, we will be left without any possible pairings after step 3 of Algorithm 1.

### 4.1.2 The `sort_dc`-method

If we know for each predicate, which argument/property represents its starting and which one its ending variable, a much more efficient algorithm can be used. Given a rule $C_{head} \leftarrow L_1, \ldots, L_n$, we determine the starting and ending variable of $C_{head}$, $from(C_{head})$ and $to(C_{head})$, which are the $LeftBlock$ and $RightBlock$. Given the $LeftBlock$, we search for a literal $L \in C_{body}$, whose starting variable equals the $LeftBlock$. This literal becomes the first member of the sorted premise. We update $LeftBlock$ with $to(L)$ and repeat the search for the next literal until we have a chain of length $|C_{body}|$. The pseudo-code for the method is given as Algorithm 2 below.

If we use Algorithm 2, assumption 1 does not have to be satisfied. Assume, that for the binary predicates $p, a, b, c, d, e$, the starting variable is represented by the first argument

---

`sort_dc`$(C_{head} \leftarrow C_{body})$
**begin**

1. $LeftBlock := from(C_{head})$;
2. $RightBlock := to(C_{head})$;
3. $Premise := C_{body}$;
4. i:=1;
5. **while** $Premise \neq \emptyset$
   **begin**
   (a) select $L \in Premise$, such that $from(L) = LeftBlock$;
   (b) $LeftBlock := to(L)$;
   (c) $Premise := Premise - \{L\}$;
   (d) $L_i = L$;
   (e) $i := i + 1$;

   **end**
6. **if** $to(L_{i-1}) = RightBlock$
   **then** **return** $C_{head} \leftarrow L_1, \ldots, L_{i-1}$;
   **else** backtrack through step 5a;

**end**

Algorithm 2: `sort_dc`

---

and the ending variable by its second argument. Then, given the rule

$$p(X,Y) \leftarrow e(X_1,Y), c(X_2,X_1), a(X,X_1), d(X_1,X_1), b(X_1,X_2)$$

the method `sort_dc` will find the ordering

$$p(X,Y) \leftarrow a(X,X_1), b(X_1,X_2), c(X_2,X_1), d(X_1,X_1), e(X_1,Y),$$

which the method `sort` is not able to find. It is even possible, that the starting and ending variable of a predicate is represented by the same argument. In our application domain, it means, that the premise literals of rules including events happening at a time point instead of during a time interval can be sorted. An example of such a rule (see [22]) is

$$\begin{aligned} standing(Tr,X,Y,PerPDir,PSide,LPerc) \quad \leftarrow \quad & tp\_perception(Tr,X,Perc,PDir,PSide), \\ & stand(Tr,X,Y). \end{aligned}$$

where the *tp_perception*-predicate represents an observation at time point $X$, for which we define $from$ and $to$, such that both return the second argument.

If assumption 2 is satisfied, `sort_dc` will find one solution[8]. The method does not work if assumption 2 is not satisfied. Take for example the rule

$$p(X,Y) \leftarrow d(X_2,Y), a(X,X_1), b(X_1,X_2), c(X_2,Y).$$

[8] In order to make the algorithm `sort` output only one solution, we have integrated the heuristic that the starting variable has to occur before the ending variable in $C_{head}$.

We have $b(X_1, X_2) \not\ll c(X_1, X_2)$ and $c(X_1, X_2) \not\ll b(X_1, X_2)$. A unique ordering is not possible. Algorithm 2 finds the chain $a(X, X_1), b(X_1, X_2), d(X_2, Y)$ without being able to include $c(X_1, X_2)$. In our application domain, rules of this type represent events/observations which happen in parallel, i.e., during the same time interval. Therefore, the sorting method cannot deal, for example, with rules for sensor group features, such as the one for `sg_jump` given in Section 2.

### 4.1.3 Related work

The methods, presented above, sort the premise literals according to the relation $\ll$, which is defined via the relation $\rightsquigarrow$ on the chaining variables (see Section 3). This precedence relation excludes equality (i.e., in terms of the application, parallel events), if the $\rightsquigarrow$-relation satisfies assumptions 1 and 2. Motivated by the application, the goal of sorting is to make the sequence of premise literals reflect the chronological order of the events, in order to support efficient evaluation methods (see Section 5).

Ordered clauses are used in logic programming as well as in inductive logic programming. In logic programming, ordered clauses (no matter how the ordering itself has been achieved) are used to support efficient inference procedures, e.g., linear resolution (see [4]). In inductive logic programming ordered clauses are used to define certain characteristics in order to restrict the hypothesis language or to guide the search for hypotheses. Assuming ordered clauses to be given (no matter how the ordering has been achieved), Muggleton and Feng [16] define the depth and degree of their premise literals. By specifying maximal values on both, depth and degree, the hypothesis language is restricted. Morik et.al. [12] sort the premise literals of a rule in order to prune the search in the hypothesis space. They define the relation $\leq_P$ between premise literals via the minimum distance of the variables occurring in the literals. But, given the rule $C_{body} \leftarrow L_1, L_2, L_3, L_4$

$$\begin{aligned} p_3(Tr, S, X, Y) \quad \leftarrow \quad & a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, X_3), \\ & d(Tr, O, S, X_3, Y). \end{aligned}$$

the minimal distance of a variable occurring in $L_i, i = 1, 2, 3, 4$ is the same, namely 1, for each literal. This is due to the fact, that each literal shares a variable with the rule head, i.e., $vars(C_{head}) \cap vars(L_i) \neq 0$. So, for the purpose of hypothesis testing, each permutation of $L_1, L_2, L_3, L_4$ would do equally well. Obviously, we do not get deterministically the result, which we need for our purpose.

## 4.2 The `prefix` tree-method

In this section, we present the `prefix_tree` method, which maps a chain Datalog program, which satisfies the following conditions

C1: the rules are not recursive,

C2: IDB predicates occur only in rule heads, and

C3: the premise literals of each rule are sorted according to the relation $\ll$, to a *prefix acceptor*, which is a deterministic finite state automaton, whose input and output alphabet consists of predicates, not of propositional constants. This method has already been presented in [19] and [20]. It takes as input a set of *cases*, which associate a *target predicate*, i.e., an IDB predicate, with a sequence of sorted *defining predicates*, i.e., a premise chain of EDB predicates. The cases can be ground or non ground. In the latter case, they represent the set of chain Datalog rules, which are to be mapped to the prefix acceptor. In [19] and [20], we used ground cases as a training set, such that each *case* associated an example with its relevant background knowledge. So, the `prefix_tree` method can be used to infer the prefix acceptor directly from the training data without generating the rules explicitly, or it can be applied to the chain Datalog rules, which may have been learned by some other learning algorithm (see Figure 2). The cases

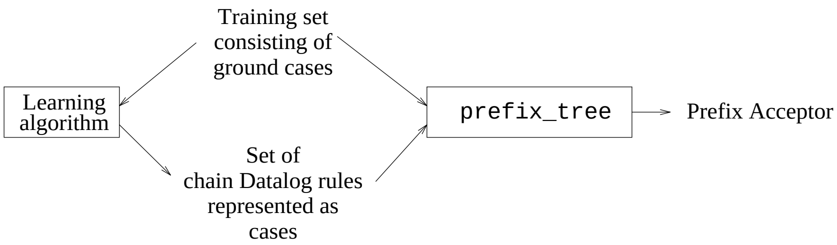


Figure 2: The `prefix_tree` method

are organized in a tree, such that for each case $[C_{head}, L_1, \ldots, L_n]$, there exists one path beginning at the root node, such that the labels of the edges on the path are unifiable with the respective literal $L_i, i \in \{1, \ldots, n\}$. As in [19], [20], the emphasis was on inferring the probabilistic automata, the algorithm (see Appendix A.1) contains some details, which are not so relevant for the application to rules. Here, we present the method from the logic programming point of view. In order to illustrate the basic ideas, we make extensive use of the regular grammars, which correspond to the chain Datalog programs. Furthermore, our presentation takes into account, that prefix acceptors for the propositional case have already been introduced by Angluin in [1].

We illustrate the method with our example program $\mathbf{P}_\mathrm{I}$

$$\begin{array}{lcl}
p_1(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y).\\
p_2(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y).\\
p_3(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3),\\
 & & d(Tr,O,S,X_3,Y).\\
p_4(Tr,S,X,Y) & \leftarrow & b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,Y).\\
p_5(Tr,S,X,Y) & \leftarrow & b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,X_3),\\
 & & a(Tr,O,S,X_3,X_4), b(Tr,O,S,X_4,Y).
\end{array}$$

According to Definition 4 (see Section 3), the CFG corresponding to $\mathbf{P}_\mathrm{I}$ is $G = (V, \Sigma, P, s)$ with $V = \{s, p_1, p_2, p_3, p_4, p_5\}$ and $\Sigma = \{a, b, c, d\}$. $P$ is the set containing the productions

$$s \quad \rightarrow \quad p_1|p_2|p_3|p_4|p_5$$

$$\begin{array}{rcl} p_1 & \rightarrow & abc \\ p_2 & \rightarrow & abc \\ p_3 & \rightarrow & abcd \\ p_4 & \rightarrow & bcd \\ p_5 & \rightarrow & bcdab. \end{array}$$

The language, generated by $G$, is $\hat{L}(G) = \{abc, abcd, bcd, bcdab\}$, which is a regular one. It is accepted by the DFA illustrated in Figure 3. Note, that we can rewrite $G$ according to

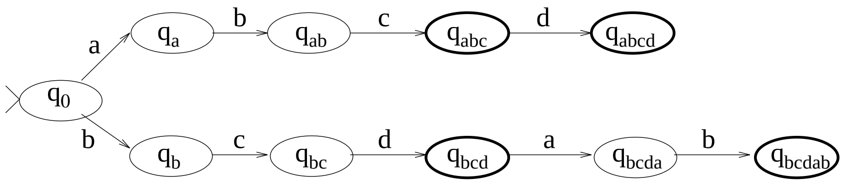


Figure 3: DFA, which accepts the language $\hat{L}(G) = \hat{L}(G')$

the transitions of the DFA, such that we get the equivalent, left-linear, regular grammar $G' = (V', \Sigma', P', s)$ with $V' = \{s, p_1, p_2, p_3, p_4, p_5, q_a, q_{ab}, q_{abc}, q_{abcd}, q_b, q_{bc}, q_{bcd}, q_{bcda}, q_{bcdab}\}$ and $\Sigma' = \{a, b, c, d\}$. $P'$ is the set of productions

$$\begin{array}{rclcrcl} s & \rightarrow & p_1|p_2|p_3|p_4|p_5 & & & & \\ p_1 & \rightarrow & q_{abc} & & q_{bcda} & \rightarrow & q_{bcd}\ a \\ p_2 & \rightarrow & q_{abc} & & q_{abc} & \rightarrow & q_{ab}\ c \\ p_3 & \rightarrow & q_{abcd} & & q_{bcd} & \rightarrow & q_{bc}\ d \\ p_4 & \rightarrow & q_{bcd} & & q_{ab} & \rightarrow & q_a\ b \\ p_5 & \rightarrow & q_{bcdab} & & q_{bc} & \rightarrow & q_b\ c \\ q_{bcdab} & \rightarrow & q_{bcda}\ b & & q_a & \rightarrow & a \\ q_{abcd} & \rightarrow & q_{abc}\ d & & q_b & \rightarrow & b \end{array}$$

We use the strings in $\hat{L}(G)$ to generate the prefix acceptor. A string $u$ is a *prefix* of a string $v$, if and only if there exists a string $w$, such that $uw = v$. Let $\hat{L}$ be a set of strings. Then the set of prefixes of the elements in $\hat{L}$ is defined as

$$Prefix(\hat{L}) = \{u : \ u \text{ is either the empty string } \epsilon \text{ or a non-empty string and there exists a string } v \text{, such that } uv \in \hat{L}\}.$$

Now, we structure the rules of the original basic chain Datalog program with inference depth 1 in a *prefix (tree) acceptor*, which is a deterministic finite state automaton, defined by the tuple $(Q, \Sigma, Z, \Delta, q_0, F, \lambda)$. $Q$ denotes a finite set of states, $\Sigma = \mathbf{BF}$ is the set of input predicates, $Z = \mathbf{SF}$ is the set of output predicates, $\Delta$ is the set of transitions, $q_0$ is the starting state, $F$ is the set of final states, and $\lambda$ is the output function.

Let $\hat{L}$ be the language generated by the grammar, associated with the program. Then, the prefix tree acceptor is constructed as follows: For each string $u \in Prefix(\hat{L})$ a state

$q_u \in Q$ is established. The initial state becomes the state, which is associated with the empty string $\epsilon$, i.e., $q_0 = q_\epsilon$. The final states are those, which have been established for the strings in $\hat{L}$. For a string $y \in \hat{L}$, there are rules $C_1, \ldots, C_n, n \geq 1$, whose premise chains correspond to $y$. The final state $q_y$ is associated with the set of sensor feature predicates in **SF**, which correspond to $C_{1,head}, \ldots, C_{n,head}$. The output function $\lambda$ maps each state to a subset of $Z$. Of course, $\lambda$ maps each non-final state to the empty set. Let $u$ be a string in $Prefix(\hat{L})$ and $a$ be a terminal symbol. Whenever there are two states $q_u$ and $q_{ua}$, which have been established for the strings $u$ and $ua$, then we establish a transition from the state $q_u$ to state $q_{ua}$, which is labeled by the EDB predicate $a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}) \in \mathbf{BF}$, i.e., $(q_u, a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{ua})$. Given the original program, $\mathbf{P}_\mathrm{I}$, the prefix acceptor $PA$, which is constructed, is illustrated in Figure 4. Note the correspondence between $PA$ and the DFA in Figure 3. It accepts as input a ground chain of basic feature predicates, e.g.,

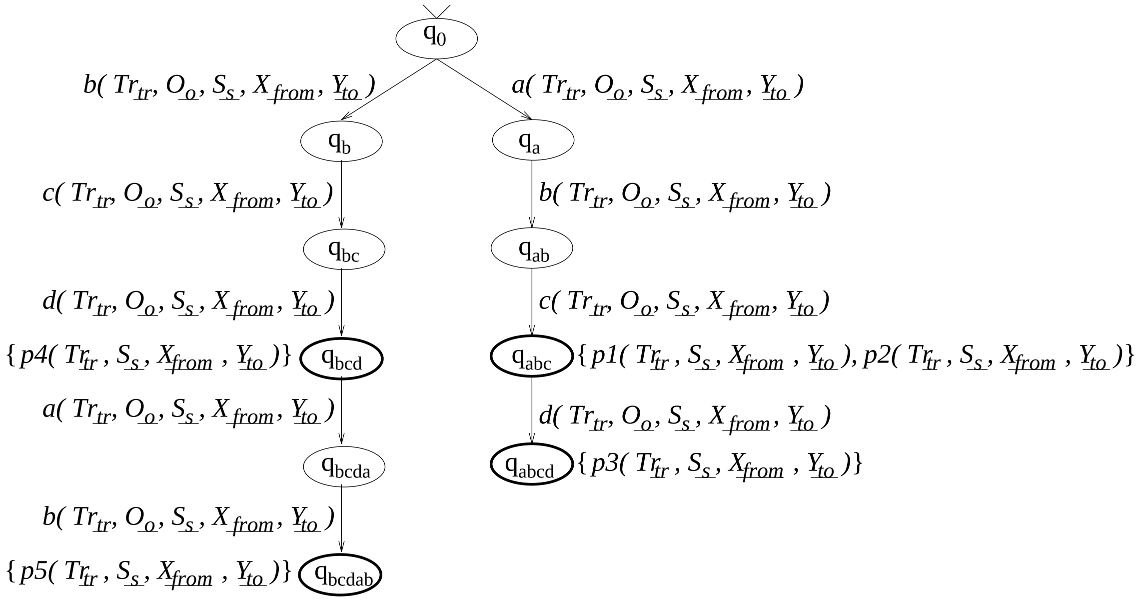


Figure 4: Prefix tree acceptor $PA$

$\mathbf{P}_\mathrm{E}$, and outputs one or several ground instances of sensor feature predicates, whenever one of its final states is reached. These are exactly those, which are derivable from the original program $\mathbf{P}_\mathrm{I}$, i.e., which are in the minimum Herbrand model of $\mathbf{P} = \mathbf{P}_\mathrm{I} \cup \mathbf{P}_\mathrm{E}$ equal to $\mathrm{T}^{\omega}_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. This is shown in Section 5.

**Related Work** The construction of a prefix (tree) acceptor has been first proposed by Angluin for the propositional case in [1]. Here, we have extended the construction to an acceptor, which works on chain Datalog rules. Structuring chain Datalog rules in a prefix acceptor allows for a fast forward inference method, which avoids the redundant evaluation of the same EDB fact with respect to similar rules. This inference method is the topic of Section 5.

## 4.3 Restructuring chain Datalog programs

In this section, we show how a program $\mathbf{P}_{\mathrm{I}}$, which satisfies conditions C1, C2, and C3, can be restructured yielding a program $\mathbf{P}'_{\mathrm{I}}$, such that $\mathbf{P}'_{\mathrm{I}}$ is equivalent to $\mathbf{P}_{\mathrm{I}}$ with respect to $\mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$ (see Definition 2 in 3.2). The rules of $\mathbf{P}'_{\mathrm{I}}$ have a special syntactical form, which allows to map them directly to a prefix acceptor.

The restructured program $\mathbf{P}'_{\mathrm{I}}$ has an inference depth which is greater than one. During the restructuring process new IDB predicates are introduced. So, from a machine learning point of view, we introduce new, possibly meaningful concepts, without changing the coverage of the original target concepts.

Again, we present two methods. The first one, `restruct` ( 4.3.1), exploits the syntactical characteristics of chain Datalog rules. The second one, `restruct_dc` ( 4.3.2), is more efficient, but requires, like `sort_dc`, the background knowledge about the data classes, to which the rules belong.

We proof the equivalence of the original program $\mathbf{P}_{\mathrm{I}}$ and the restructured program $\mathbf{P}'_{\mathrm{I}}$ ( 4.3.3). Then, we show, how the rules of $\mathbf{P}'_{\mathrm{I}}$ can be mapped to the prefix acceptor, yielding the same result as the application of the method `prefix_tree` to the original program $\mathbf{P}_{\mathrm{I}}$ ( 4.3.4).

### 4.3.1 The `restruct`-method

The procedure `restruct` takes as input a non-recursive basic chain Datalog program $\mathbf{P}_{\mathrm{I}}$ with rules, whose premises are sorted according to the relation $\ll$ and whose IDB predicates occur only in rule heads, i.e., its inference depth is one. For each rule, the relation $\leadsto$, which is defined by its chaining variables and its left and right block has to be intransitive, irreflexive and asymmetric. Furthermore, it is not allowed, that two premise literals have the same starting and ending variable. The procedure generates a modified basic chain Datalog program $\mathbf{P}'_{\mathrm{I}}$, which is equivalent to $\mathbf{P}_{\mathrm{I}}$ with respect to $\mathcal{I} \subseteq IDB(\mathbf{P}_{\mathrm{I}})$. The resulting program $\mathbf{P}'_{\mathrm{I}}$ has inference depth greater one. Furthermore, $EDB(\mathbf{P}_{\mathrm{I}}) = EDB(\mathbf{P}'_{\mathrm{I}})$ and $IDB(\mathbf{P}_{\mathrm{I}}) \subseteq IDB(\mathbf{P}'_{\mathrm{I}})$. One one hand, this program supports more efficient evaluations (see decompositions in Section 7), on the other hand it can be directly mapped to a deterministic finite prefix acceptor, which supports an even more efficient inference procedure.

We illustrate the method with the example program $\mathbf{P}_{\mathrm{I}}$

$$\begin{array}{lcl}
p_1(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). \\
p_2(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). \\
p_3(Tr,S,X,Y) & \leftarrow & a(Tr,O,S,X,X_1), b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,X_3), \\
 & & d(Tr,O,S,X_3,Y). \\
p_4(Tr,S,X,Y) & \leftarrow & b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,Y). \\
p_5(Tr,S,X,Y) & \leftarrow & b(Tr,O,S,X,X_1), c(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,X_3), \\
 & & a(Tr,O,S,X_3,X_4), b(Tr,O,S,X_4,Y).
\end{array}$$

For each EDB predicate $A = a(X_1,\ldots,X_m)$, which occurs as first element in some premise chain of a rule in $\mathbf{P}_{\mathrm{I}}$, we introduce a new IDB predicate symbol $q$, generate a predicate, which has the same arguments as $A$ and introduce the rule

$$q(X_1, \ldots, X_m) \leftarrow a(X_1, \ldots, X_m).$$

For our example program, we get

$$\begin{aligned} q_a(Tr, O, S, X, X1) &\leftarrow a(Tr, O, S, X, X_1). & (11)\\ q_b(Tr, O, S, X, X1) &\leftarrow b(Tr, O, S, X, X_1). & (12) \end{aligned}$$

We fold the rules of the program with the newly introduced rules, yielding the first intermediate result:

$$\begin{aligned} p_1(Tr, S, X, Y) &\leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y).\\ p_2(Tr, S, X, Y) &\leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y).\\ p_3(Tr, S, X, Y) &\leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, X_3),\\ &\quad\; d(Tr, O, S, X_3, Y).\\ p_4(Tr, S, X, Y) &\leftarrow q_b(Tr, O, S, X, X_1), c(Tr, O, S, X_1, X_2), d(Tr, O, S, X_2, Y).\\ p_5(Tr, S, X, Y) &\leftarrow q_b(Tr, O, S, X, X_1), c(Tr, O, S, X_1, X_2), d(Tr, O, S, X_2, X_3),\\ &\quad\; a(Tr, O, S, X_3, X_4), b(Tr, O, S, X_4, Y). \end{aligned}$$

Note, that here and in the following steps, if we fold the rules of the program with a new rule $Q \leftarrow A_1, \ldots, A_n, n \leq 2$, we replace $A_1, \ldots, A_n$ by $Q$ only if the chain$A_1, \ldots, A_n$ is a prefix of a premise chain. As long as the program has rules with more than two premise literals, we perform the second step: We select a rule $B \leftarrow A_1 A_2 A_3 \ldots A_n$, e.g.,

$$p_1(Tr, S, X, Y) \leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y).$$

Then, we generate a new rule $B^{new} \leftarrow A_1 A_2$. We generate a new predicate symbol for $B^{new}$ and determine its head variables. Goal of the restructuring process is to eliminate with the new rules those variables, which occur only in $A_1$ and $A_2$, and not in $B, A_i, i > 2$. We determine the variables shared by $A_1$ and $A_2$, $vars(A_1) \cap vars(A_n)$ (for our example, these are the variables $\{Tr, O, S, X_1\}$), remove the variables occurring in the head ($\{O, X_1\}$) and the variables occurring in $A_i, i > 2$ ($\{X_1\}$). This gives us the variables occurring only in $A_1$ and $A_2$, and thus should not occur in the head of the new rule. If we remove these variables from $vars(A_1) \cap vars(A_2)$, we get the variables, which we keep in the head $B^{new}$ ($\{Tr, O, S\}$). As the new rule has to be a chain Datalog rule, we have to determine the new left and right block. The potential variables for the left block are among the variables shared by $B$ and $A_1$, $vars(B) \cap vars(A_1)$ ($\{Tr, S, X\}$). We subtract from this set the variables occurring in $A_2 \ldots A_n$. This yields the potential candidates for the left block. The potential variables for the right block are determined from the variables shared by $A_2$ and $A_3$, $vars(A_2) \cap vars(A_3)$ ($\{Tr, O, S, X_2\}$). We remove those variables, which occur in $A_1, A_3, \ldots, A_n$ ($\{X_2\}$). This set contains the potential right blocks. It has to be different from the set for the left block (left and right block should not coincide). The variables for the head of the new rule are the variables to keep and the candidates for the left and right block. For our example program, we get the rules

$$q_{ab}(Tr, O, S, X, X_2) \leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2) \tag{13}$$

and

$$q_{bc}(Tr, O, S, X, X_2) \leftarrow q_b(Tr, O, S, X, X_1), c(Tr, O, S, X_1, X_2) \tag{14}$$

We fold the rules of the intermediate program with rules r13 and r14 and get

$$\begin{aligned}
p_1(Tr, S, X, Y) &\leftarrow q_{ab}(Tr, O, S, X, X_2), c(Tr, O, S, X_2, Y).\\
p_2(Tr, S, X, Y) &\leftarrow q_{ab}(Tr, O, S, X, X_2), c(Tr, O, S, X_2, Y).\\
p_3(Tr, S, X, Y) &\leftarrow q_{ab}(Tr, O, S, X, X_2), c(Tr, O, S, X_2, X_3), d(Tr, O, S, X_3, Y).\\
p_4(Tr, S, X, Y) &\leftarrow q_{bc}(Tr, O, S, X, X_2), d(Tr, O, S, X_2, Y).\\
p_5(Tr, S, X, Y) &\leftarrow q_{bc}(Tr, O, S, X, X_2), d(Tr, O, S, X_2, X_3), a(Tr, O, S, X_3, X_4),\\
&\qquad b(Tr, O, S, X_4, Y).
\end{aligned}$$

If we repeat the process until there are no more rules with more than two premises, we get

$$q_{abc}(Tr, O, S, X, Y) \leftarrow q_{ab}(Tr, O, S, X, X_2), c(Tr, O, S, X_2, Y). \tag{15}$$

$$q_{bcd}(Tr, O, S, X, Y) \leftarrow q_{bc}(Tr, O, S, X, X_2), c(Tr, O, S, X_2, Y). \tag{16}$$

By folding the intermediate rules with r15 and r16, we get

$$p_1(Tr, S, X, Y) \leftarrow q_{abc}(Tr, O, S, X, Y). \tag{17}$$

$$p_2(Tr, S, X, Y) \leftarrow q_{abc}(Tr, O, S, X, Y). \tag{18}$$

$$p_3(Tr, S, X, Y) \leftarrow q_{abc}(Tr, O, S, X, Y), d(Tr, O, S, X_3, Y).$$

$$p_4(Tr, S, X, Y) \leftarrow q_{bcd}(Tr, O, S, X, Y). \tag{19}$$

$$p_5(Tr, S, X, Y) \leftarrow q_{bcd}(Tr, O, S, X, X_3), a(Tr, O, S, X_3, X_4), b(Tr, O, S, X_4, Y).$$

Rules r17, r18, and r19 do not need any further consideration. With

$$q_{bcda}(Tr, O, S, X, X_4) \leftarrow q_{bcd}(Tr, O, S, X, X_3), a(Tr, O, S, X_3, X_4). \tag{20}$$

we get the folded rules

$$\begin{aligned}
p_3(Tr, S, X, Y) &\leftarrow q_{abc}(Tr, O, S, X, X_3), d(Tr, O, S, X_3, Y).\\
p_5(Tr, S, X, Y) &\leftarrow q_{bcda}(Tr, O, S, X, X_4), b(Tr, O, S, X_4, Y).
\end{aligned}$$

Now, we are left with rules of the form $B \leftarrow A_1, A_2$. Note that in this case we do not have other premise atoms, in order to determine the variables to keep. If we determine $vars(A_1) \cap vars(A_2)$ (for the rule with the head predicate $p_3$, we get $(\{Tr, O, S, X_3\})$ and remove the head variables $(\{O, X_3\})$, we have the variables, which should not occur in the head of the new rule. So, for our example, in contrast to the case with three or more premise literals, the variable $O$ does not appear in the head of the new rule. We get as new rules

$$q_{abcd}(Tr, S, X, Y) \leftarrow q_{abc}(Tr, O, S, X, X_3), d(Tr, O, S, X_3, Y). \tag{21}$$

$$q_{bcdab}(Tr, S, X, Y) \leftarrow q_{bcda}(Tr, O, S, X, X_4), b(Tr, O, S, X_4, Y). \tag{22}$$

and end up with the folded rules

$$p_3(Tr, S, X, Y) \leftarrow q_{abcd}(Tr, S, X, Y). \tag{23}$$

$$p_5(Tr, S, X, Y) \leftarrow q_{bcdab}(Tr, S, X, Y). \tag{24}$$

The restructured program consists of the rules r11,...,r24.

```
restruct(P_I)
begin
  1. restruct_init(P_I, ToDo, Done);
  2. restruct3(ToDo, ToDo1, Done1);
  3. restruct2(ToDo1, Done2);
  4. return P'_I := Done ∪ Done1 ∪ Done2;
end
```

Algorithm 3: `restruct`

```
restruct_init(Rules, ToDo, Done)
begin
  1. Done := ∅;
  2. ToDo := Rules;
  3. EDBS := set of all EDB predicates of Rules;
  4. while there exists C ∈ ToDo with C = B ← A_1, ..., A_n, and A_1 is a literal over a predicate
         in EDB
     (a) q :=new_predicate_symbol;
     (b) Head:= new_atom(q, vars(A_1));
     (c) Done := Done ∪ {Head ← A_1};
     (d) ToDo := fold(ToDo, Head ← A_1);
end
```

Algorithm 4: `restruct_init`

### 4.3.2 The `restruct_dc`-method

In the same way, as we have implemented a more efficient sorting method, we have implemented a more efficient restructuring method, `restruct_dc`, which requires the background knowledge about the data classes, to which the predicates of the rules belong.

In the first step, it introduces, just like the method `restruct`, for each EDB predicate $A = a(X_1, \ldots, X_m)$, which occurs as first element in some premise chain of $\mathbf{P}_\mathrm{I}$, a new rule with a new head predicate, which belongs to the same data class as $A$ and has the same property values as $A$. In the second step, we try to introduce for each rule with at least two premise literals $C \leftarrow A_1, \ldots, A_n, n \geq 2$, a new rule $Q \leftarrow A_1, A_2$ in the following way. The method is provided with the background knowledge, to which data class $Q$ is to belong, if $A_1$ and $A_2$ belong to specific data classes. In our case, if $A_1$ and $A_2$ are basic feature predicates, $Q$ will also be a basic feature predicate. Furthermore, the method is provided with the background knowledge, which property values the new predicate $Q$ "inherits" from the predicates $A_1$ and $A_2$. In our case, these are the trace, orientation, sensor and starting point from $A_1$ and the end point from $A_2$. Given the rule

$$p_1(Tr, S, X, Y) \leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y),$$

`restruct3`$(Rules, ToDo, Done)$
**begin**

1. $Done := \emptyset$;
2. $ToDo := Rules$;
3. **while** there exists $C \in ToDo$ such that $C = B \leftarrow A$ or $C = B \leftarrow A_1 A_2 A_3 \ldots A_n$
   **if** $C = B \leftarrow A$ **then**
   (a) $Done := Done \cup \{C\}$;
   (b) $ToDo := ToDo - \{C\}$;

   **else**
   (a) $EliminateVars := (vars(A_1) \cap vars(A_2)) - vars(B) - vars(A_3, \ldots, A_n)$;
   (b) $KeepVars := (vars(A_1) \cap vars(A_2)) - EliminateVars$;
   (c) $LeftVars := (vars(B) \cap vars(A_1)) - vars(A2, \ldots, A_n)$;
   (d) $RightVars := (vars(A_1) \cap vars(A_2)) - vars(A_1, A_3, \ldots, A_n)$;
   (e) $q :=$`new_predicate_symbol`;
   (f) $HeadVars := KeepVars \cup LeftVars \cup RightVars$;
   (g) $Head :=$ `new_atom`$(q, HeadVars)$;
   (h) $Done := Done \cup \{Head \leftarrow A_1, A_2\}$;
   (i) $ToDo :=$ `fold`$(ToDo, Head \leftarrow A_1, A_2)$;

**end**

Algorithm 5: `restruct3`

with $A_1 = q_a(Tr, O, S, X, X_1)$ and $A_2 = b(Tr, O, S, X_1, X_2)$, we get

$$Q = q_{ab}(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}})\kappa_{\mathbf{SF}} = q_{ab}(Tr, O, S, X, X_2)$$

with $\kappa_{\mathbf{SF}} = \{\mathtt{tr}_{\mathbf{BF}} = tr_{\mathbf{BF}}(A_1), \mathtt{o}_{\mathbf{BF}} = o_{\mathbf{BF}}(A_1), \mathtt{s}_{\mathbf{BF}} = s_{\mathbf{BF}}(A_1), \mathtt{from}_{\mathbf{BF}} = from_{\mathbf{BF}}(A_1), \mathtt{to}_{\mathbf{BF}} = to_{\mathbf{BF}}(A_2)\}$, i.e., we get the new rule

$$q_{ab}(Tr, O, S, X, X_2) \leftarrow q_a(Tr, O, S, X, X_1), b(Tr, O, S, X_1, X_2).$$

If we fold the program rules with a new rule $Q \leftarrow A_1, A_2$, we replace $A_1$ and $A_2$ only by $Q$, if they occur as first premise literals of a premise chain. The pseudo-code for the `restruct_dc`-method is given in Algorithm 13 in Appendix A.2.

If we apply `restruct_dc` to the example program $\mathbf{P}_{\mathrm{I}}$, we get rules r11, ..., r20. The difference between `restruct_dc` and `restruct` lies in the treatment of rules with exactly two premise literals. For the folded rules

$$\begin{aligned} p_3(Tr, S, X, Y) &\leftarrow q_{abc}(Tr, O, S, X, X_3), d(Tr, O, S, X_3, Y). \\ p_5(Tr, S, X, Y) &\leftarrow q_{bcda}(Tr, O, S, X, X_4), b(Tr, O, S, X_4, Y). \end{aligned}$$

the rules

$$\begin{aligned} q_{abcd}(Tr, \boldsymbol{O}, S, X, Y) &\leftarrow q_{abc}(Tr, O, S, X, X_3), d(Tr, O, S, X_3, Y). \\ q_{bcdab}(Tr, \boldsymbol{O}, S, X, Y) &\leftarrow q_{bcda}(Tr, O, S, X, X_4), b(Tr, O, S, X_4, Y). \end{aligned}$$

---

`restruct2`$(Rules, Done)$
**begin**

1. $Done := \emptyset$;
2. $ToDo := Rules$;
3. **while** $ToDo \neq \emptyset$
   select a rule $C \in ToDo$
   **if** $C = B \leftarrow A$ **then**
   (a) $Done := Done \cup \{C\}$;
   (b) $ToDo := ToDo - \{C\}$;

   **else**
   (a) $EliminateVars := (vars(A_1) \cap vars(A_2)) - vars(B)$;
   (b) $KeepVars := vars(A_1) \cap vars(A_2) - EliminateVars$;
   (c) $LeftVars := (vars(B) \cap vars(A_1)) - vars(A2)$;
   (d) $RightVars := (vars(B) \cap vars(A_2)) - vars(A_1)$;
   (e) $q :=$ `new_predicate_symbol`;
   (f) $HeadVars := KeepVars \cup LeftVars \cup RightVars$;
   (g) $Head :=$ `new_atom`$(q, HeadVars)$;
   (h) $Done := Done \cup \{Head \leftarrow A_1, A_2\}$;
   (i) $ToDo :=$ `fold`$(ToDo, Head \leftarrow A_1, A_2)$;

**end**

Algorithm 6: `restruct2`

---

are introduced (instead of rules r21 and r22), yielding the folded rules

$$\begin{array}{rcl} p_3(Tr, S, X, Y) & \leftarrow & q_{abcd}(Tr, \boldsymbol{O}, S, X, Y). \\ p_5(Tr, S, X, Y) & \leftarrow & q_{bcdab}(Tr, \boldsymbol{O}, S, X, Y). \end{array}$$

(instead of rules r23 and r24). So, to summarize, we get as result the restructured program $\mathbf{P}'_{\mathrm{I}}$

$$\begin{array}{rcl} q_a(Tr, O, S, X, Y) & \leftarrow & a(Tr, O, S, X, Y). \\ q_b(Tr, O, S, X, Y) & \leftarrow & b(Tr, O, S, X, Y). \\ q_{ab}(Tr, O, S, X, Y) & \leftarrow & q_a(Tr, O, S, X_1, X_2), b(Tr, O, S, X_2, Y). \\ q_{bc}(Tr, O, S, X, Y) & \leftarrow & q_b(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y). \\ q_{abc}(Tr, O, S, X, Y) & \leftarrow & q_{ab}(Tr, O, S, X_1, X_2), c(Tr, O, S, X_2, Y). \\ q_{bcd}(Tr, O, S, X, Y) & \leftarrow & q_{bc}(Tr, O, S, X_1, X_2), d(Tr, O, S, X_2, Y). \\ p_1(Tr, S, X, Y) & \leftarrow & q_{abc}(Tr, O, S, X, Y). \\ p_2(Tr, S, X, Y) & \leftarrow & q_{abc}(Tr, O, S, X, Y). \\ p_3(Tr, S, X, Y) & \leftarrow & q_{abcd}(Tr, O, S, X, Y). \\ q_{abcd}(Tr, O, S, X, Y) & \leftarrow & q_{abc}(Tr, O, S, X_1, X_2), d(Tr, O, S, X_2, Y). \\ q_{bcda}(Tr, O, S, X, Y) & \leftarrow & q_{bcd}(Tr, O, S, X_1, X_2), a(Tr, O, S, X_2, Y). \end{array}$$

$$
\begin{array}{rcl}
p_4(Tr,S,X,Y) & \leftarrow & q_{bcd}(Tr,O,S,X,Y). \\
q_{bcdab}(Tr,O,S,X,Y) & \leftarrow & q_{bcda}(Tr,O,S,X_1,X_2), b(Tr,O,S,X_2,Y). \\
p_5(Tr,S,X,Y) & \leftarrow & q_{bcdab}(Tr,O,S,Y,Y).
\end{array}
$$

The grammar which corresponds to $\mathbf{P}'_\mathrm{I}$ is $G' = (V', \Sigma', P', s)$ with $V' = \{s, p_1, p_2, p_3, p_4, p_5, q_a, q_{ab}, q_{abc}, q_{abcd}, q_b, q_{bc}, q_{bcd}, q_{bcda}, q_{bcdab}\}$ and $\Sigma' = \{a, b, c, d\}$. $P'$ is the set of productions

$$
\begin{array}{rclcrcl}
s & \rightarrow & p_1|p_2|p_3|p_4|p_5 & & & & \\
p_1 & \rightarrow & q_{abc} & & q_{bcda} & \rightarrow & q_{bcd}\ a \\
p_2 & \rightarrow & q_{abc} & & q_{abc} & \rightarrow & q_{ab}\ c \\
p_3 & \rightarrow & q_{abcd} & & q_{bcd} & \rightarrow & q_{bc}\ d \\
p_4 & \rightarrow & q_{bcd} & & q_{ab} & \rightarrow & q_a\ b \\
p_5 & \rightarrow & q_{bcdab} & & q_{bc} & \rightarrow & q_b\ c \\
q_{bcdab} & \rightarrow & q_{bcda}\ b & & q_a & \rightarrow & a \\
q_{abcd} & \rightarrow & q_{abc}\ d & & q_b & \rightarrow & b
\end{array}
$$

Remember, that $G'$ has already been derived from the DFA accepting the language $\hat{L}(G)$ (see 4.2). $G$ is the grammar corresponding to the original program $\mathbf{P}_\mathrm{I}$.

### 4.3.3 Equivalence of the restructured program

**Lemma 2** *Let $\mathbf{P}_\mathrm{I}$ be a non-recursive basic chain Datalog program with rules, where the IDB predicates occur only in rule heads. Let $\mathbf{P}'_\mathrm{I}$ be the program which results from restructuring $\mathbf{P}_\mathrm{I}$ with either* `restruct` *or* `restruct_dc`*. Then, for a given EDB instance $\mathbf{P}_\mathrm{E}$*

$$
\begin{array}{c}
\{p_i(t_1,\ldots,t_s) \quad | \quad p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in T^\omega_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})\} \\
= \\
\{p_i(t_1,\ldots,t_s) \quad | \quad p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in T^\omega_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})\}
\end{array}
$$

*with $\mathcal{I} \subseteq$IDB($\mathbf{P}_\mathrm{I}$), i.e., the coverage for the target predicates $p_i \in \mathcal{I}$ is the same for $\mathbf{P}_\mathrm{I}$ and $\mathbf{P}'_\mathrm{I}$.*

**Proof** As IDB($\mathbf{P}_\mathrm{I}$) $\subseteq$ IDB($\mathbf{P}'_\mathrm{I}$), it suffices for the $\subseteq$-part to show that $\mathrm{T}^\omega_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E}) \subseteq \mathrm{T}^\omega_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. We know, that the inference depth of $\mathbf{P}_\mathrm{I}$ is 1, i.e., $\mathrm{T}^\omega_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E}) = \mathrm{T}^1_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. Let $\breve{B} \in \mathrm{T}^1_{\mathbf{P}_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. Then, there exists a $C \in \mathbf{P}_\mathrm{I}$ such that $C\sigma = (\breve{B} \leftarrow \breve{A}_1, \ldots, \breve{A}_n)$ and $\breve{A}_1, \ldots, \breve{A}_n \in \mathbf{P}_\mathrm{E}$. We have to show, that $\breve{B} \in \mathrm{T}^\omega_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. For the rule $C = (B \leftarrow A_1, \ldots, A_n)$, the restructuring method has produced $n+1$ rules,

$$
\begin{array}{rcl}
C_1 = (Q_1 & \leftarrow & A_1) \\
C_2 = (Q_2 & \leftarrow & Q_1 A_2) \\
C_3 = (Q_3 & \leftarrow & Q_2 A_3)
\end{array}
$$

$$\begin{array}{rcl} & \dots & \\ C_n = (Q_n & \leftarrow & Q_{n-1}A_n) \\ C_{n+1} = (B & \leftarrow & Q_n). \end{array}$$

As $\breve{A}_1 \in \mathbf{P}_\mathrm{E}$, $\breve{Q}_1 \in \mathrm{T}^1_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$, as $\breve{A}_2 \in \mathbf{P}_\mathrm{E}$, $\breve{Q}_2 \in \mathrm{T}^2_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$, ..., as $\breve{A}_n \in \mathbf{P}_\mathrm{E}$, $\breve{Q}_n \in \mathrm{T}^n_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$, and $\breve{B} \in \mathrm{T}^{n+1}_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$.

The $\supseteq$-part   Let $\breve{B} = p_r(t_1, \dots, t_s) \in \{p_x(t_1, \dots, t_s) | p_x \in \mathcal{I} \text{ and } p_x(t_1, \dots, t_s) \in \mathrm{T}^{i+1}_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})\}$. Then, there is a rule $C \in \mathbf{P}'_\mathrm{I}$, such that $C\sigma = (\breve{B} \leftarrow \breve{Q}_l)$ with $\breve{Q}_l \in \mathrm{T}^i_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$. We have a sequence of rules

$$\begin{array}{rcl} C_l \in \mathbf{P}'_\mathrm{I}, & \text{such that } C_l\sigma_l = (\breve{Q}_l \leftarrow \breve{Q}_{l-1}\breve{A}_l) & \text{with } \breve{A}_l \in \mathbf{P}_\mathrm{E} \\ C_{l-1} \in \mathbf{P}'_\mathrm{I}, & \text{such that } C_{l-1}\sigma_{l-1} = (\breve{Q}_{l-1} \leftarrow \breve{Q}_{l-2}\breve{A}_{l-1}) & \text{with } \breve{A}_{l-1} \in \mathbf{P}_\mathrm{E} \\ & \dots & \\ C_1 \in \mathbf{P}'_\mathrm{I}, & \text{such that } C_1\sigma_1 = (\breve{Q}_1 \leftarrow \breve{A}_1) & \text{with } \breve{A}_1 \in \mathbf{P}_\mathrm{E} \end{array}$$

If we unfold $C$, we get the rule $C_{unfolded} = (B \leftarrow A_1, \dots, A_l)$ with $C_{unfolded} \in \mathbf{P}_\mathrm{I}$. As $\breve{A}_1, \dots, \breve{A}_l \in \mathbf{P}_\mathrm{E}$, it follows that $\breve{B} \in \mathrm{T}^\omega_{\mathbf{P}'_\mathrm{I}}(\mathbf{P}_\mathrm{E})$.□

### 4.3.4 Mapping the restructured program to a prefix acceptor

The rules of a program $\mathbf{P}'_\mathrm{I}$, which is the result of the restructuring methods presented above, have one of following syntactical forms:

$$\begin{array}{rcll} q_a(Tr, O, S, X, Y) & \leftarrow & a(Tr, O, S, X, Y) & (25) \\ q_j(Tr, O, S, X, Y) & \leftarrow & q_i(Tr, O, S, X, X_1), a(Tr, O, S, X_1, Y) & (26) \\ p_r(Tr, S, X, Y) & \leftarrow & q_s(Tr, O, S, X, Y) & (27) \end{array}$$

with $a \in$ EDB($\mathbf{P}_\mathrm{I}$)=EDB($\mathbf{P}'_\mathrm{I}$), $q_i \in$ IDB($\mathbf{P}'_\mathrm{I}$) $- \mathcal{I}$, and $p_r \in \mathcal{I} \subseteq$ IDB($\mathbf{P}'_\mathrm{I}$). The prefix acceptor is defined by the tuple $(Q, \Sigma, Z, \Delta, q_0, F, \lambda)$. Given $\mathbf{P}'_\mathrm{I}$, we map the EDB predicates to the set of input predicates $\Sigma$, i.e.,

$$\Sigma = \{a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}) | a \in \mathrm{EDB}(\mathbf{P}'_\mathrm{I})\}.$$

The IDB predicates in $\mathcal{I}$ are mapped to the set of output predicates, i.e.,

$$Z = \{p_i(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}) | p_i \in \mathcal{I}\}.$$

For each $q_i \in$ IDB($\mathbf{P}'_\mathrm{I}$) $- \mathcal{I}$ we establish a state for the prefix acceptor, i.e.,

$$Q = \{q_i | q_i \in \mathrm{IDB}(\mathbf{P}'_\mathrm{I}) - \mathcal{I}\} \cup \{q_0\},$$

where $q_0$ is a newly introduced symbol for the starting state. Each $q_i$, which appears as IDB subgoal in a rule of form (27) is a final state, i.e.,

$$F = \{q_i \mid q_i \in \mathrm{IDB}(\mathbf{P}'_\mathrm{I}) - \mathcal{I} \text{ and } p_r(Tr, S, X, Y) \leftarrow q_i(Tr, O, S, X, Y) \in \mathbf{P}'_\mathrm{I}\}.$$

For each rule of form (25) we establish a transition from the starting state to $q_i$, i.e.,

$$(q_0, a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_i) \in \Delta.$$

For each rule of form (26), we establish a transition from state $q_i$ to state $q_j$

$$(q_i, a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_j) \in \Delta.$$

For each rule of form (27), we add $p_r(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}})$ to the set, to which the function $\lambda$ maps state $q_s$

$$\lambda(q_s) = \{p_r(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}) | p_r \in \mathcal{I} \text{ and } p_r(Tr, S, X, Y) \leftarrow q_s(Tr, O, S, X, Y) \in \mathbf{P}'_{\mathrm{I}}\}.$$

If we map the restructured example program $\mathbf{P}'_{\mathrm{I}}$ to a prefix acceptor, we get the $PA$ illustrated in Figure 4. Note, that this mapping procedure can be applied to any linear program (see Definition 1 in Section 3) with rules with at most one EDB subgoal, i.e., it is not restricted to non-recursive programs (see Section 6.3 for an example).

Furthermore, each prefix acceptor can be (re-) transformed to a basic chain Datalog program by introducing the respective chain rules for the transition and output function.

#### 4.3.5 Related work

Sommer has presented in [23] a method for theory restructuring, called `FENDER`. It restructures the rules for one concept, whereas we restructure the rules for several concepts, which share a lot of common features. `FENDER` searches for common partial premises (CPPs), each of which is collected around one variable, which appears only in a rule body. Given a chain Datalog program with rules, such as

$$p(Tr, S, X, Y) \leftarrow r_1(Tr, O, S, X, X_1), \ldots, r_{k+1}(Tr, O, S, X_k, Y),$$

`FENDER` would consider the whole premise chain, collected around the variable $O$, and the set of $k$ overlapping CPPs of the form $r_i(Tr, O, S, X_{i-1}, X_i), r_{i+1}(Tr, O, S, X_i, X_{i+1})$, collected around the chaining variables $X_i, i = 1, \ldots, k$, as candidates for intermediate concepts. Neither of these is what we are aiming at.

The restructuring method `restruct` implements, in principle, the W-operator, which was introduced by Muggleton, as inter-construction operator ([14], [15]), which in [13] is called an inductive inference rule

$$\textbf{Inter-construction:} \qquad \frac{p \leftarrow G, H \qquad q \leftarrow G, K}{p \leftarrow r, H \quad r \leftarrow G \quad q \leftarrow r, K},$$

where $p$ and $q$ represent propositional constants and $G, H$ and $K$ conjunctions of propositional constants. The method `restruct` implements three specific inter-construction steps for chain Datalog rules

$$\textbf{Step 1:} \qquad \frac{B \leftarrow A_1, A_2, \ldots, A_n}{B \leftarrow Q_1, A_2, \ldots, A_n \qquad Q_1 \leftarrow A_1}$$

$$\textbf{Step 2:} \qquad \frac{B \leftarrow A_1, A_2, \ldots, A_n \qquad C \leftarrow A_1, L_2, \ldots, L_m}{B \leftarrow Q_1, A_2, \ldots, A_n \quad Q_1 \leftarrow A_1 \quad C \leftarrow Q_1, L_2, \ldots, L_m}$$

$$\textbf{Step 3:} \qquad \frac{B \leftarrow A_1, A_2}{B \leftarrow Q \quad Q \leftarrow A_1, A_2},$$

where $A_i, Q_j, L_k$, and $B$ represent atoms.

# 5 MP: An Efficient Forward Inference Method

The marker passing method, which we present in this section, has already been introduced in [19] and [20]. As the main focus in these papers was on a related topic, we omitted some details of the method, which we want to elaborate here from a logic programming point of view. Marker passing methods have been developed, e.g., by Charniak [5] and Hendler [8]. We present a marker passing method, which is applied to a prefix acceptor, and which calculates (part of) the minimum Herbrand model of the chain Datalog program which has been mapped to the prefix acceptor.

## 5.1 The marker passing method

Assume, that a basic chain Datalog program $\mathbf{P}_{\mathrm{I}}$ has been mapped to a prefix (tree) acceptor $PA$. Given an EDB instance, $\mathbf{P}_{\mathrm{E}}$, which is a ground chain, the goal is, from the logic programming point of view, to calculate the minimum Herbrand model of the extended logic program $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$. In this section, we present a marker passing method, called **MP**, which is applied to the prefix acceptor $PA$, in order to calculate the minimum Herbrand model via forward inferences. The EDB instance $\mathbf{P}_{\mathrm{E}}$ is required to be a ground chain of atoms over the EDB predicates (here, the predicates in $PS_{\mathbf{BF}}$), and is denoted $\breve{A}^1 \ldots \breve{A}^k$. Remember, that in our robotics domain $\breve{A}^1 \ldots \breve{A}^k$ represents a sequence of chronologically ordered observations, i.e., basic features, from which sensor features are to be derived. The important point to note, is, that the EDB instance $\mathbf{P}_{\mathrm{E}}$ is generated incrementally, while the robot moves through the environment, i.e., for each time point $1 \leq t \leq k$, we have $\mathbf{P}^t_{\mathrm{E}} = \mathbf{P}^{t-1}_{\mathrm{E}} \cup \{\breve{A}^t\}$, where $\mathbf{P}^0_{\mathrm{E}} = \emptyset$. However, at each time point $t = 1, 2, \ldots$, $\mathbf{P}^t_{\mathrm{E}}$ is finite and so is the Herbrand base $\mathcal{B}_H(\mathbf{P}^t)$ of $\mathbf{P}^t = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}^t_{\mathrm{E}}$. Now, let $\mathbf{MP}(\breve{A}^1 \ldots \breve{A}^k)$ denote the output of the marker passing method for the last element of the chain $\breve{A}^k$. **MP** calculates at each time point $t, 1 \leq t \leq k$, the IDB-portion of the minimum Herbrand model for the $p_i \in \mathcal{I}$, such that $\mathbf{MP}(\breve{A}^1) \cup \mathbf{MP}(\breve{A}^1\breve{A}^2) \cup \ldots \cup \mathbf{MP}(\breve{A}^1\breve{A}^2 \ldots \breve{A}^k)$ is a subset of the fixpoint of the mapping $\mathrm{T}_{\mathbf{P}_{\mathrm{I}}}$, applied to $\mathbf{P}^t_{\mathrm{E}}$, i.e., $\mathrm{T}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}^t_{\mathrm{E}})$. Remember, that the $PA$ can be mapped back to a program $\mathbf{P}'_{\mathrm{I}}$ and that according to Lemma 2, we have $\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} = \{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}$.

The method exploits the special syntax of the chain Datalog rules and uses the constraints introduced in 3.5, in order to set the arguments/property values of the predicates associated with the transitions and final states of the $PA$.

In the following, we use $A$ to denote an atom over a basic feature predicate in $PS_{\mathbf{BF}}$, and $B, B_i, i = 1, 2, \ldots$ to denote an atom over a sensor feature predicate in $PS_{\mathbf{SF}}$. Given a ground chain $\breve{A}^1 \ldots \breve{A}^k$, we know the following:

**Initialization of basic feature constraints:** Given the first atom, $\breve{A}^1$, the following equations have to be satisfied:

$$
\begin{aligned}
tr_{\mathbf{BF}}(\breve{A}^1) &= tr_{\mathbf{BF}}(\breve{A}^2) &= tr_{\mathbf{BF}}(\breve{A}^3) = \ldots \\
o_{\mathbf{BF}}(\breve{A}^1) &= o_{\mathbf{BF}}(\breve{A}^2) &= o_{\mathbf{BF}}(\breve{A}^3) = \ldots \\
s_{\mathbf{BF}}(\breve{A}^1) &= s_{\mathbf{BF}}(\breve{A}^2) &= s_{\mathbf{BF}}(\breve{A}^3) = \ldots,
\end{aligned}
$$

i.e., the trace, orientation and sensor of each member of the sequence of basic feature predicates $\breve{A}^1 \dots \breve{A}^k$, beginning with $\breve{A}^1$, have to be the same. Given these equations and an atom $\breve{A}^i$ over a basic feature predicate, we can set the constraints, which all following basic feature atoms $\breve{A}^i, i > 1$ have to satisfy, to

$$\kappa_{\mathbf{BF},init} = \{\mathtt{tr}_{\mathbf{BF}} = tr_{\mathbf{BF}}(\breve{A}^i), \mathtt{o}_{\mathbf{BF}} = o_{\mathbf{BF}}(\breve{A}^i), \mathtt{s}_{\mathbf{BF}} = s_{\mathbf{BF}}(\breve{A}^i)\}.$$

The function `init_bf_constraints` (see Algorithm 7) returns these initial basic feature constraints for a given $\breve{A}^i$. Given the atom $\breve{A} = a(t1, 90, s5, 1, 8)$, we get

$$\kappa_{\mathbf{BF},init} = \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}.$$

**Initialization of sensor feature constraints:** For any atom $B$, which is derivable from the chain beginning with $\breve{A}^1$, the following equations have to be satisfied:

$$\begin{aligned} tr_{\mathbf{BF}}(\breve{A}^1) &= tr_{\mathbf{SF}}(B) \\ s_{\mathbf{BF}}(\breve{A}^1) &= s_{\mathbf{SF}}(B) \\ from_{\mathbf{BF}}(\breve{A}^1) &= from_{\mathbf{SF}}(B), \end{aligned}$$

i.e., the trace, sensor and starting point of an atom over a sensor feature predicate, derivable from the sequence $\breve{A}^1 \dots \breve{A}^k$ has to be the same as for $\breve{A}^1$. Given these equations and an atom $\breve{A}^i$, we can set the constraints, which an atom $B$ derivable from the chain beginning with $\breve{A}^1$ has to satisfy, to

$$\kappa_{\mathbf{SF},init} = \{\mathtt{tr}_{\mathbf{SF}} = tr_{\mathbf{BF}}(\breve{A}^i), \mathtt{s}_{\mathbf{SF}} = s_{\mathbf{BF}}(\breve{A}^i), \mathtt{from}_{\mathbf{SF}} = from_{\mathbf{BF}}(\breve{A}^i)\}.$$

The function `init_sf_constraints` returns these initial sensor feature constraints for $\breve{A}^i$. Given the atom $\breve{A} = a(t1, 90, s5, 1, 8)$, we get

$$\kappa_{\mathbf{SF},init} = \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}.$$

**Update of basic feature constraints:** Given an atom $\breve{A}^i$ of the chain, for the next atom $\breve{A}^{i+1}$, the equation

$$to_{\mathbf{BF}}(A^i) = from_{\mathbf{BF}}(A^{i+1})$$

has to be satisfied, i.e., the end point of the previous basic feature has to be the starting point of the next one. Given this equation and an atom $\breve{A}^i$, we can set the constraints for $\breve{A}^{i+1}$ to

$$\kappa_{\mathbf{BF},update} = \{\mathtt{from}_{\mathbf{BF}} = to_{\mathbf{BF}}(\breve{A}^i)\}.$$

The function `update_bf_constraints` returns these constraint for a $\breve{A}^i$. Given the atom $\breve{A} = a(t1, 90, s5, 1, 8)$, we get

$$\kappa_{\mathbf{BF},update} = \{\mathtt{from}_{\mathbf{BF}} = 8\}.$$

**Update of sensor feature constraints:** Given a chain $\breve{A}^1 \dots \breve{A}^i$, for any atom $B$, which is derivable from the current sequence starting with $\breve{A}^1$, the equation

$$to_{\mathbf{BF}}(A^i) = to_{\mathbf{SF}}(B)$$

has to be satisfied, i.e., the end point of the last basic feature has to coincide with the end point of the derived sensor feature $B$. Given this equation and an atom $\breve{A}^i$, we can set the update constraints, which the sensor feature has to satisfy, to

$$\kappa_{\mathbf{SF},update} = \{\mathtt{to}_{\mathbf{SF}} = to_{\mathbf{BF}}(\breve{A}^i)\}.$$

The function `update_sf_constraints` returns these sensor feature constraints for $\breve{A}^i$. Given the atom $\breve{A} = a(t1, 90, s5, 1, 8)$, it returns

$$\kappa_{\mathbf{SF},update} = \{\mathtt{to}_{\mathbf{SF}} = 8\}.$$

Now assume, that the robot perceives a sequence of ground atoms over basic feature predicates, $\breve{A}^1\breve{A}^2 \dots, \breve{A}^n, n \in N$. At each time point $1 \leq t \leq n$, the ground chain $\breve{A}^1\breve{A}^2 \dots \breve{A}^t$ denotes the EDB instance $\mathbf{P}_{\mathrm{E}}^t$ for the chain Datalog program $\mathbf{P}_{\mathrm{I}}$, which is compiled in the prefix acceptor $PA$. The marker passing method, **MP**, works as follows: At each time point $1 \leq t \leq n$, we check, whether there exists a transition from the starting state $q_0$, labeled $A \in \mathbf{BF}$, leading to state $q_j \in Q_{PA}$, such that $A$ is unifiable with $\breve{A}^t$. If that is the case, we generate a marker, which is associated with state $q_j$. It is represented by the tuple

$$(t, q_i, \kappa_{\mathbf{BF},init}, \kappa_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa_{\mathbf{SF},update}).$$

The constraints $\kappa_{\mathbf{BF}} = \kappa_{\mathbf{BF},init} \cup \kappa_{\mathbf{BF},update}$ are those, which the *next* basic feature $\breve{A}^{t+1}$ has to satisfy. The constraints $\kappa_{\mathbf{SF}} = \kappa_{\mathbf{SF},init} \cup \kappa_{\mathbf{SF},update}$ are those, which a sensor feature has to satisfy, if it is derivable from the sequence $\breve{A}^1\breve{A}^2 \dots \breve{A}^t$.

Each of the markers $m_r, 1 \leq r < t$, which has been generated at previous time points, is checked, whether it can be passed along a transition to a successor state. Let the information associated with marker $m_r$ at the previous time point $t-1$, be

$$m_r^{t-1} = (r, q^{t-1}, \kappa_{\mathbf{BF},init}, \kappa_{\mathbf{BF},update}^{t-1}, \kappa_{\mathbf{SF},init}, \kappa_{\mathbf{SF},update}^{t-1}).$$

Let $q^{t-1}(m_r) = q_i$ denote the state, with which a marker $m_r$ is associated at time point $t-1$. Then, we check, whether there exists a transition $(q_i, A, q_j)$, such that $A\kappa_{\mathbf{BF}}^{t-1}$ is unifiable with $\breve{A}^t$. If that is the case, we update $\kappa_{\mathbf{BF},update}^{t-1}$ and $\kappa_{\mathbf{SF},update}^{t-1}$ with respect to $\breve{A}^t$, yielding $\kappa_{\mathbf{BF},update}^{t}$ and $\kappa_{\mathbf{SF},update}^{t}$. So the marker info for $m_r$ at time point $t$ is

$$m_r^{t} = (r, q^{t} = q_j, \kappa_{\mathbf{BF},init}, \kappa_{\mathbf{BF},update}^{t}, \kappa_{\mathbf{SF},init}, \kappa_{\mathbf{SF},update}^{t}).$$

Finally, we have to check for each marker $m_s, 1 \leq s \leq t$, whether it is now associated with a final state $q \in F_{PA}$. If that is the case, we apply to each element in $\lambda(q) = \{B_1, \dots, B_l\}$ the constraints $\kappa_{\mathbf{SF}}^t = \kappa_{\mathbf{SF},init} \cup \kappa_{\mathbf{SF},update}^t$, yielding the ground atoms $\breve{B}_1\kappa_{\mathbf{SF}}^t, \dots, \breve{B}_l\kappa_{\mathbf{SF}}^t$, which are output and are part of the minimum Herbrand model of $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}^t$. The pseudo-code of the procedure **MP** is given in Algorithm 7. It takes as input a prefix acceptor $PA$ (representing a basic chain Datalog program $\mathbf{P}_{\mathrm{I}}$) and an EDB instance $\mathbf{P}_{\mathrm{E}}$, represented by the ground chain $\breve{A}^1\breve{A}^2\ldots,\breve{A}^k$. The prefix acceptor $PA$ is represented by the tuple $(Q_{PA},\Sigma_{PA},Z_{PA},\Delta_{PA},q_{0,PA},F,\lambda_{PA})$. The procedure outputs incrementally the IDB portion of the target predicates in $\mathcal{I}$ of the minimum Herbrand model for each subsequence $\breve{A}^1$, $\breve{A}^1\breve{A}^2$, $\breve{A}^1\breve{A}^2\breve{A}^3$, etc.

The important point to note is, that the marker passing method is much more efficient than a naive forward chaining procedure, which tries to match each $\breve{A}^i$ to each premise literal of each rule. In the case of marker passing, $\breve{A}^i$ has to be matched, depending on the transitions emanating from a given state, to at most $l$ predicates, where $l = |\mathrm{EDB}(\mathbf{P}_\mathrm{I})$—$= |\Sigma|$. A naive forward chaining inference procedure would try $y \gg l$ matches, where $y = \sum_{C\in\mathbf{P}_\mathrm{I}}|C_{body}|$. For a given fact $\breve{A}^t$, the markers $1,\ldots,t$ can be processed in parallel as they are independent of each other. The method terminates after $k$ steps, where $k$ is the length of the input chain $\mathbf{P}_\mathrm{E}$. The time and space requirements depend linearly on the length of the input chain $\mathbf{P}_\mathrm{E}$.

---

**MP**($PA, \mathbf{P}_\mathrm{E} = \breve{A}^1\breve{A}^2\ldots,\breve{A}^k$)
**for** $t = 1,2,\ldots$
**begin**

1. **if** there exists a $(q_0,A,q_k)\in\Delta_{PA}$, such that $\breve{A}^t$ is unifiable with $A$, **then**
   **begin**
   $\kappa_{\mathbf{BF},init}$ := `init_bf_constraints`($\breve{A}^t$);
   $\kappa_{\mathbf{SF},init}$ := `init_sf_constraints`($\breve{A}^t$);
   $\kappa^t_{\mathbf{BF},update}$ := `update_bf_constraints`($\breve{A}^t$);
   $\kappa^t_{\mathbf{SF},update}$ := `update_sf_constraints`($\breve{A}^t$);
   $m^t_t := (t, q_k, \kappa_{\mathbf{BF},init}, \kappa^t_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^t_{\mathbf{SF},update})$;
   **end**
2. **for** $r = 1,\ldots,t-1$
   % Let $m^{t-1}_r = (r, q^{t-1} = q_i, \kappa_{\mathbf{BF},init}, \kappa^{t-1}_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^{t-1}_{\mathbf{SF},update})$
   **begin**
   $\kappa^{t-1}_{\mathbf{BF}} := \kappa_{\mathbf{BF},init} \cup \kappa^{t-1}_{\mathbf{BF},update}$;
   **if** there exists a $(q_i,A,q_j)\in\Delta_{PA}$, such that $\breve{A}^t$ is unifiable with $A\kappa^{t-1}_{\mathbf{BF}}$, **then**
   **begin**
   $\kappa^t_{\mathbf{BF},update}$ := `update_bf_constraints`($\breve{A}^t$);
   $\kappa^t_{\mathbf{SF},update}$ := `update_sf_constraints`($\breve{A}^t$);
   $m^t_r := (r, q_j, \kappa_{\mathbf{BF},init}, \kappa^t_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^t_{\mathbf{SF},update})$;
   **end**
   **end**
3. **for** $s = 1,\ldots,t$
   **if** $q^t(m_s) = q_k \in F_{PA}$, **then**
   **begin**
   $\kappa^t_{\mathbf{SF}} := \kappa_{\mathbf{SF},init} \cup \kappa^t_{\mathbf{SF},update}$;
   **for** each $B_i \in \lambda(q_k) = \{B_1,\ldots,B_n\}$: output $B_i\kappa^t_{\mathbf{BF}}$;
   **end**

**end**

Algorithm 7: **MP**

---

**Example run of MP on $PA$:** If we apply the procedure **MP** to the prefix acceptor $PA$ in Figure 4 and the chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1\breve{A}^2\breve{A}^3\breve{A}^4$

$$\mathbf{P}_{\mathrm{E}} = \{a(t1, 90, s5, 1, 8), b(t1, 90, s5, 8, 10), c(t1, 90, s5, 10, 15), d(t1, 90, s5, 15, 17)\}$$

the markers are passed through the graph of the prefix acceptor as illustrated in Figure 5. Note, that the initial constraints for a sequence beginning with $\breve{A}^i$ are determined once, whereas the update constraints have to be updated once for each new member of the respective sequence. For each atom $\breve{A}^i$, the property values, which are updated via the constraints, are printed boldly.

**Input:** $\breve{A}^1 = a(t1, 90, s5, 1, 8)$:

$$\begin{aligned} m_1^1 = (1, \quad q_a, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = 8\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = 8\}) \end{aligned}$$

**Input:** $\breve{A}^2 = b(t1, 90, s5, 8, 10)$:

$$\begin{aligned} m_1^2 = (1, \quad q_{ab}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{10}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{10}\}) \\ m_2^2 = (2, \quad q_b, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = 10\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = 10\}) \end{aligned}$$

**Input:** $\breve{A}^3 = c(t1, 90, s5, 10, 15)$:

$$\begin{aligned} m_1^3 = (1, \quad q_{abc}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{15}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{15}\}) \\ m_2^3 = (2, \quad q_{bc}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{15}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{15}\}) \end{aligned}$$

**Output:** $\mathbf{MP}(\breve{A}^1\breve{A}^2\breve{A}^3) = \{p_1(t1, s5, 1, 15), p_2(t1, s5, 1, 15)\}$

**Input:** $\breve{A}^4 = d(t1, 90, s5, 15, 17)$:

$$\begin{aligned} m_1^4 = (1, \quad q_{abcd}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{17}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{17}\}) \\ m_2^4 = (2, \quad q_{bcd}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{17}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{17}\}) \end{aligned}$$

**Output:** $\mathbf{MP}(\breve{A}^1\breve{A}^2\breve{A}^3\breve{A}^4) = \{p_3(t1, s5, 1, 17), p_4(t1, s5, 8, 17)\}$

## 5.2 Soundness and completeness

Let $PA$ be a prefix acceptor, which corresponds to a linear chain Datalog program $\mathbf{P}_{\mathrm{I}}$. Let $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$ denote the *success set* of the marker passing method, i.e., the set of ground instances of the target predicates $p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$, which are calculated by the marker passing algorithm. In this section, we want to show, that the marker passing method is *sound* and *complete*, i.e., for a given EDB instance $\mathbf{P}_{\mathrm{E}}$, the success set $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$ is the minimum Herbrand model of the extended program $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$.

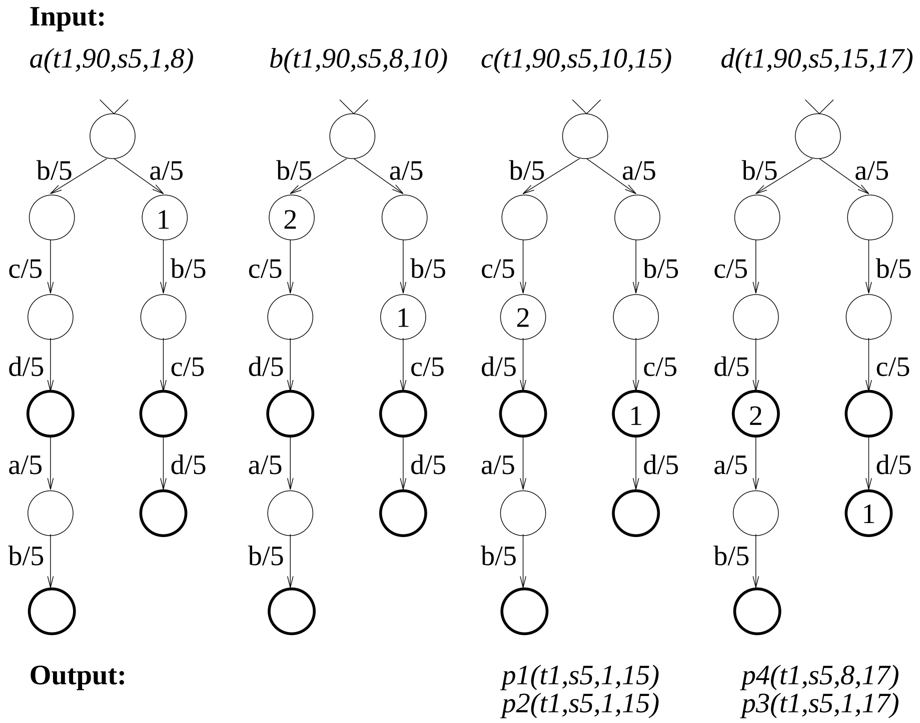


Figure 5: Example 1

In Lemma 2, we have shown, that for a given $\mathbf{P}_{\mathrm{I}}$ our restructuring methods generate a program $\mathbf{P}'_{\mathrm{I}}$, such that

$$\begin{array}{ccc} \{p_i(t_1,\ldots,t_s) & \mid & p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} \\ & = & \\ \{p_i(t_1,\ldots,t_s) & \mid & p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} \end{array}$$

where $\mathcal{I} \subseteq$IDB($\mathbf{P}_{\mathrm{I}}$). Each rule of the program $\mathbf{P}'_{\mathrm{I}}$ has one of the forms

$$q_a(Tr,O,S,X,Y) \leftarrow a(Tr,O,S,X,Y) \tag{28}$$

$$q_j(Tr,O,S,X,Y) \leftarrow q_i(Tr,O,S,X,X_1), a(Tr,O,S,X_1,Y) \tag{29}$$

$$p_r(Tr,S,X,Y) \leftarrow q_s(Tr,O,S,X,Y) \tag{30}$$

with $a \in$ EDB($\mathbf{P}_{\mathrm{I}}$), $q_i \in$ IDB($\mathbf{P}'_{\mathrm{I}}$)$-\mathcal{I}$, and $p_r \in \mathcal{I} \subseteq$ IDB($\mathbf{P}_{\mathrm{I}}$). This program can be directly mapped to a prefix acceptor $PA$, such that for each rule of the form (28), there is a transition $(q_0, a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_a) \in \Delta_{PA}$ emanating from the starting state $q_0$. For each rule of the form (29), there is a transition $(q_i, a(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_j) \in \Delta_{PA}$. For a rule of the form (30), there exists a final state $q_s$, such that $p_r(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}) \in \lambda_{PA}(q_s)$.

In principal, our marker passing method works incrementally, i.e., it receives sequentially the components of the ground chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1\breve{A}^2\ldots\breve{A}^k$. However, in order to proof the soundness and completeness of the method, we show it for each finite subsequence, which provides us with a finite Herbrand base.

We define the configuration $\mathcal{C}_{PA}$ of a prefix acceptor $PA$ at a given time point $t$, to be the set of markers associated with the states of the $PA$ at $t$, with $\mathcal{C}^0_{PA} = \emptyset$.

Given an EDB instance $\mathbf{P}_{\mathrm{E}}$, which is required to be a ground chain, the function $Succ_{\mathbf{P}_{\mathrm{E}}}$ maps a configuration $\mathcal{C}_{PA}$ at time point $t$ to the successor configuration at time point $t+1$. We have $\mathcal{C}^{t+1}_{PA} = Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^t_{PA})$. We define

$$
\begin{aligned}
Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^0_{PA}) = \{ \;& (t, q, \kappa_{\mathbf{BF},init}, \kappa_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa_{\mathbf{SF},update}) \mid t \in \{1,\ldots,k\} \text{ and} \\
& \text{for } \breve{A}^t \in \mathbf{P}_{\mathrm{E}} \text{ there exists } (q_0, A, q) \in \Delta_{PA}, \text{ such that} \\
& A \text{ is unifiable with } \breve{A}^t \text{ and} \\
& \kappa_{\mathbf{BF},init} = \{\mathtt{tr}_{\mathbf{BF}} = tr_{\mathbf{BF}}(\breve{A}^t), \mathtt{o}_{\mathbf{BF}} = o_{\mathbf{BF}}(\breve{A}^t), \mathtt{s}_{\mathbf{BF}} = s_{\mathbf{BF}}(\breve{A}^t)\} \\
& \kappa_{\mathbf{SF},init} = \{\mathtt{tr}_{\mathbf{SF}} = tr_{\mathbf{BF}}(\breve{A}^t), \mathtt{s}_{\mathbf{SF}} = s_{\mathbf{BF}}(\breve{A}^t), \mathtt{from}_{\mathbf{SF}} = from_{\mathbf{BF}}(\breve{A}^t)\} \\
& \kappa_{\mathbf{BF},update} = \{\mathtt{from}_{\mathbf{BF}} = to_{\mathbf{BF}}(\breve{A}^t)\} \\
& \kappa_{\mathbf{SF},update} = \{\mathtt{to}_{\mathbf{SF}} = to_{\mathbf{BF}}(\breve{A}^t)\} \\
\}&
\end{aligned}
$$

and for $i > 0$

$$
\begin{aligned}
Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^i_{PA}) = \{ \;& (t, q_k, \kappa_{\mathbf{BF},init}, \kappa^{i+1}_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^{i+1}_{\mathbf{SF},update}) \mid t \le k-i+1 \text{ and} \\
& (t, q_j, \kappa_{\mathbf{BF},init}, \kappa^{i}_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^{i}_{\mathbf{SF},update}) \in \mathcal{C}^i_{PA} \text{ and there} \\
& \text{exists } (q_j, A, q_k) \in \Delta_{PA} \text{ such that } A\kappa_{\mathbf{BF}} \text{ is unifiable with} \\
& \breve{A}^{t+i-1} \in \mathbf{P}_{\mathrm{E}} \text{ and } \kappa_{\mathbf{BF}} = \kappa_{\mathbf{BF},init} \cup \kappa^i_{\mathbf{BF},update} \\
& \kappa^{i+1}_{\mathbf{BF},update} = \{\mathtt{from}_{\mathbf{BF}} = to_{\mathbf{BF}}(\breve{A}^{t+i-1})\} \\
& \kappa^{i+1}_{\mathbf{SF},update} = \{\mathtt{to}_{\mathbf{SF}} = to_{\mathbf{BF}}(\breve{A}^{t+i-1})\} \\
\}&
\end{aligned}
$$

Each member $\breve{A}^t, t \in \{1,\ldots,k\}$ of the ground chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1 \ldots \breve{A}^k$ is the beginning of a subsequence from which an atom over $p_i \in \mathcal{I}$ may be derivable. $Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^0_{PA})$ extracts, if possible, for each $\breve{A}^t$ the sensor and basic feature constraints (initial and update), and passes a marker to a direct successor of the starting state $q_0$. Thus, each marker $t$ represents the current processing status of the sequence beginning with $\breve{A}^t$. For every configuration $Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^i_{PA})$, the successor function tries to pass forward marker $t$ by considering the $(t+i-1)$-th element of $\mathbf{P}_{\mathrm{E}}$.

We define the mapping $\Gamma$, which is applied to a configuration of a prefix acceptor, and which generates the ground facts over the target predicates $p_r \in \mathcal{I} \subseteq$IDB$(\mathbf{P}_{\mathrm{I}})$, which are associated with the final states of the acceptor, and which in the current configuration are occupied by a marker.

$$
\begin{aligned}
\Gamma(\mathcal{C}_{PA}) = \{\breve{B} \mid \;& (t, q, \_, \_, \kappa_{\mathbf{SF},init}, \kappa_{\mathbf{SF},update}) \in \mathcal{C}_{PA} \text{ and } \breve{B} = B_r\kappa_{\mathbf{SF}} \text{ with} \\
& B_r \in \lambda(q) \text{ and } \kappa_{\mathbf{SF}} = \kappa_{\mathbf{SF},init} \cup \kappa_{\mathbf{SF},update}\}.
\end{aligned}
$$

Note, that the application of rules of the form (28) and (29) is simulated by the $Succ_{\mathbf{P}_{\mathrm{E}}}$ mapping. The application of rules of the form (30) is simulated by the $\Gamma$ mapping. We consider the sequence

$$
\begin{aligned}
\Gamma^1(\mathcal{C}^1_{PA}) &= \Gamma^1(Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^0_{PA})) \\
\Gamma^2(\mathcal{C}^2_{PA}) &= \Gamma^2(Succ_{\mathbf{P}_{\mathrm{E}}}(Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^0_{PA}))) \\
&\dots \\
\Gamma^i(\mathcal{C}^i_{PA}) &= \Gamma^i(\underbrace{Succ_{\mathbf{P}_{\mathrm{E}}}(\dots Succ_{\mathbf{P}_{\mathrm{E}}}}_{i\ times}(\mathcal{C}^0_{PA})\dots))
\end{aligned}
$$

As the length of the input chain $\mathbf{P}_{\mathrm{E}}$ is $k$, the $\Gamma$ ( and $Succ_{\mathbf{P}_{\mathrm{E}}}$) mapping can be applied exactly $k$ times. We define the success set to be

$$
\begin{aligned}
\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}}) &= \Gamma^1(\mathcal{C}^1_{PA}) \cup \Gamma^2(\mathcal{C}^2_{PA}) \cup \dots \cup \Gamma^k(\mathcal{C}^k_{PA}) \\
&\quad \Gamma^1(Succ_{\mathbf{P}_{\mathrm{E}}}(\mathcal{C}^0_{PA})) \cup \dots \cup \Gamma^k(\underbrace{Succ_{\mathbf{P}_{\mathrm{E}}}(\dots Succ_{\mathbf{P}_{\mathrm{E}}}}_{k\ times}(\mathcal{C}^0_{PA})\dots))
\end{aligned}
$$

Given a ground chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1\breve{A}^2\dots\breve{A}^k$, assume, that there exists a subsequence of length $i$, $\breve{A}^t\dots\breve{A}^{t+i-1}, 1 \le t \le k, t+i-1 \le k$, from which $\breve{B} = p_i(tr,s,x,y), p_i \in \mathcal{I}$ can be derived. Then, there exists a unique sequence of rules $C_1, C_2, \dots, C_i, C_{i+1} \in \mathbf{P}'_{\mathrm{I}}$

$C_1 = (Q_1 \leftarrow A_1)$, such that $C_1\sigma_1 = (\breve{Q}_1 \leftarrow \breve{A}_1)$ with $\breve{A}_1 = \breve{A}^t$ and $\breve{Q}_1 \in \mathrm{T}^1_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$

$(C_2 = Q_2 \leftarrow Q_1A_2)$, such that $C_2\sigma_2 = (\breve{Q}_2 \leftarrow \breve{Q}_1\breve{A}_2)$ with $\breve{A}_2 = \breve{A}^{t+1}$ and $\breve{Q}_2 \in \mathrm{T}^2_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$

$C_3 = (Q_3 \leftarrow Q_2A_3)$, such that $C_3\sigma_3 = (\breve{Q}_3 \leftarrow \breve{Q}_2\breve{A}_3)$ with $\breve{A}_3 = \breve{A}^{t+2}$ and $\breve{Q}_3 \in \mathrm{T}^3_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$

$\dots$

$C_i = (Q_i \leftarrow Q_{i-1}A_i)$, such that $C_i\sigma_i = (\breve{Q}_i \leftarrow \breve{Q}_{i-1}\breve{A}_i)$ with $\breve{A}_i = \breve{A}^{t+i-1}$ and $\breve{Q}_i \in \mathrm{T}^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$

$C_{i+1} = (B \leftarrow Q_i)$, such that $C_{i+1}\sigma_{i+1} = (\breve{B} \leftarrow \breve{Q}_i)$, and $\breve{B} \in \mathrm{T}^{i+1}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$

where $Q_j, j \in \{1,\dots,i\}$ are atoms over the predicates $q_j \in$IDB$(\mathbf{P}'_{\mathrm{I}})-\mathcal{I}$. Let $PA$ be the prefix acceptor, which corresponds to $\mathbf{P}'_{\mathrm{I}}$. The sequence of transitions, which corresponds to the rules $C_1, C_2, \dots, C_i, C_{i+1} \in \mathbf{P}'_{\mathrm{I}}$ is

$(q_0, A_1, q_1)$ such that $A_1$ is unifiable with $\breve{A}^t$ and $m \in \mathcal{C}^1_{PA}$ with

$$m = \{t, q_1, \kappa_{\mathbf{BF},init}, \kappa^1_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^1_{\mathbf{SF},update}\}$$

$(q_1, A_2, q_2)$ such that $A_2\kappa^1_{\mathbf{BF}}$ is unifiable with $\breve{A}^{t+1}$ and $m \in \mathcal{C}^2_{PA}$ with

$$m = \{t, q_2, \kappa_{\mathbf{BF},init}, \kappa^2_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^2_{\mathbf{SF},update}\}$$

$(q_2, A_3, q_3)$ such that $A_3\kappa^2_{\mathbf{BF}}$ is unifiable with $\breve{A}^{t+2}$ and $m \in \mathcal{C}^3_{PA}$ with

$$m = \{t, q_3, \kappa_{\mathbf{BF},init}, \kappa^3_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^3_{\mathbf{SF},update}\}$$

$$\ldots$$

$(q_{i-1}, A_i, q_i)$ such that $A_i\kappa^{i-1}_{\mathbf{BF}}$ is unifiable with $\breve{A}^{t+i-1}$ and $m \in \mathcal{C}^i_{PA}$ with

$$m = \{t, q_i, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update}\}$$

$B \in \lambda(q_i)$ such that $\breve{B} = B\kappa^i_{\mathbf{SF}}$ and $\breve{B} \in \Gamma^i(\mathcal{C}^i_{PA})$.

with $\kappa_{\mathbf{BF},init} = \texttt{init\_bf\_constraints}(\breve{A}^t)$ and $\kappa_{\mathbf{SF},init} = \texttt{init\_sf\_constraints}(\breve{A}^t)$. For $j \in \{1, \ldots, i\}$, we have $\kappa^j_{\mathbf{BF},update} = \texttt{update\_bf\_constraints}(\breve{A}^{t+j-1})$, $\kappa^j_{\mathbf{SF},update} = \texttt{update\_sf\_constraints}(\breve{A}^{t+j-1})$, $\kappa^j_{\mathbf{BF}} = \kappa_{\mathbf{BF},init} \cup \kappa^j_{\mathbf{BF},update}$, and $\kappa^j_{\mathbf{SF}} = \kappa_{\mathbf{SF},init} \cup \kappa^j_{\mathbf{SF},update}$.

We establish this correspondence with the following lemma

**Lemma 3** *Let $\mathbf{P}'_{\mathrm{I}}$ be a linear chain Datalog program with rules, which have at most one EDB subgoal. Let $PA$ be the prefix acceptor which corresponds to $\mathbf{P}'_{\mathrm{I}}$. Let $\mathcal{I} = \{p_1, \ldots, p_n\} \subseteq IDB(\mathbf{P}'_{\mathrm{I}})$. Consider the set of predicate symbols $\{q_s | q_s \in IDB(\mathbf{P}'_{\mathrm{I}}) - \mathcal{I}\}$. This set corresponds to the set of predicate symbols for the predicates, which have been introduced when the original program $\mathbf{P}_{\mathrm{I}}$ was restructured, such that each rule of $\mathbf{P}'_{\mathrm{I}}$ has the form (28), (29), or (30). Let $\mathbf{P}_{\mathrm{E}} = \breve{A}^1 \ldots, \breve{A}^k$ be a ground chain of length $k$, which is input to $\mathbf{P}'_{\mathrm{I}}$ and $PA$, respectively. If*

$$q_r(x_1, x_2, x_3, x_4, x_5) \in \{\breve{Q} | \breve{Q} \textit{ is a ground atom over a predicate } q \in IDB(\mathbf{P}'_{\mathrm{I}}) - \mathcal{I} \textit{ and } \breve{Q} \in T^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}, 1 \leq i \leq \omega$$

*then configuration $\mathcal{C}^i_{PA}$ of the prefix acceptor, contains a marker $m$, which is associated with state $q_r$, i.e.,*

$$m = (t, q_r, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update})$$

*such that*

$$\begin{aligned}
\kappa_{\mathbf{BF},init} &= \{\texttt{tr}_{\mathbf{BF}} = x_1, \texttt{o}_{\mathbf{BF}} = x_2, \texttt{s}_{\mathbf{BF}} = x_3\} \\
\kappa_{\mathbf{SF},init} &= \{\texttt{tr}_{\mathbf{SF}} = x_1, \texttt{s}_{\mathbf{SF}} = x_3, \texttt{from}_{\mathbf{SF}} = x_4\} \\
\kappa^i_{\mathbf{BF},update} &= \{\texttt{from}_{\mathbf{BF}} = x_5\} \\
\kappa^i_{\mathbf{SF},update} &= \{\texttt{to}_{\mathbf{SF}} = x_5\}
\end{aligned}$$

*The converse statement also holds, i.e., if there is a configuration $\mathcal{C}^i_{PA}$ with $m \in \mathcal{C}^i_{PA}$ and $m = (t, q_r, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update})$, where the constraints are the same as specified above, then $q_r(x_1, x_2, x_3, x_4, x_5) \in T^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}), 1 \leq i \leq \omega$.*

**Proof** We show the first part by induction on the number $i$ of applications of the $\mathrm{T}_{\mathbf{P}'_{\mathrm{I}}}$ mapping.

Induction basis for $i = 1$: Let $\breve{Q}_1 = q_r(x_1, x_2, x_3, x_4, x_5) \in \mathrm{T}^1_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. Then, there exists a clause $C_1 \in \mathbf{P}'_{\mathrm{I}}$ with $C_1\sigma_1 = \breve{Q}_1 \to \breve{A}$, such that $\breve{A} = \breve{A}^t \in \mathbf{P}_{\mathrm{E}}, 1 \leq t \leq k$. For rule $C_1$, we have the transition $(q_0, A, q_1) \in \Delta_{PA}$. As $\breve{A} = a(x_1, x_2, x_3, x_4, x_5) = \breve{A}^t \in \mathbf{P}_{\mathrm{E}}, a \in \mathrm{EDB}(\mathbf{P}_{\mathrm{I}})$, the application of the $Succ_{\mathbf{P}_{\mathrm{E}}}$ mapping to $\mathcal{C}^0_{PA}$ will produce the marker $m \in \mathcal{C}^1_{PA}$ with $m = (t, q_1, \kappa_{\mathbf{BF},init}, \kappa^1_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^1_{\mathbf{SF},update})$ with

$$\begin{aligned}
\kappa_{\mathbf{BF},init} = \quad & \texttt{init_bf_constraints}(\breve{A}^t) & = \{\texttt{tr}_{\mathbf{BF}} = x_1, \texttt{o}_{\mathbf{BF}} = x_2, \texttt{s}_{\mathbf{BF}} = x_3\} \\
\kappa_{\mathbf{SF},init} = \quad & \texttt{init_sf_constraints}(\breve{A}^t) & = \{\texttt{tr}_{\mathbf{SF}} = x_1, \texttt{s}_{\mathbf{SF}} = x_3, \texttt{from}_{\mathbf{SF}} = x_4\} \\
\kappa^1_{\mathbf{BF},update} = \quad & \texttt{update_bf_constraints}(\breve{A}^t) & = \{\texttt{from}_{\mathbf{BF}} = x_5\} \\
\kappa^1_{\mathbf{SF},update} = \quad & \texttt{update_sf_constraints}(\breve{A}^t) & = \{\texttt{to}_{\mathbf{SF}} = x_5\}.
\end{aligned}$$

Suppose, as the induction assumption, that for each $\breve{Q} = q_r(x_1, x_2, x_3, x_4, x_5) \in \mathrm{T}^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$, there exists a marker $(t, q_r, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update}) \in \mathcal{C}^i_{PA}, t \in \{1, \ldots, k\}$. Then, we have to show the induction hypothesis, that for each $\breve{Q} \in \mathrm{T}^{i+1}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$, there exists a marker $m \in \mathcal{C}^{i+1}_{PA}$. Let $\breve{Q}_{i+1} = q_r(x_1, x_2, x_3, x_4, x_5) \in \mathrm{T}^{i+1}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. Then, there exists a rule $C_{i+1} \in \mathbf{P}'_{\mathrm{I}}$, such that $C_{i+1}\sigma = (\breve{Q}_{i+1} \leftarrow \breve{Q}_i\breve{A}_{i+1})$ with $\breve{Q}_i \in \mathrm{T}^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$ and $\breve{A}_{i+1} = \breve{A}^{t+i} \in \mathbf{P}_{\mathrm{E}}$. For rule $C_{i+1}$, we have the transition $(q_i, A_{i+1}, q_{i+1}) \in \Delta_{PA}$. According to the induction assumption, $\mathcal{C}^i_{PA}$ contains a marker $(t, q_i, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update})$, such that

$$\begin{aligned}
\kappa_{\mathbf{BF},init} &= \{\texttt{tr}_{\mathbf{BF}} = tr_{\mathbf{BF}}(\breve{Q}_i), \texttt{o}_{\mathbf{BF}} = o_{\mathbf{BF}}(\breve{Q}_i), \texttt{s}_{\mathbf{BF}} = s_{\mathbf{BF}}(\breve{Q}_i)\} \\
\kappa_{\mathbf{SF},init} &= \{\texttt{tr}_{\mathbf{SF}} = tr_{\mathbf{BF}}(\breve{Q}_i), \texttt{s}_{\mathbf{SF}} = s_{\mathbf{BF}}(\breve{Q}_i), \texttt{from}_{\mathbf{SF}} = from_{\mathbf{BF}}(\breve{Q}_i)\} \\
\kappa^i_{\mathbf{BF},update} &= \{\texttt{from}_{\mathbf{BF}} = to_{\mathbf{BF}}(\breve{Q}_i)\} \\
\kappa^i_{\mathbf{SF},update} &= \{\texttt{to}_{\mathbf{SF}} = to_{\mathbf{BF}}(\breve{Q}_i)\}.
\end{aligned}$$

As $A_{i+1}\kappa^i_{\mathbf{SF}}$ is unifiable with $\breve{A}_{i+1} = \breve{A}^{t+i} \in \mathbf{P}_{\mathrm{E}}$, the application of the $Succ_{\mathbf{P}_{\mathrm{E}}}$ mapping results in a marker $(t, q_{i+1}, \kappa_{\mathbf{BF},init}, \kappa^{i+1}_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^{i+1}_{\mathbf{SF},update}) \in \mathcal{C}^{i+1}_{PA}$, with

$$\begin{aligned}
\kappa^{i+1}_{\mathbf{BF},update} &= \{\texttt{from}_{\mathbf{BF}} = to_{\mathbf{BF}}(\breve{A}_{i+1})\} \\
\kappa^{i+1}_{\mathbf{SF},update} &= \{\texttt{to}_{\mathbf{SF}} = to_{\mathbf{BF}}(\breve{A}_{i+1})\}.\Box
\end{aligned}$$

The proof of the converse statement follows the same line of arguments.

**Theorem 1** *(Soundness) Given an EDB instance $\mathbf{P}_{\mathrm{E}}$ and a prefix acceptor $PA$, which corresponds to a basic chain Datalog program $\mathbf{P}'_{\mathrm{I}}$, its success set $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$ is contained in the minimum Herbrand model of $\mathbf{P} = \mathbf{P}'_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$, i.e.,*

$$\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}}) \subseteq \{p_i(t_1, \ldots, t_s) \mid p_i \in \mathcal{I} \textit{ and } p_i(t_1, \ldots, t_s) \in T^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}$$

**Proof** We have $\mathbf{P}_{\mathrm{E}} = \breve{A}^1\breve{A}^2\ldots\breve{A}^k$ and $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}}) = \Gamma^1(\mathcal{C}^1_{PA}) \cup \Gamma^2(\mathcal{C}^2_{PA}) \cup \ldots \cup \Gamma^k(\mathcal{C}^k_{PA})$. Let $\breve{B} \in \Gamma^i(\mathcal{C}^i_{PA}), 1 \leq i \leq k$. Then $m = (t, q_r, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update}) \in \mathcal{C}^i_{PA}$ with $B \in \lambda(q_r)$ and $\breve{B} = B\kappa^i_{\mathbf{SF}}, \kappa^i_{\mathbf{SF}} = \kappa_{\mathbf{SF},init} \cup \kappa^i_{\mathbf{SF},update}$. Let

$$\begin{aligned}
\kappa_{\mathbf{BF},init} &= \{\mathtt{tr}_{\mathbf{BF}} = x_1, \mathtt{o}_{\mathbf{BF}} = x_2, \mathtt{s}_{\mathbf{BF}} = x_3\} \\
\kappa_{\mathbf{SF},init} &= \{\mathtt{tr}_{\mathbf{SF}} = x_1, \mathtt{s}_{\mathbf{SF}} = x_3, \mathtt{from}_{\mathbf{SF}} = x_4\} \\
\kappa^i_{\mathbf{BF},update} &= \{\mathtt{from}_{\mathbf{BF}} = x_5\} \\
\kappa^i_{\mathbf{SF},update} &= \{\mathtt{to}_{\mathbf{SF}} = x_5\}.
\end{aligned}$$

According to Lemma 3, $\breve{Q} = q_r(X_1, x_2, x_3, x_4, x_5) \in \mathrm{T}^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. There exists a rule $C \in \mathbf{P}'_{\mathrm{I}}$, such that $C\sigma = (\breve{B} \leftarrow \breve{Q})$. From this follows, that $\breve{B} \in \mathrm{T}^{i+1}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. $\square$

Given an EDB instance $\mathbf{P}_{\mathrm{E}}$, a prefix acceptor $PA$, which corresponds to a basic chain Datalog program $\mathbf{P}'_{\mathrm{I}}$, and a set of IDB predicates $\mathcal{I} \subseteq$IDB($\mathbf{P}_{\mathrm{I}}$), we say that the marker passing method is *complete with respect to* $\mathcal{I}$, if every ground atom $\breve{B}$ over a $p_i \in \mathcal{I}$, which is in the minimum Herbrand model of $\mathbf{P} = \mathbf{P}'_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$, is also in the success set $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$.

**Theorem 2** *(Completeness)*

$$\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}}) \supseteq \{p_i(t_1, \ldots, t_s) \mid p_i \in \mathcal{I} \textit{ and } p_i(t_1, \ldots, t_s) \in T^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}$$

**Proof** Let $\breve{B} \in \{p_i(t_1, \ldots, t_s) \mid p_i \in \mathcal{I}$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{i+1}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}, 1 < i \leq \omega - 1$. Then there exists a rule $C_{i+1} \in \mathbf{P}'_{\mathrm{I}}$, such that $C_{i+1}\sigma = (\breve{B} \leftarrow \breve{Q}_i)$ with $\breve{Q}_i \in \mathrm{T}^i_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. According to Lemma 3 there exists $m \in \mathcal{C}^i_{PA}$, such that $m = (t, q_i, \kappa_{\mathbf{BF},init}, \kappa^i_{\mathbf{BF},update}, \kappa_{\mathbf{SF},init}, \kappa^i_{\mathbf{SF},update}), t \leq k-i+1$. We also have $B \in \lambda(q_i)$. The application of the $\Gamma$ mapping to $\mathcal{C}^i_{PA}$ yields $\breve{B} = B\kappa^i_{\mathbf{SF}}$ with $\kappa^i_{\mathbf{SF}} = \kappa_{\mathbf{SF},init} \cup \kappa^i_{\mathbf{SF},update}$. Therefore, $\breve{B} \in \Gamma^i(\mathcal{C}^i_{PA}) \subseteq \mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$. $\square$

# 6 Post-Processing Chain Datalog Programs

Given the rules for the robotics domain, we have mapped them to a prefix acceptor to which we have applied the efficient marker passing method, in order to derive higher level concepts from sensor observations. The analysis of performance tests motivated the post-processing phase during which the acceptor and the chain Datalog program, respectively, are modified. In contrast to restructuring, which does not change the coverage for the target concepts, post-processing increases it. In 6.1, we motivate the post-processing phase from the point of view of the application. In 6.2, we present the post-processing method. The post-processed acceptor can be mapped back to a linear chain Datalog program. In 6.3 we proof that its coverage for the target predicates is really increased. The experimental results in 6.4 show the improvements, which were gained by the post-processing step.

### 6.1 Disadvantages of the rules learned for the robotics domain

The set of basic feature predicates, **BF**, contains some predicates, which do not contribute perceptual information: In general, a basic feature describes a time interval during which the robot keeps moving without changing its direction, and during which the tendency of change of successive measurements is approximately the same. The basic features are calculated in such a way, that they cover a time interval, $[Start, End]$, of a given trace completely, i.e., given a basic feature $a(Tr, O, S, T1, T2)$, $Start \leq T1$, $T2 < End$, there will be a basic feature $b(Tr, O, S, T2, T3)$, $a, b \in PS_{\mathbf{BF}}$. In order to guarantee this, basic feature predicates had to be introduced to account for the situations, in which the first assumption is not satisfied, e.g., for the situation in which the robot does not move (`no_movement/5`).

Now, two types of situations can occur during the training and testing/performance phase, respectively: During the training phase a prefix (tree) acceptor is inferred, whose transitions are labeled with an "irrelevant" perception such as `no_movement/5`, because it occurred sufficiently often in the training data, that the robot stood still. This situation is exemplified with the automaton in the upper part of Figure 6. Given this automaton, whenever state $q_i$ is reached during the performance phase, the robot *expects* the totally irrelevant perception `no_movement/5`, instead of ignoring it. On the other hand, assume,

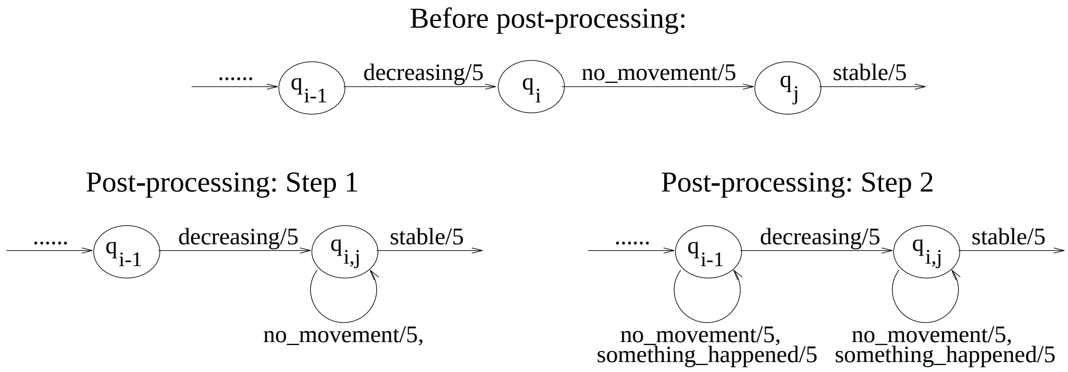


Figure 6: Post-processing of irrelevant basic features

that during the performance phase the robot has perceived an observation sequence which leads to state $q_{i-1}$ and then has to stop for some reason. This causes the generation of a `no_movement` predicate instance, which prevents the marker passing method from continuing to process the subsequence which may lead to state $q_i$. Obviously, an *unexpected* irrelevant basic feature, which does not contribute any perceptual information, should be ignored during the performance phase. The same arguments apply to the basic feature predicate `something_happened`. It indicates some outlayer, which cannot be classified as `incr_peak`, `decr_peak`, or `single_peak`.

In order to account for irrelevant basic features, we can modify the automaton in two steps. Firstly, we remove the non-cyclic transitions, which are labeled by irrelevant basic features. Assume, that there is a transition $(q_i, \texttt{no\_movement}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_j)$ (see Figure 6). Then, we remove this transition, merge the states $q_i$ and $q_j$, and add the cyclic transition $(q_{i,j}, \texttt{no\_movement}(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{i,j})$ (see Figure 6, bottom left). In the second step, we add cyclic transitions to each of the states of the acceptor (see Figure 6, bottom right).

The ultimate goal of post-processing a prefix acceptor is to generalize the rules, compiled in it, in such a way, that their predictive power is increased. If we perform the first post-processing step, we get the positive side effect, that the complexity of the prefix acceptor is reduced in terms of the number of states, the number of transitions and the maximal depth of the prefix acceptor. We define the maximal depth of the prefix acceptor to be the number of transitions on the longest path from the starting state to a final state, which does not contain any cycles.

## 6.2 Post-processing the prefix acceptor: Step 1

We explain the method informally with the prefix acceptor in Figure 4 in Section 4, i.e., we continue with our example programs, $\mathbf{P}_{\mathrm{I}}$ and $\mathbf{P}'_{\mathrm{I}}$, and the associated grammars $G$ and $G'$. Assume that $b \in PS_{\mathbf{BF}}$ denotes an irrelevant basic feature. Then, we have to delete the transition $(q_0, b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_b)$, to merge the states $q_0$ and $q_b$ of $PA$, yielding state $q_{b^*}$, and to add the cyclic transition $(q_{b^*}, b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{b^*})$. We also have to delete the transition $(q_a, b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{ab})$, to merge states $q_a$ and $q_{ab}$, yielding state $q_{b^*ab^*}$, and to add the transition $(q_{b^*ab^*}, b(Tr_{\underline{tr}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{b^*ab^*})$. Finally, we have to delete $(q_{bcda}, b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{bcdab})$, to merge the states $q_{bcda}$ and $q_{bcdab}$, yielding state $q_{b^*cdab^*}$ with the cyclic transition $(q_{b^*cdab^*}, b(Tr_{\underline{tr}}, O_{\underline{o}}, S_{\underline{s}}, X_{\underline{from}}, Y_{\underline{to}}), q_{b^*cdab^*})$. Thus, we get the prefix acceptor $PA'$ in Figure 7, where the original states $q_{bc}$,

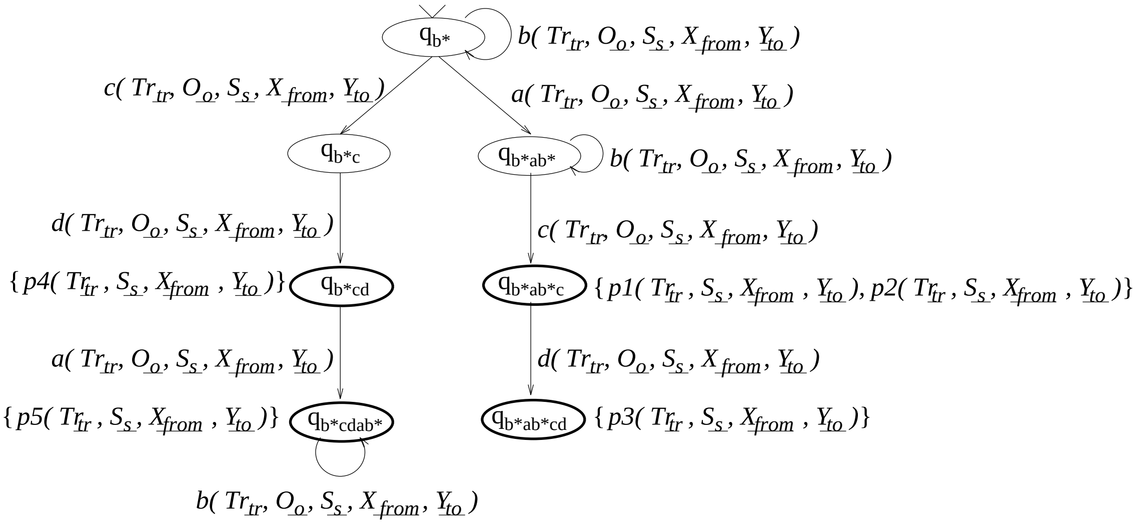


Figure 7: Prefix acceptor $PA'$

$q_{bcd}$, $q_{abc}$, and $q_{abcd}$ have been renamed by $q_{b^*c}$, $q_{b^*cd}$, $q_{b^*ab^*c}$, and $q_{b^*ab^*cd}$, respectively. In terms of the language, generated by the equivalent grammars, $G$ and $G'$, which are associated with $\mathbf{P}_{\mathrm{I}}$ and $\mathbf{P}'_{\mathrm{I}}$, post-processing amounts to generalizing the language $\hat{L}(G) = \hat{L}(G') = \{abc, abcd, bcd, bcdab\}$ to $\hat{L}(G'') = \{b^*ab^*c, b^*ab^*cd, b^*cd, b^*cdab^*\} \supset \hat{L}(G)$. Again, the states of $PA'$ represent the prefixes in $Prefix(\hat{L}(G''))$ (when we treat $b^*$ as one symbol). In other words, post-processing amounts to generalizing the rules, which are structured in the prefix acceptor. The program $\mathbf{P}_{\mathrm{I}}''$, which corresponds to $PA'$, contains newly introduced recursive rules. Our claim is, that this causes the predictive power of the rules to be increased. Given an EDB instance $\mathbf{P}_{\mathrm{E}}$, the coverage for the target predicates in $\mathcal{I}$ is increased, i.e., $Cov_{\mathbf{P}'}(\mathcal{I}) \subseteq Cov_{\mathbf{P}''}(\mathcal{I})$, where $\mathbf{P}' = \mathbf{P}_{\mathrm{I}}' \cup \mathbf{P}_{\mathrm{E}}$ and $\mathbf{P}'' = \mathbf{P}_{\mathrm{I}}'' \cup \mathbf{P}_{\mathrm{E}}$.

Furthermore, the complexity of the prefix acceptor, in terms of the number of states, the number of transitions, and the maximal depth is reduced. Note, that merging any state with a final state (e.g., $q_{bcda}$ and $q_{bcdab}$) yields a final state, whereas merging non-final states (e.g., $q_{ab}$ and $q_{abc}$) yields a non-final state.

---

`post_proc`$(PA^{old}, x, PA^{new})$
**begin**
$PA := PA^{old}$;
**while** there exists $q_i \in Q_{PA}$, such that $(q_i, x, q_j) \in \Delta_{PA}$ with $q_i \neq q_j$

1. $\Delta_{PA} := \Delta_{PA} - \{(q_i, x, q_j)\}$;
2. `merge_states`$(PA, q_i, q_j, PA^1)$ ;
3. $\Delta^1_{PA} := \Delta^1_{PA} \cup \{(q_{i,j}, x, q_{i,j})\}$ ;
4. $PA := PA^1$;

$PA^{new} := PA^1$;
**end**

Algorithm 8: `post_proc`

---

The procedure `post_proc` (Algorithm 8) takes as input the prefix acceptor $PA^{old} = (Q^{old}, \Sigma, Z, \Delta^{old}, q_0^{old}, F^{old}, \lambda^{old})$ and a label $x \in \Sigma$. The procedure generates as output the new prefix acceptor $PA^{new} = (Q^{new}, \Sigma, Z, \Delta^{new}, q_0^{new}, F^{new}, \lambda^{new})$, in which there do not exist any non-cyclic transitions, labeled $x$. The procedure does for each state $q_i$, for which there exists a transition $(q_i, x, q_j)$ with $q_i \neq q_j$ the following: The transition $(q_i, x, q_j)$ is deleted (Step 1), the states $q_i$ and $q_j$ are merged, yielding state $q_{i,j}$ (Step 2, which requires to recursively merge other states and transitions), and the cyclic transition $(q_{i,j}, x, q_{i,j})$ is added (Step 3).

The procedure `merge_states` (Algorithm 9) takes as input the prefix acceptor $PA^{old} = (Q^{old}, \Sigma, Z, \Delta^{old}, q_0^{old}, F^{old}, \lambda^{old})$ and two states, $q_i$ and $q_j$. It generates as output the new prefix acceptor $PA^{new}$. The procedure works as follows: State $q_j$ is removed from the set of states (Step 2), whereas state $q_i$ is replaced by the merged state $q_{i.j}$ (Step 3). Now, we consider all transitions, which start from the original states, $q_i$ and $q_j$, respectively. For each transition, which starts from $q_i$ and which is labeled by some $x \in \Sigma$, we check, whether there is a transition, starting from $q_j$, labeled $x$. If that is not the case, the transition $(q_i, x, q_j)$ is replaced by $(q_{i,j}, x, q_j)$ (Step 4). The corresponding step is performed for transitions, which start from state $q_j$ (Step 5). If there exist transitions, labeled by some $z \in \Sigma$, starting from both states, $q_i$ and $q_j$, they have to be merged (Step 6). If one of the states, $q_i$ or $q_j$, is a final state, then the merged state $q_{i,j}$ becomes a final state and it is associated with the final tags of both original states (Step 7). If the original state $q_i$ was the starting state, the merged state $q_{i,j}$ becomes the starting state of the new prefix acceptor $PA^{new}$ (Step 8).

The procedure `merge_transitions` (Algorithm 10) takes as input a prefix acceptor $PA^{old}$, and the two transitions, which have to be merged. It produces as output the new

---

`merge_states`$(PA^{old}, q_i, q_j, PA^{new})$
**begin**

1. $PA := PA^{old}$; $F_{PA} = \emptyset$;
2. $Q_{PA} := Q_{PA} - \{q_i\}$
3. $Q_{PA} := (Q_{PA} - \{q_i\}) \cup \{q_{i,j}\}$
4. for each $x \in \Sigma$, such that $(q_i, x, q_r) \in \Delta_{PA}$, but $(q_j, x, \_) \notin \Delta_{PA}$:
   $\Delta_{PA} := (\Delta_{PA} - \{(q_i, x, q_r)\}) \cup \{(q_{i,j}, x, q_r)\}$
5. for each $y \in \Sigma$, such that $(q_j, y, q_s) \in \Delta_{PA}$, but $(q_i, y, \_) \notin \Delta_{PA}$:
   $\Delta_{PA} := (\Delta_{PA} - \{(q_j, y, q_s)\}) \cup \{(q_{i,j}, y, q_s)\}$
6. for each $z \in \Sigma$, such that $(q_i, z, q_k) \in \Delta_{PA}$ and $(q_j, z, q_l) \in \Delta_{PA}$:
   `merge_transitions`$(PA, (q_i, z, q_k), (q_j, z, q_l), PA^{new})$;
7. **if** $q_i \in F_{PA^{old}}$ or $q_j \in F_{PA^{old}}$, **then**
   **begin**
   $F_{PA^{new}} := F_{PA^{new}} \cup \{q_{i,j}\}$; $\lambda_{PA^{new}}(q_{i,j}) := \lambda_{PA^{old}}(q_i) \cup \lambda_{PA^{old}}(q_j)$;
   **end**
8. **if** $q_{0,old} = q_i$ **then** $q_{0,new} := q_{i,j}$;

**end**

Algorithm 9: `merge_states`

---

prefix acceptor $PA^{new}$. It works as follows: The two transitions, $(q_i, z, q_k)$ and $(q_j, z, q_l)$, are deleted (Step 2). Then, the two states, $q_k$ and $q_j$, have to be merged (Step 3). Finally, the new transition $(q_{i,j}, z, q_{k,l})$ is added.

---

`merge_transitions`$(PA^{old}, (q_i, z, q_k), (q_j, z, q_l), PA^{new})$
**begin**

1. $PA := PA^{old}$;
2. $\Delta_{PA} := \Delta_{PA} - \{(q_i, z, q_k), (q_j, z, q_j)\}$
3. `merge_states`$(PA, q_k, q_l, PA^{new})$
4. $\Delta_{PA^{new}} := \Delta_{PA^{new}} \cup \{(q_{i,j}, z, q_{k,l})\}$

**end**

Algorithm 10: `merge_transitions`

---

**Example** The effect of reducing the complexity of the graph by merging states and transitions, respectively, is illustrated with the acceptor $PA_1$ in Figure 8, which after post-processing for $b$, is transformed to the acceptor $PA_2$ in Figure 9. $PA_1$ accepts the language generated by the grammar $G_1 = (V_1, \Sigma_1, P_1, s)$ with $PA_1 = \{s \rightarrow p_1|p_2|p_3|p_4, p_1 \rightarrow abc, p_2 \rightarrow aca, p_3 \rightarrow abca, p_4 \rightarrow abcd\}$ for the language $\hat{L}(G_1) = \{abc, aca, abca, abcd\}$. Again let $b$ denote an irrelevant basic feature. Post-processing is to generalize $\hat{L}(G_1)$ to the language $\hat{L}(G_2) = \{ab^*c, ab^*ca, ab^*cd\}$. Applying the algorithm `post_proc` to $P_1$ causes the states $q_1$ and $q_2$ to be merged. Given that, the transitions, which lead from $q_2$ to $p_1$, and from $q_1$ to $q_3$, respectively, and which are both labeled by $c$, have to be merged.

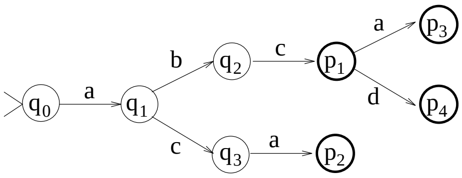


Figure 8: $PA_1$

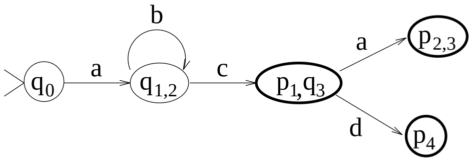


Figure 9: $PA_2$

This, in turn, requires to merge states $p_1$ and $q_3$. The recursive call to merge transitions, then causes the edges from $p_1$ to $p_3$, and from $q_3$ to $p_2$ to be merged. The recursion ends, when the states $p_3$ and $p_2$ have been merged. So, in this example, post-processing reduces the number of states from 8 to 5, the number of edges from 7 to 5, and the max. depth of a non-cyclic path from 4 to 3.

### 6.3 The post-processed chain Datalog program

The grammar $G''$, which is associated with acceptor $PA'$ (see Figure 7) is $G'' = (V'', \Sigma'', P'', s)$, with $V'' = \{s, p_1, p_2, p_3, p_4, p_5, q_{b^*}, q_{b^*ab^*}, q_{b^*c}, q_{b^*cd}, q_{b^*ab^*c}, q_{b^*cdab^*}, q_{b^*ab^*cd}\}$, $\Sigma'' = \{a, b, c, d\}$, and the set $P''$ of productions

$$
\begin{aligned}
s &\rightarrow p_1|p_2|p_3|p_4|p_5 & q_{b^*cd} &\rightarrow q_{b^*c}\ d\\
q_{b^*} &\rightarrow b & p_1 &\rightarrow q_{b^*ab^*c}\\
q_{b^*ab^*} &\rightarrow a & p_2 &\rightarrow q_{b^*ab^*c}\\
q_{b^*c} &\rightarrow c & p_4 &\rightarrow q_{b^*cd}\\
q_{b^*} &\rightarrow q_{b^*}\ b & q_{b^*ab^*cd} &\rightarrow q_{b^*ab^*c}\ d\\
q_{b^*ab^*} &\rightarrow q_{b^*}\ a & q_{b^*cdab^*} &\rightarrow q_{b^*cd}\ a\\
q_{b^*c} &\rightarrow q_{b^*}\ c & p_3 &\rightarrow q_{b^*ab^*cd}\\
q_{b^*ab^*} &\rightarrow q_{b^*ab^*}\ b & p_5 &\rightarrow q_{b^*cdab^*}\\
q_{b^*ab^*c} &\rightarrow q_{b^*ab^*}\ c & q_{b^*cdab^*} &\rightarrow q_{b^*cdab^*}\ b.
\end{aligned}
$$

The program $\mathbf{P}''_{\mathrm{I}}$, which can be derived from these productions by transforming them to elementary chain rules, and by introducing the variables $Tr$, $O$, and $S$ at the appropriate positions (see 4.3.4), is

$$q_{b^*}(Tr, O, S, X, Y) \rightarrow b(Tr, O, S, X, Y). \tag{31}$$

$$q_{b^*ab^*}(Tr, O, S, X, Y) \rightarrow a(Tr, O, S, X, Y). \tag{32}$$

$$q_{b^*c}(Tr, O, S, X, Y) \rightarrow c(Tr, O, S, X, Y). \tag{33}$$

$$q_{b^*}(Tr, O, S, X, Y) \rightarrow q_{b^*}(Tr, O, S, X, X_1), b(Tr, O, S, X_1, Y). \tag{34}$$

$$q_{b^*ab^*(Tr,O,S,X,Y)} \rightarrow q_{b^*}(Tr, O, S, X, X_1), a(Tr, O, S, X_1, Y). \tag{35}$$

$$q_{b^*c}(Tr, O, S, X, Y) \rightarrow q_{b^*}(Tr, O, S, X, X_1), c(Tr, O, S, X_1, Y). \tag{36}$$

$$q_{b^*ab^*}(Tr, O, S, X, Y) \rightarrow q_{b^*ab^*}(Tr, O, S, X, X_1), b(Tr, O, S, X_1, Y). \tag{37}$$

$$q_{b^*ab^*c}(Tr, O, S, X, Y) \rightarrow q_{b^*ab^*}(Tr, O, S, X, X_1), c(Tr, O, S, X_1, Y). \tag{38}$$

$$
\begin{aligned}
q_{b^*cd}(Tr,O,S,X,Y) &\rightarrow q_{b^*c}(Tr,O,S,X,X_1), d(Tr,O,S,X_1,Y). && (39)\\
p_1(Tr,S,X,Y) &\rightarrow q_{b^*ab^*c}(Tr,O,S,X,Y). && (40)\\
p_2(Tr,S,X,Y) &\rightarrow q_{b^*ab^*c}(Tr,O,S,X,Y). && (41)\\
p_4(Tr,S,X,Y) &\rightarrow q_{b^*cd}(Tr,O,S,X,Y). && (42)\\
q_{b^*ab^*cd}(Tr,O,S,X,Y) &\rightarrow q_{b^*ab^*c}(Tr,O,S,X,X_1), d(Tr,O,S,X_1,Y). && (43)\\
q_{b^*cdab^*}(Tr,O,S,X,Y) &\rightarrow q_{b^*cd}(Tr,O,S,X,X_1), a(Tr,O,S,X_1,Y). && (44)\\
p_3(Tr,S,X,Y) &\rightarrow q_{b^*ab^*cd}(Tr,O,S,X,Y), && (45)\\
p_5(Tr,S,X,Y) &\rightarrow q_{b^*cdab^*}(Tr,O,S,X,Y), && (46)\\
q_{b^*cdab^*}(Tr,O,S,X,Y) &\rightarrow q_{b^*cdab^*}(Tr,O,S,X,X_1), b(Tr,O,S,X_1,Y). && (47)
\end{aligned}
$$

Our claim is that post-processing increases the coverage (see Definition 3 in Section 3) of the set of target predicates in $\mathcal{I}$. This claim is supported by the following lemma:

**Lemma 4** *Let $\mathbf{P}'_{\mathrm{I}}$ be a program, which has been mapped to a prefix acceptor $PA$. Let $PA''$ be the acceptor, which results from post-processing $PA$ for some EDB predicates of $\mathbf{P}'_{\mathrm{I}}$. Let $\mathbf{P}''_{\mathrm{I}}$ be the chain Datalog program, which corresponds to $PA'$ and $\mathcal{I}$ a set of target predicates $p_i \in \mathcal{I} \subseteq IDB(\mathbf{P}'_{\mathrm{I}}), IDB(\mathbf{P}''_{\mathrm{I}})$. Then,*

$$
\begin{array}{c}
\{p_i(t_1,\ldots,t_s) \mid p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in T^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}\\
\subseteq\\
\{p_i(t_1,\ldots,t_s) \mid p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in T^{\omega}_{\mathbf{P}''_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\},
\end{array}
$$

*i.e., the coverage of $\mathbf{P}'_{\mathrm{I}}$ for the target predicates in $\mathcal{I}$ is a subset of the coverage of $\mathbf{P}''_{\mathrm{I}}$ for $\mathcal{I}$: $Cov_{\mathbf{P}'}(\mathcal{I}) \subseteq Cov_{\mathbf{P}''}(\mathcal{I})$ with $\mathbf{P}' = \mathbf{P}'_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$ and $\mathbf{P}'' = \mathbf{P}''_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$.*

**Proof** We have EDB($\mathbf{P}'_{\mathrm{I}}$)=EDB($\mathbf{P}''_{\mathrm{I}}$). Furthermore, consider the sets of predicate symbols IDB($\mathbf{P}'_{\mathrm{I}}$)$-\mathcal{I}$ and IDB($\mathbf{P}''_{\mathrm{I}}$)$-\mathcal{I}$. They correspond to the states of the prefix acceptors $PA$ and $PA'$, respectively. Each state of $PA'$ is either a state of the original $PA$ or a state which resulted from merging several states of $PA$. The post-processing method guarantees, that each state of $PA$ is merged into at most one state of $PA'$. Therefore, there exists a mapping $f$ from the predicate symbols IDB($\mathbf{P}'_{\mathrm{I}}$)$\cup$EDB($\mathbf{P}'_{\mathrm{I}}$) to the predicate symbols IDB($\mathbf{P}''_{\mathrm{I}}$)$\cup$EDB($\mathbf{P}''_{\mathrm{I}}$) which is defined as follows

$$
f(r) = \begin{cases} r & \text{if } r \in \mathrm{EDB}(\mathbf{P}'_{\mathrm{I}})\text{=}\mathrm{EDB}(\mathbf{P}''_{\mathrm{I}}) \text{ or } r \in \mathcal{I}\\ r_i \in \mathrm{IDB}(\mathbf{P}''_{\mathrm{I}})-\mathcal{I} & \text{if } r \in \mathrm{IDB}(\mathbf{P}'_{\mathrm{I}})-\mathcal{I}. \end{cases}
$$

For our example programs, $\mathbf{P}'_{\mathrm{I}}$ and $\mathbf{P}''_{\mathrm{I}}$, this mapping is

$$
\begin{aligned}
f \;=\; & \{(p_i,p_i)|p_i \in \mathcal{I}\} \cup \{(a,a)|a \in \mathrm{EDB}(\mathbf{P}'_{\mathrm{I}})\text{=}\mathrm{EDB}(\mathbf{P}''_{\mathrm{I}})\} \cup\\
& \{(q_a,q_{b^*ab^*}),(q_b,q_{b^*}),(q_{ab},q_{b^*ab^*}),(q_{bc},q_{b^*c}),(q_{abc},q_{b^*ab^*c}),(q_{bcd},q_{b^*cd}),\\
& (q_{abcd},q_{b^*ab^*cd}),(q_{bcda},q_{b^*cdab^*}),(q_{bcdab},q_{b^*cdab^*})\}.
\end{aligned}
$$

If we apply this predicate renaming function to a rule $C$, we exchange each predicate symbol according to the function $f$. We denote the result by $f(C)$. Given that, for each rule $C' \in \mathbf{P}'_{\mathrm{I}}$, there exists a rule $C'' \in \mathbf{P}''_{\mathrm{I}}$, such that $f(C')$ is a variant[9] of $C''$. As $f$ is the identity function for $p_i \in \mathcal{I}$ and $a \in$ EDB($\mathbf{P}'_{\mathrm{I}}$)=EDB($\mathbf{P}''_{\mathrm{I}}$), it follows, that for each $\breve{B} = p_i(p_1, \ldots, p_s)$, if $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$, then $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}''_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$.□

## 6.4 Experiments

We have applied the method for structuring chain Datalog programs in prefix (tree) acceptors and the method for post-processing the acceptors to the data of the robot navigation domain developed within the BLearn-project. We worked with four data sets for four environments, denoted $P, Q, R$, and $S$ (see Figure 10, 11, 12, and 13).

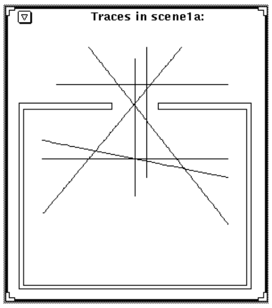


Figure 10: Traces for data set $P$

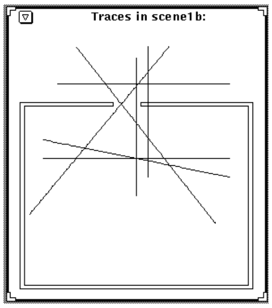


Figure 11: Traces for data set $Q$

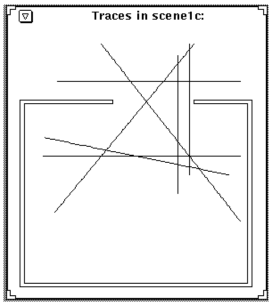


Figure 12: Traces for data set $R$

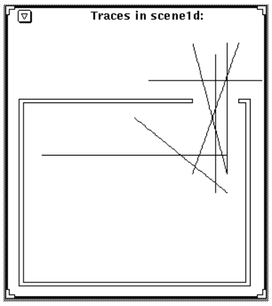


Figure 13: Traces for data set $S$

Each data set contains the measurements of 24 sonar sensors, which have been perceived during seven traces[10]. In Figure 14, the sequence of sonar sensor measurements is

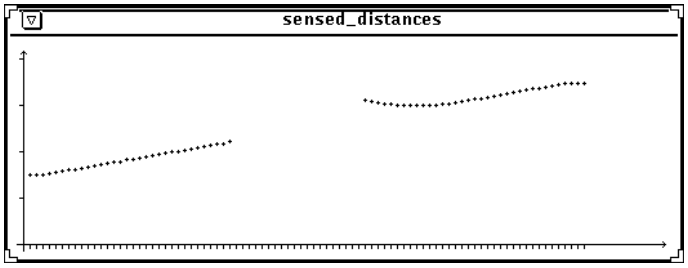


Figure 14: Sequence of sensor measurements

shown, which has been perceived by a sensor on the robot's left side during the trace in $P$, in which the robot moves diagonally along the doorway. Given the sonar sensor data, we generated the examples $E$ for the concepts to be learned, i.e., the sensor features. We applied the method, developed by Wessel [26], in order to calculate the basic features, which constitute the background knowledge $B$ for learning. The calculation of basic features is guided by a parameter, which represents the tolerance within which successive gradients of sensor measurements are considered to be approximately equal. This gradient is used

[9]Clauses $C_1$ and $C_2$ are *variants*, if there exist substitutions $\theta$ and $\sigma$ such that $C_1 = C_2\theta$ and $C_2 = C_1\sigma$ (see [11]).

[10]The data has been provided by the University of Karlsruhe.

to decide during the calculation, whether the measurement at a given time point is added to the time interval for the previous measurements or to a new interval for the next basic feature. By considering the tendency of change of successive measurements, i.e., the ratio between the values and not the absolute values themselves, we try to smooth out the inaccuracies of the sensor measurements. The effect of calculating basic features with

*Tolerance*=6:

```
         increasing(t7,75,s6,3,32).
     no_measurement(t7,75,s6,32,53).
         decreasing(t7,75,s6,53,59).
             stable(t7,75,s6,59,65).
         increasing(t7,75,s6,65,69).
something_happened(t7,75,s6,69,70).
         increasing(t7,75,s6,70,85).
```

*Tolerance*=15:

```
    increasing(t7,75,s6,3,32).
no_measurement(t7,75,s6,32,53).
        stable(t7,75,s6,53,69).
    increasing(t7,75,s6,69,85).
```

Figure 15: Different ways of calculating basic features

different parameters is shown in Figure 15 for the measurements in Figure 14. For each of the four data sets, $P, Q, R$, and $S$, we have calculated the basic features with four different tolerance values, i.e., 6, 8, 10, and 15.

Given the examples $E$ and the background knowledge $B$, i.e., the basic features calculated with one specific parameter value, we used for training the `prefix_tree` method to learn rules for deriving sensor features from basic features (see 4). In [10], we have already shown, that due to the sensor noise, the coverage of these rules is not very high. For this reason, we accepted rules, which covered at least one positive example. Then, we structured the rules in a prefix tree acceptor. During the post-processing phase we applied the procedure `post_proc` for the two basic features `no_movement` and `something_happened`.

Given the examples $E$ and the background knowledge $B$, i.e., the basic features calculated with one specific parameter value, we performed the training, post-processing, and testing phase four times with the training/test sets $QRS/P$, $PRS/Q$, $PQS/R$, and $PQR/S$. So we used the data of three environments for learning and tested the results with the data of the fourth environment. Thus, each row of Table 1 (and of Table 2),

| | | $PA$ before post-processing | | | $PA'$ after PP: Step 1 | | | | | Step 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| BF-Param | \|Train\| | $\|Q\|$ | $\|\Delta\|$ | Depth | $\|Q\|$ | $Q_{Red}$ | $\|\Delta\|$ | $\Delta_{Red}$ | Depth | $\|\Delta\|$ |
| $Tol$=6 | 1215 | 438 | 437 | 9 | 149 | 65.9% | 211 | 51.9% | 6 | 447 |
| $Tol$=8 | 1196 | 323 | 322 | 7 | 129 | 59.9% | 170 | 47.1% | 5 | 387 |
| $Tol$=10 | 1176 | 287 | 286 | 7 | 120 | 58.3% | 159 | 44.5% | 5 | 358 |
| $Tol$=15 | 1121 | 225 | 224 | 7 | 100 | 55.8% | 133 | 40.5% | 5 | 298 |
| | | | | 8 | | **60.0%** | | **46.0%** | 5 | |

Table 1: Complexity of the prefix acceptors before and after post-processing

which is indexed by a tolerance value contains the average results of four training/post-processing/test runs. The tables with the detailed results can be found in Appendix A.5. In Table 1, we present the results, which reflect the improvements with respect to the complexity of the acceptors, which we achieved by post-processing. BF-Param is the value for the tolerance parameter, which was used to calculate the basic features. |Train| denotes the number of training examples. $|Q|$ is the number of states, $|\Delta|$ is the number of transitions, and Depth is the maximal depth of the prefix acceptors before and after post-processing. $Q_{Red}$ denotes the percentage, by which the number of states was reduced, $\Delta_{Red}$ the percentage by which the number of transitions was reduced via post-processing. If we consider the columns for the $PA$ before post-processing, we see that the number of states and transitions decreases with increasing tolerance values. This reflects the fact, that the more sensitive the method for calculating basic features is, the longer the sequence of basic features become, yielding rules with long premise chains and large acceptors. After having performed the first post-processing step, we see, that also the reduction of the number of states and transitions decreases with increasing tolerance values. The average reduction of 60% of the states and 46% percent of the transitions is notable and justifies the effort to perform the first post- processing step. Obviously, the second one increases the complexity enormously. In Table 2, we present the results of testing the

| | | | No PP | PP: Step 1 | | PP: Step 2 | | |
|---|---|---|---|---|---|---|---|---|
| BF-Param | \|Train\| | \|Test\| | $C_0$ | $C_1$ | $I_{0,1}$ | $C_2$ | $I_{0,2}$ | $I_{1,2}$ |
| $Tol$=6 | 1215 | 405 | 59.9% | 67.8% | 7.9% | 68.7% | 8.8% | 0.9% |
| $Tol$=8 | 1196 | 399 | 61.4% | 67.6% | 6.2% | 68.4% | 7.1% | 0.9% |
| $Tol$=10 | 1176 | 392 | 61.3% | 66.2% | 4.9% | 66.9% | 5.6% | 0.7% |
| $Tol$=15 | 1121 | 374 | 62.4% | 66.2% | 3.8% | 68.2% | 4.3% | 0.5% |
| | | | 61.3% | 67.0% | **5.7%** | 68.1% | **6.5%** | **0.8%** |

Table 2: Testing results before and after post-processing

learning results before and after post-processing. During the testing phase we used the marker passing method, presented in Section 5, in order to derive sensor features from the basic features in the test sets, and compared them with the testing examples. |Test| denotes the number of testing examples, $C_0$ the percentage of correctly derived examples before post-processing, $C_1$ the percentage after the first, and $C_2$ the percentage after the second post-processing step. $I_{0,1}$ denotes the improvement, which we achieved by the first, $I_{0,2}$ the one achieved by the second post-processing step, when compared to the testing results before post-processing. $I_{1,2}$ denotes the improvement achieved by the second step compared to the results of the first post-processing step. The percentage of correctly derived test examples before post-processing increases with increasing tolerance values (see column $C_0$). After post-processing step 1, we have the opposite effect (see column $C_1$). We get the highest improvements for the case, that the basic features have been calculated with tolerance 6. The average improvement, we get, is 5.7%. We get only slightly better results for step 2. However, our claim, that the predictive power of the post-processed program/acceptors is increased, is confirmed. Obviously, the second post-processing step, which increases the complexity of the acceptor enormously, does not pay off, in terms of the improvements of the predictive power. So, in order to summarize, we can say, that the first post-processing step achieves good results in terms of the complexity and predictive power.

# 7 Restructuring, Marker Passing and Decompositions

By mapping a set of chain Datalog rules to a prefix acceptor, we have gained a compilation of rules, which allows to optimize forward chaining inferences. In Appendix A.4, we have added another example of a run of **MP** on the (post-processed) prefix acceptor $PA'$ in Figure 7. In this section, we show that this way of proceeding is similar to decomposing chain Datalog rules for query optimization (see the work by Dong and Ginsburg in [7]).

Consider the example program $\mathbf{P}'_{\mathrm{I}}$, which is the result of restructuring (see 4.3.2).

$$q_a(Tr,O,S,X,Y) \leftarrow a(Tr,O,S,X,Y). \tag{48}$$
$$q_b(Tr,O,S,X,Y) \leftarrow b(Tr,O,S,X,Y). \tag{49}$$
$$q_{ab}(Tr,O,S,X,Y) \leftarrow q_a(Tr,O,S,X_1,X_2), b(Tr,O,S,X_2,Y). \tag{50}$$
$$q_{bc}(Tr,O,S,X,Y) \leftarrow q_b(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). \tag{51}$$
$$q_{abc}(Tr,O,S,X,Y) \leftarrow q_{ab}(Tr,O,S,X_1,X_2), c(Tr,O,S,X_2,Y). \tag{52}$$
$$q_{bcd}(Tr,O,S,X,Y) \leftarrow q_{bc}(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,Y). \tag{53}$$
$$p_1(Tr,S,X,Y) \leftarrow q_{abc}(Tr,O,S,X,Y). \tag{54}$$
$$p_2(Tr,S,X,Y) \leftarrow q_{abc}(Tr,O,S,X,Y). \tag{55}$$
$$p_4(Tr,S,X,Y) \leftarrow q_{bcd}(Tr,O,S,X,Y). \tag{56}$$
$$q_{abcd}(Tr,O,S,X,Y) \leftarrow q_{abc}(Tr,O,S,X_1,X_2), d(Tr,O,S,X_2,Y). \tag{57}$$
$$q_{bcda}(Tr,O,S,X,Y) \leftarrow q_{bcd}(Tr,O,S,X_1,X_2), a(Tr,O,S,X_2,Y). \tag{58}$$
$$p_3(Tr,S,X,Y) \leftarrow q_{abcd}(Tr,O,S,X,Y). \tag{59}$$
$$q_{bcdab}(Tr,O,S,X,Y) \leftarrow q_{bcda}(Tr,O,S,X_1,X_2), b(Tr,O,S,X_2,Y). \tag{60}$$
$$p_5(Tr,S,X,Y) \leftarrow q_{bcdab}(Tr,O,S,Y,Y). \tag{61}$$

Based on the notion of *dependent rules*, we can decompose the rules of program $\mathbf{P}'_{\mathrm{I}}$ into disjoint sets of rules.

**Definition 6** *([7]) Given a basic logic program $\mathbf{P}_{\mathrm{I}}$ and two rules $C_1, C_2 \in \mathbf{P}_{\mathrm{I}}$, $C_1$ is said to* depend *on $C_2$ (in $\mathbf{P}_{\mathrm{I}}$), denoted by $C_1 \succ_{\mathbf{P}_{\mathrm{I}}} C_2$, if either the predicate occurring in the head of $C_2$ occurs in the body of $C_1$, or there is a rule $C \in \mathbf{P}_{\mathrm{I}}$, such that $C_1 \succ_{\mathbf{P}_{\mathrm{I}}} C$ and $C \succ_{\mathbf{P}_{\mathrm{I}}} C_2$.*

The direct dependencies among the rules of $\mathbf{P}'_{\mathrm{I}}$ are r61 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r60 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r58 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r53 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r51 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r49, r59 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r57 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r52 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r50 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r48, r56 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r53, r55 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r52, and r54 $\succ_{\mathbf{P}_{\mathrm{I}}'}$ r52.

Similar to Dong and Ginsburg [7], we define a program decomposition as follows:

**Definition 7** *For a given set of IDB predicates $\mathcal{I} = \{p_1, \ldots, p_n\} \subseteq$IDB($\mathbf{P}_{\mathrm{I}}$) of a basic logic program $\mathbf{P}_{\mathrm{I}}$ a sequence $\mathbf{P}_{\mathrm{I,1}} \ldots \mathbf{P}_{\mathrm{I,n}} (n \geq 1)$ of programs is called a $\{p_1, \ldots, p_n\}$-* decomposition *of $\mathbf{P}_{\mathrm{I}}$ if*

$$\begin{array}{rcl} \{p_i(t_1,\ldots,t_n) & | & p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_n) \in T^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ T^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(I)\} \\ & = & \\ \{p_i(t_1,\ldots,t_n) & | & p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_n) \in T^{\omega}_{\mathbf{P}_{\mathrm{I}}}(I)\} \end{array}$$

*for interpretations $I$, which are restricted to be EDB instances of $\mathbf{P}_{\mathrm{I}}$.*

Here, $\circ$ denotes a composition of mappings, where the component mappings are applied from right to left. Each $\mathbf{P}_{\mathrm{I,i}}$ is called a *component program* or simply *component* of the decomposition. Note, that $\mathbf{P}_{\mathrm{I,1}} \cup \ldots \cup \mathbf{P}_{\mathrm{I,n}}$ does not have to coincide with $\mathbf{P}_{\mathrm{I}}$, i.e., new predicates are introduced. In our case it is the program $\mathbf{P}'_{\mathrm{I}} = \mathbf{P}_{\mathrm{I,1}} \cup \ldots \cup \mathbf{P}_{\mathrm{I,n}}$, which is the result of applying the procedure `restruct` or `restruct_dc` to the original program $\mathbf{P}_{\mathrm{I}}$.

The method `decompose` (see Algorithm 11) finds one possible decomposition of a (restructured) program. The rules with no IDB predicates in their bodies are put into the first component. Then, we repeat the following step until each rule has been assigned to a component: Add to component $i$ all rules, which depend direcly on some rule in component $i - 1$.

---

`decompose`($\mathbf{P}_{\mathrm{I}}$)
**begin**

1. $\mathbf{P}_{\mathrm{I,1}} := \{C | C \in \mathbf{P}_{\mathrm{I}}$ and $C_{body}$ consists of EDB predicates only $\}$;
2. $ToDo := \mathbf{P}_{\mathrm{I}} - \mathbf{P}_{\mathrm{I,1}}$;
3. $i = 2$;
4. **while** $ToDo \neq \emptyset$
   **begin**
   $\mathbf{P}_{\mathrm{I,i}} := \{C | C \in ToDo$ and $C \succ_{\mathbf{P}_{\mathrm{I}}} C_j$ with $C_j \in \mathbf{P}_{\mathrm{I,i\text{-}1}}\}$;
   $ToDo := ToDo - \mathbf{P}_{\mathrm{I,i}}$;
   $i := i + 1$;
   **end**
5. **return** $\mathbf{P}_{\mathrm{I,1}} \ldots \mathbf{P}_{\mathrm{I,i\text{-}1}}$;

**end**

Algorithm 11: `decompose`

---

If we apply this method to $\mathbf{P}'_{\mathrm{I}}$, we get the components

$$\begin{aligned}
\mathbf{P}_{\mathrm{I,1}} &= \{\text{ r48, r49}\}\\
\mathbf{P}_{\mathrm{I,2}} &= \{\text{ r50, r51}\}\\
\mathbf{P}_{\mathrm{I,3}} &= \{\text{ r52, r53}\}\\
\mathbf{P}_{\mathrm{I,4}} &= \{\text{ r54, r55, r56, r57, r58}\}\\
\mathbf{P}_{\mathrm{I,5}} &= \{\text{ r59, r60}\}\\
\mathbf{P}_{\mathrm{I,6}} &= \{\text{ r61}\}
\end{aligned}$$

$\mathbf{P}_{\mathrm{I,1}}$ contains the rules, which do not depend on any other rule of $\mathbf{P}'_{\mathrm{I}}$, $\mathbf{P}_{\mathrm{I,2}}$ contains the rules, which depend directly on those in $\mathbf{P}_{\mathrm{I,1}}$, $\mathbf{P}_{\mathrm{I,3}}$ contains the rules, which depend directly on those in $\mathbf{P}_{\mathrm{I,2}}$, etc. For each interpretation $I$, which is an EDB instance for $\mathbf{P}'_{\mathrm{I}}$, the minimum Herbrand model can be determined by first computing the fixpoint $F1$ of $\mathrm{T}_{\mathbf{P}_{\mathrm{I,1}}}$ on $I$, followed by the fixpoint $F2$ of $\mathrm{T}_{\mathbf{P}_{\mathrm{I,2}}}$ on $F1$, followed by the fixpoint $F3$ of $\mathrm{T}_{\mathbf{P}_{\mathrm{I,3}}}$ on $F2$, etc. There is no need to consider computations, where the rules in $\mathbf{P}_{\mathrm{I,6}}$ are applied first, followed by the application of other rules. So the sequence $\mathbf{P}_{\mathrm{I,1}}, \ldots, \mathbf{P}_{\mathrm{I,6}}$ is a decomposition of program $\mathbf{P}'_{\mathrm{I}}$ for the target predicates $p_1, \ldots, p_5$. Note, that decompositions are not unique. The `decompose`-method finds the one with maximal components. For the post-processed program $\mathbf{P}''_{\mathrm{I}}$ (see 6.2), `decompose` finds the $\{p_1, p_2, p_3, p_4, p_5\}$-decomposition $\mathbf{P}''_{\mathrm{I},1}\mathbf{P}''_{\mathrm{I},2}\mathbf{P}''_{\mathrm{I},3}\mathbf{P}''_{\mathrm{I},4}$

$$
\begin{array}{lcl}
\mathbf{P}''_{\mathrm{I},1} & = & \{\ \mathrm{r31},\ \mathrm{r32},\ \mathrm{r33}\} \\
\mathbf{P}''_{\mathrm{I},2} & = & \{\ \mathrm{r34},\ \mathrm{r35},\ \mathrm{r36},\ \mathrm{r37},\ \mathrm{r38},\ \mathrm{r39}\} \\
\mathbf{P}''_{\mathrm{I},3} & = & \{\ \mathrm{r40},\ \mathrm{r41},\ \mathrm{r42},\ \mathrm{r43},\ \mathrm{r44}\} \\
\mathbf{P}''_{\mathrm{I},4} & = & \{\ \mathrm{r45},\ \mathrm{r46},\ \mathrm{r47}\}
\end{array}
$$

The purpose of decompositions is to divide programs into smaller clusters, in order to achieve more efficient evaluations of programs. As a side-effect, some interactions among rules may be removed. Here, it is the redundant evaluation of premise chains, which are prefixes of other premise chains. Separation of these interactions may also help a user to better understand the programs. From a sequential processing point of view, each rule $C$ in a decomposition is evaluated after those, on which $C$ depends. For example, rule r60 is evaluated after rule r58, which is evaluated after rule r53, etc. From a parallel processing point of view, if each rule in a component program is independent of any other rule in the same component, they can be processed in parallel.

Now, assume that the robot perceives the sequence of ground basic feature predicates, i.e., that the basic logic program $\mathbf{P}'_{\mathrm{I}} = \mathbf{P}_{\mathrm{I},1} \cup \ldots \cup \mathbf{P}_{\mathrm{I,n}}$ gets as input the EDB instance

$$\mathbf{P}_{\mathrm{E}} = \{a(t1, 90, s5, 1, 8), b(t1, 90, s5, 8, 10), c(t1, 90, s5, 10, 15), d(t1, 90, s5, 15, 17)\}.$$

Now, if we calculate $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},6}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},5}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},4}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},3}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}})$ according to Definition 7, we get

$$
\begin{array}{rcl}
F1 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}}) & = & \mathbf{P}_{\mathrm{E}} \cup \{q_a(t1, 90, s5, 1, 8), q_b(t1, 90, s5, 8, 10)\} \\
F2 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}}(F1) & = & F1 \cup \{q_{ab}(t1, 90, s5, 1, 10), q_{bc}(t1, 90, s5, 8, 15)\} \\
F3 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},3}}(F2) & = & F2 \cup \{q_{abc}(t1, 90, s5, 1, 15), q_{bcd}(t1, 90, s5, 8, 17)\} \\
F4 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},4}}(F3) & = & F3 \cup \{p_1(t1, s5, 1, 15), p_2(t1, s5, 1, 15), p_4(t1, s5, 8, 17), \\
 & & \quad q_{abcd}(t1, 90, s5, 1, 17)\} \\
F5 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},4}}(F4) & = & F4 \cup \{p_3(t1, s5, 1, 17)\} \\
F6 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},5}}(F5) & = & F5.
\end{array}
$$

For this example, it is obvious, that

$$F5 = \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},6}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},5}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},4}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},3}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}}) = \mathrm{T}^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}) \supseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$$

and that

$$
\begin{array}{rcl}
\{p_i(tr, s, x, y) & | & p_i \in \mathcal{I} = \{p_1, p_2, \ldots, p_5\} \text{ and} \\
 & & p_i(tr, s, x, y) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},6}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},5}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},4}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},3}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}})\} \\
 & = & \\
\{p_i(tr, s, x, y) & | & p_i \in \mathcal{I} = \{p_1, p_2, \ldots, p_5\} \text{ and } p_i(tr, s, x, y) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} \\
 & = & \{p_1(t1, s5, 1, 15), p_2(t1, s5, 1, 15), p_3(t1, s5, 1, 17), p_4(t1, s5, 8, 17)\}.
\end{array}
$$

i.e., the IDB-portion of the minimum Herbrand model of the predicates $p_1, \ldots, p_5$ in $\mathcal{I}$ is the same, no matter whether we apply the T mapping for the whole program or sequentially for its components. We show the validity of this relation in the following lemma:

**Lemma 5** *Let* $\mathbf{P}_{\mathrm{I}}$ *be a*

- *a non-recursive basic program with rules, in which the IDB-predicates occur only in rule heads, to which we apply one of the restructuring methods, presented in 4.3, or*
- *a linear basic logic program with rules which have at most one EDB-subgoal,*

*then the method* `decompose` *generates a decomposition* $\mathbf{P}_{\mathrm{I,1}} \ldots \mathbf{P}_{\mathrm{I,n}}$ *such that*

$$\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in T^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\}$$

$$=$$

$$\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in T^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ T^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\},$$

where $\mathcal{I} \subseteq$IDB$(\mathbf{P}_{\mathrm{I}})$.

**Proof** Remember, that the fixpoint $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$ can be determined in a finite number of steps. In Section 3, we defined $\mathrm{T}^{0}_{\mathbf{P}_{\mathrm{I}}}$ for a given EDB instance $\mathbf{P}_{\mathrm{E}}$ to be $\mathrm{T}^{0}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}) = \mathbf{P}_{\mathrm{E}}$. Note, that $\mathcal{I} \subseteq$IDB$(\mathbf{P}_{\mathrm{I}})\subseteq$IDB$(\mathbf{P}_{\mathrm{I,1}} \cup \ldots \cup \mathbf{P}_{\mathrm{I,n}})$. So, in order to show the $\subseteq$-part, it suffices to show $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}) \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$.

$\subseteq$**:** We show this part by induction on the number $i$ of applications of the $\mathrm{T}_{\mathbf{P}_{\mathrm{I}}}$ mapping, necessary to calculate the fixpoint $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$.

If $i = 1$ and $\breve{B} \in \mathrm{T}^{1}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$ with $\breve{B} = p_r(x_1, \ldots, x_s)$ where $p_r \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$. Then, there are two possibilities. The first is, that there exists a rule $C \in \mathbf{P}_{\mathrm{I}}$ with one premise and $C\sigma = (\breve{B} \leftarrow \breve{A})$, such that $\breve{A} \in \mathbf{P}_{\mathrm{E}}$. The restructuring method does not change any rules with one premise. So $C \in \mathbf{P}_{\mathrm{I,1}}$ and $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. The other possibility is, that $C$ is a rule with more than one premise with $C\sigma = (\breve{B} \leftarrow \breve{A}_1, \ldots, \breve{A}_n)$ and $\breve{A}_1, \ldots, \breve{A}_n \in \mathbf{P}_{\mathrm{E}}$. During the restructuring phase $C$ has been transformed to the rules $(Q_1 \leftarrow A_1), (Q_2 \leftarrow Q_1, A_2), \ldots, (Q_l \leftarrow Q_{l-1}, A_l), (B \leftarrow Q_l)$. Therefore, $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,l+1}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. This takes care of the induction basis.

Suppose, as the induction assumption, that $\mathrm{T}^{i}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}) \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. Then, we have to show the hypothesis $\mathrm{T}^{i+1}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}) \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. Let $\breve{B} = p_r(x_1, \ldots, x_s)$ and $\breve{B} \in \mathrm{T}^{i+1}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. Then, there exists a clause $C \in \mathbf{P}_{\mathrm{I}}$ with $C\sigma = (\breve{B} \leftarrow \breve{A}_1, \ldots, \breve{A}_l)$ and $\breve{A}_1, \ldots, \breve{A}_l \in \mathrm{T}^{i}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. According to the assumption, $\breve{A}_1, \ldots, \breve{A}_l \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. Consider the component $\mathrm{T}_{\mathbf{P}_{\mathrm{I,t}}}, 1 \leq t < n$, with $C \in \mathrm{T}_{\mathbf{P}_{\mathrm{I,t}}}$. In order to be able to apply $C$, we have to show, that the $\breve{A}_1 \ldots \breve{A}_l$ are already in the set to which the $\mathrm{T}_{\mathbf{P}_{\mathrm{I,t}}}$ mapping is applied or that they are added to the interpretation during the calculation of the fixpoint $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,t}}}$. Assume the contrary, i.e., $\breve{A}_1 \ldots \breve{A}_l \notin \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,t}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$. Then, either $\breve{A}_1 \ldots \breve{A}_l \notin \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$, which is a contradiction to the assumption. Or $\breve{A}_1 \ldots \breve{A}_l \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})$, but $\breve{A}_1 \ldots \breve{A}_l$ are calculated only after the fixpoint $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,t}}}$ has been determined. From this follows, that for at least one $\breve{A}_v, v \in \{1, \ldots, l\}$, there is a component $\mathbf{P}_{\mathrm{I},v}$ with $v > t$, which contains a rule $C_v = (A_v \leftarrow C_{body})$. Clearly, we have $C \succ C_v$. This again leads to a contradiction, because according to our method for determining the components, we have $C \not\succ C_w$ for each rule $C$ in a given component $\mathbf{P}_{\mathrm{I,t}}, 1 \leq t < n$ and any rule $C_w \in \mathbf{P}_{\mathrm{I},x}, t < x \leq n$.

$\supseteq$: Again, it suffices to show, that $\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{E}})$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\} \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. We show this part by induction on the number $n$ of components of the decomposition.

*Case* $n = 1$. Each rule $C \in \mathbf{P}_{\mathrm{I,1}}$ for a predicate $p_i(X_1, \ldots, X_s)$ with $p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$ is either a member of $\mathbf{P}_{\mathrm{I}}$ or it can be unfolded, yielding $C_{unfolded}$ with $C_{unfolded} \in \mathbf{P}_{\mathrm{I}}$. From this follows, that $\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{E}})$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\} \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$.

*Case* $n > 1$. For the induction basis, we have to show, that $\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{E}})$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\} \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. The component $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}$ contains all rules, which are independent of any other rule in $\mathbf{P}_{\mathrm{I,1}} \cup \ldots \cup \mathbf{P}_{\mathrm{I,n}}$. All rules $C \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}$ have the form $B \leftarrow A$. Therefore, $A$ has to be an atom over a predicate in $\mathrm{EDB}(\mathbf{P}_{\mathrm{I}})$. If $B$ is an atom over a predicate $p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$, then, according to the restructuring method, the rule $B \leftarrow A$ is also in $\mathbf{P}_{\mathrm{I}}$. So if $\breve{B} = p_r(x_1, \ldots, x_s)$ with $p_r \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$ and $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}$, then there exists a $C \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}$, with $C\sigma = (\breve{B} \leftarrow \breve{A})$ and $\breve{A} \in \mathbf{P}_{\mathrm{E}}$. As $C \in \mathbf{P}_{\mathrm{I}}$, we also have $\breve{B} \in \mathrm{T}^{1}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$, and therefore $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$.

Let $I_{1,\ldots,j}, 1 < j \leq n$ denote the set $\{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{E}})$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,j}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\}$. Suppose, as the induction assumption, that $I_{1,\ldots,j} \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}}), 1 < j < n$. Then, we have to show the induction hypothesis $I_{1,\ldots,j+1} \subseteq \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$ with $I_{1,\ldots,j+1} = \{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{E}})$ and $p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,j+1}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\}$. Let $\breve{B} = p_r(t_1, \ldots, t_s)$ with $p_r \in \mathcal{I} \subseteq \mathrm{IDB}(\mathbf{P}_{\mathrm{I}})$ and $\breve{B} \in I_{1,\ldots,j+1}$. Then, there are two possibilities. The first is, that $\breve{B} \in I_{1,\ldots,j}$ and thus $\breve{B} \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$ according to the assumption. If that is not the case, then there exists a $C \in \mathrm{T}_{\mathbf{P}_{\mathrm{I,j+1}}}$, such that $C\sigma = (\breve{B} \leftarrow \breve{A}_1, \ldots, \breve{A}_l)$ and $\breve{A}_1, \ldots, \breve{A}_l \in \mathrm{T}^{r}_{\mathbf{P}_{\mathrm{I,j+1}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}}), r < \omega$. This rule $C$ is either a member of $\mathbf{P}_{\mathrm{I}}$ or it can be unfolded, such that $C_{unfolded}$ is a member of $\mathbf{P}_{\mathrm{I}}$. From this follows, that $\breve{B}$ will also be in $\mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})$. $\square$

With Lemma 2 (see 4.3.3), Theorem 1 and 2 (see 5.2), and Lemma 5, we have

$$
\begin{aligned}
\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}}) &\overset{Theorems\ 1,\ 2}{=} \{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}'_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} \\
&\overset{Lemma\ 2}{=} \{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I}}}(\mathbf{P}_{\mathrm{E}})\} \\
&\overset{Lemma\ 5}{=} \{p_i(t_1, \ldots, t_s) | p_i \in \mathcal{I} \text{ and } p_i(t_1, \ldots, t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,1}}}(\mathbf{P}_{\mathrm{E}})\}
\end{aligned}
$$

Furthermore, we have

$$\{p_i(t_1,\ldots,t_s)|p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}}) = \Gamma^1(\mathcal{C}^1_{PA})$$

$$\{p_i(t_1,\ldots,t_s)|p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}})\} = \Gamma^{i_2}(\mathcal{C}^{i_2}_{PA}) \cup \ldots \cup \Gamma^1(\mathcal{C}^1_{PA})$$

$$\{p_i(t_1,\ldots,t_s)|p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},3}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},2}} \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}})\}$$
$$= \Gamma^{i_3}(\mathcal{C}^{i_3}_{PA}) \cup \ldots \cup \Gamma^{i_2}(\mathcal{C}^{i_2}_{PA}) \cup \ldots \cup \Gamma^1(\mathcal{C}^1_{PA})$$
$$\ldots$$
$$\{p_i(t_1,\ldots,t_s)|p_i \in \mathcal{I} \text{ and } p_i(t_1,\ldots,t_s) \in \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}^{\omega}_{\mathbf{P}_{\mathrm{I},1}}(\mathbf{P}_{\mathrm{E}})\}$$
$$= \Gamma^k(\mathcal{C}^k_{PA}) \cup \ldots \cup \Gamma^1(\mathcal{C}^1_{PA}) = \mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$$

with $1 < i_2 < i_3 < \ldots < k$, where $k$ is the length of the input chain $\mathbf{P}_{\mathrm{E}}$.

The process of incrementally calculating the fixpoint of the T mapping, i.e., of calculating incrementally the minimum Herbrand model, can also be illustrated by the complete derivation trees for $p_1(t1, s5, 1, 15)$, $p_3(t1, s5, 1, 17)$, and $p_4(t1, s5, 8, 17)$, which are presented in Figure 16, 17, and 18, respectively. The sequential application of the T

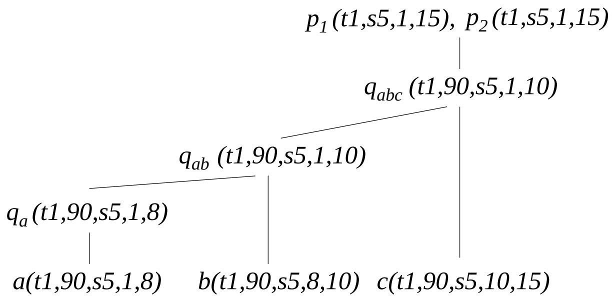


Figure 16: Derivation tree for $p_1(t1, s5, 1, 15)$

mapping for the component programs and the passing forward of the marker in the prefix acceptor is equivalent to constructing the derivation trees incrementally from left to right and bottom-up. This incremental construction is exactly simulated by the marker passing method, which we have presented in Section 5. The difference is, that the marker passing method does not calculate the IDB-portion of the minimum Herbrand model for the auxiliary IDB predicates $q_i \in IDB(\mathbf{P}'_{\mathrm{I}}) - \mathcal{I}$. Remember, that the order of the EDB facts in $\mathbf{P}_{\mathrm{E}}$ corresponds to the relation $\ll$. Due to the syntactical characteristics of chain Datalog programs, it can never happen, that, given the chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1 \ldots \breve{A}^k$, some permuted subsequence of it appears as the fringe of a complete derivation tree. So, the point we want to make, is, that the compilation of a chain Datalog program into a prefix acceptor and the application of the marker passing method for efficient forward inferences corresponds to the decomposition of chain Datalog programs and to calculating the minimum Herbrand model by calculating sequentially the fixpoint of the T mapping to the components of the decomposition starting with a given an EDB-instance.

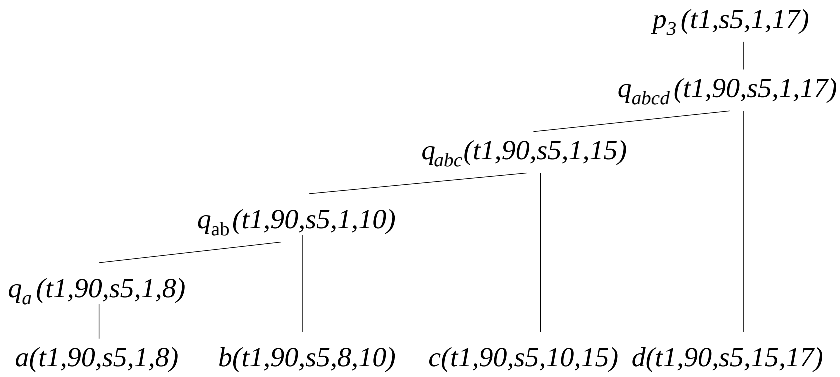


Figure 17: Derivation tree for $p_3(t1, s5, 1, 17)$

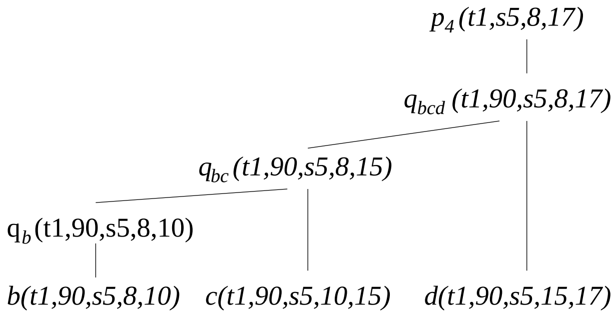


Figure 18: Derivation tree for $p_4(t1, s5, 8, 17)$

**Related work** Dong and Ginsburg have introduced uniform decompositions (see [6]) and $p$-decompositions (see [7]). A sequence $\mathbf{P}_{\mathrm{I},1} \ldots \mathbf{P}_{\mathrm{I,n}} (n \geq 1)$ of programs is called a *uniform decomposition* of program $\mathbf{P}_{\mathrm{I}}$, if $\mathrm{T}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}_{\mathbf{P}_{\mathrm{I},1}}(I) = \mathrm{T}_{\mathbf{P}_{\mathrm{I}}}(I)$ for every interpretation $I$ of $\mathbf{P}_{\mathrm{I}}$. For a predicate $p$, a sequence $\mathbf{P}_{\mathrm{I},1} \ldots \mathbf{P}_{\mathrm{I,n}} (n \geq 1)$ of programs is called a *p-decomposition* of program $\mathbf{P}_{\mathrm{I}}$, if $\{p(t_1, t_2) | p(t_1, t_2) \in \mathrm{T}_{\mathbf{P}_{\mathrm{I,n}}} \circ \ldots \circ \mathrm{T}_{\mathbf{P}_{\mathrm{I},1}}(I)\} = \{p(t_1, t_2) | p(t_1, t_2) \in \mathrm{T}_{\mathbf{P}_{\mathrm{I}}}(I)\}$. Common to all types of decompositions is the ordered, compositional manner of computation of the component programs. The differences between $\{p_1, \ldots, p_n\}$-decompositions, on one hand, and $p$-decompositions and uniform decompositions, on the other hand, are the following: Like $p$-decompositions, $\{p_1, \ldots, p_n\}$-decompositions take as input only EDB instances of $\mathbf{P}_{\mathrm{I}}$, whereas uniform decompositions take as input interpretations of both, IDB and EDB predicates. The decompositions differ in the predicates, for which they "simulate" the original program $\mathbf{P}_{\mathrm{I}}$: A uniform decomposition "simulates" $\mathbf{P}_{\mathrm{I}}$ for every IDB predicate in IDB($\mathbf{P}_{\mathrm{I}}$), a $p$-decomposition "simulates" $\mathbf{P}_{\mathrm{I}}$ only for one predicate $p$. Finally, $\{p_1, \ldots, p_n\}$-decompositions "simulate" $\mathbf{P}_{\mathrm{I}}$ for a subset of IDB predicates $\{p_1, \ldots, p_n\} \subseteq$ IDB($\mathbf{P}_{\mathrm{I}}$). Like $p$-decompositions, but in contrast to uniform decompositions, $\{p_1, \ldots, p_n\}$-decompositions may use newly introduced predicates, i.e., predicates not in EDB($\mathbf{P}_{\mathrm{I}}$) $\cup$ IDB($\mathbf{P}_{\mathrm{I}}$). For example, the $\{p_1, p_2, p_3, p_4, p_5\}$-decomposition $\mathbf{P}_{\mathrm{I},1}\mathbf{P}_{\mathrm{I},2}\mathbf{P}_{\mathrm{I},3}\mathbf{P}_{\mathrm{I},4}\mathbf{P}_{\mathrm{I},5}\mathbf{P}_{\mathrm{I},6}$ uses nine newly introduced predicates $q_j, j \in$

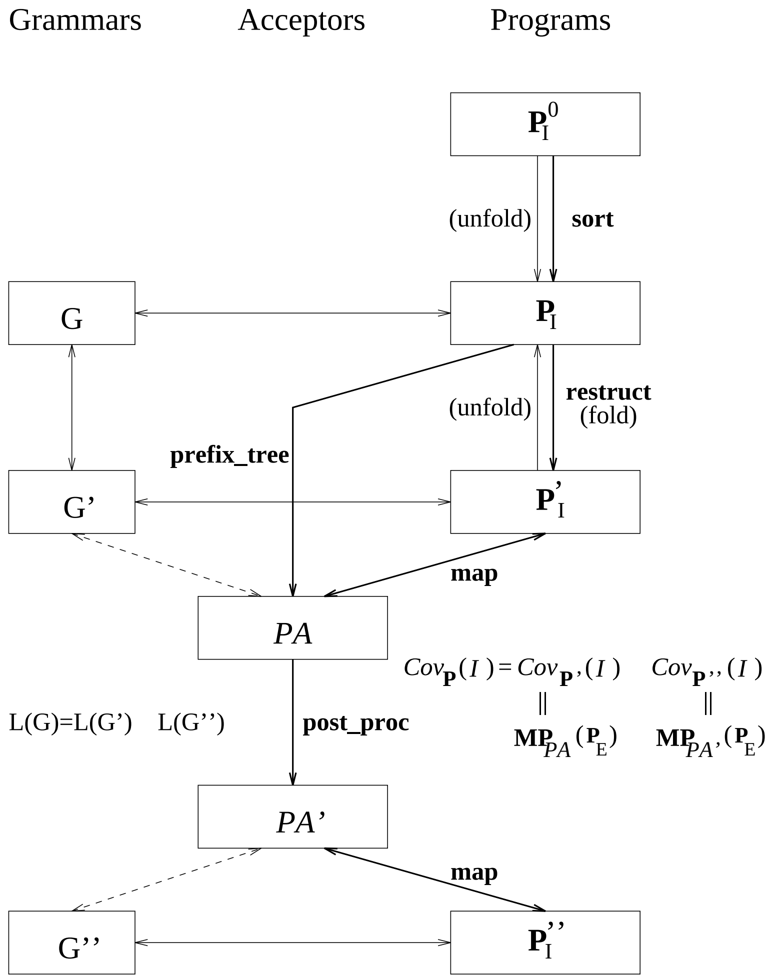


Figure 19: Summary

$Prefix(\hat{L}(G)) = Prefix(\hat{L}(G')) - \{\epsilon\}$.

# 8 Conclusions

## 8.1 Summary

Figure 19 gives an overview of the work presented in this paper. We started with a non-recursive chain Datalog program $\mathbf{P}_\mathrm{I}^0$, whose rules define intensionally several target concepts represented by the predicates with the symbols $p_i \in \mathcal{I} \subseteq IDB(\mathbf{P}_\mathrm{I}^0)$, which occur only in rule heads. We have used the syntactical features of chain Datalog programs to develop methods, which sort automatically the premise literals of a chain Datalog rule according to the relations, $\ll$ and $\rightsquigarrow$, respectively. By sorting the premise literals and by unfolding the rules for the IDB predicates in all possible ways, we can transform an arbitrary non-recursive program $\mathbf{P}_\mathrm{I}^0$ to a non-recursive chain Datalog program $\mathbf{P}_\mathrm{I}$ with sorted premise chains and with all its IDB predicates $p_i \in \mathcal{I} \subseteq (\mathbf{P}_\mathrm{I})$ occurring in rule heads only. We have used the correspondence between chain Datalog programs and CFGs to characterize $\mathbf{P}_{\mathrm{I}}$ by the regular grammar $G$. The method `prefix_tree` takes as input a chain Datalog program of the above mentioned type, structures the rules in a prefix tree and maps the tree to a prefix tree acceptor $PA$. We can obtain the same result, if we restructure the program $\mathbf{P}_{\mathrm{I}}$ with one of the methods presented in 4.3. The rules of the resulting program $\mathbf{P}'_{\mathrm{I}}$ have a special form, which allowed us to define a procedure to map the rules directly to the prefix acceptor and vice versa. Again, the restructured program $\mathbf{P}'_{\mathrm{I}}$ can be characterized by a regular left-linear grammar $G'$, which can also be obtained from the transitions of the DFA accepting the language $\hat{L}(G)$. The restructuring methods do not change the coverage of the target concepts represented by the $p_i \in \mathcal{I}$, i.e., $Cov_{\mathbf{P}}(\mathcal{I}) = Cov_{\mathbf{P}'}(\mathcal{I})$. The goal of post-processing is to increase the coverage of these target predicates. The method `post_proc` transforms the $PA$ by deleting non-cyclic transitions for some EDB predicates, by merging the affected states and transitions, and by introducing cyclic transitions. This is a generalization step. In terms of the grammar, the language $\hat{L}(G) = \hat{L}(G')$ is generalized to $\hat{L}(G'')$, such that $\hat{L}(G) = \hat{L}(G') \subseteq \hat{L}(G'')$, where $G''$ is the grammar corresponding to $PA'$. The post-processed acceptor $PA'$ can be mapped to a linear chain Datalog program $\mathbf{P}''_{\mathrm{I}}$ (see Definition 1), whose coverage is a superset of the one of $\mathbf{P}'_{\mathrm{I}}$ and $\mathbf{P}_{\mathrm{I}}$, respectively, i.e., $Cov_{\mathbf{P}}(\mathcal{I}) = Cov_{\mathbf{P}'}(\mathcal{I}) \subseteq Cov_{\mathbf{P}''}(\mathcal{I})$ with $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}, \mathbf{P}' = \mathbf{P}'_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$ and $\mathbf{P}'' = \mathbf{P}''_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$.

The original rules in $\mathbf{P}_{\mathrm{I}}$ are used to infer for a given ground chain $\mathbf{P}_{\mathrm{E}}$ via forward inferences the higher-level concepts represented by $p_i \in \mathcal{I}$. The reasons for optimizing the program and the inference procedure, respectively, are the prefix effect and the ambiguities, which require to match EDB facts redundantly with premise literals of several rules. We have presented an efficient marker passing method, which is sound and complete, i.e., its success set $\mathbf{MP}_{PA}(\mathbf{P}_{\mathrm{E}})$ for a given EDB instance $\mathbf{P}_{\mathrm{E}}$ is equal to the subset of the minimum Herbrand model for the predicates in $\mathcal{I}$ of the extended program $\mathbf{P} = \mathbf{P}_{\mathrm{I}} \cup \mathbf{P}_{\mathrm{E}}$, where $\mathbf{P}_{\mathrm{I}}$ is the program compiled in the respective prefix acceptor.

With the restructuring methods we have contributed to the field of theory restructuring, whose goal it is to transform a program without changing the coverage of the learned concepts. We map pairs of existing terms to a new combined term, in order to support more efficient evaluations. These evaluations are realized by the marker passing method **MP**. The post-processing phase is a generalization step in which the coverage of the learned concepts is increased and the complexity of the prefix acceptors is reduced. Furthermore, we have shown the relation of rule structuring and marker passing, on one hand, and program decompositions for query optimization of chain Datalog programs, on the other hand. So our methods can be considered as efficient implementations of the theoretical concepts introduced by Dong, Ginsburg and others. Finally, we have applied all the methods successfully to a robotics domain, thus contributing to applications of machine learning methods to real-world domains.

## 8.2 Current and future work

The relation of the restructuring method to inverse resolution and inter-construction has to be elaborated more formally, i.e., we have to show, that the selection of the variables for the invented predicates preserves soundness and correctness.

The idea of rule structuring and marker passing can also be applied to the chain

Datalog programs for operational concepts (see, e.g., [22],[10]). Operational concepts are defined in terms of perception-integrating action features (see Figure 1 in Section 2), which define the pre-condition for executing the concept, the action itself, and the post-condition, which has to be satisfied after executing the concept (see [22] for details and examples of the chain Datalog rules, which have been learned with ILP algorithms). Plans can be specified as sequences of operational concepts, whose pre- and post-conditions may overlap. Based on the idea, that chain Datalog programs correspond to CFGs ( in our case, regular languages), we have succeeded in specifying an automaton, which accepts sequences of perception-integrating action features, which represent plans. Its final states are associated with operational concepts. The graph structure of this automaton can be used for a depth-bounded breadth-first search, as proposed by Klingspor. The depth bound can be realized by specifying the maximal number of final states, which can be visited during a plan. The point, we want to make here, is that this search can be implemented by a modified marker passing method, where the init and update functions for the constraints have to be specified for the respective data classes, i.e., operational concepts and perception-integrating features.

Future work will also address the integration of the probabilities, estimated with the method described in [20], in the logic programming framework. The goal is to modify the marker passing method for the probabilistic case, such that it constitutes an inference procedure for a probabilistic logic based on the semantics given by Ng and Subrahmanian (see, e.g., [17]).

**Acknowledgements** The author would like to thank A. Hallmann and St. Weber for discussions and constructive comments, which helped to clarify the ideas and to improve this paper. Thanks to St. Wessel for providing the algorithm for calculating basic features. St. Sklorz developed the perception-integrating action features and applied RDT to learn the (chain Datalog) rules for operational concepts. K. Morik helped with comments for comparing related approaches.

# A Appendix

## A.1 Algorithm `prefix_tree`

`prefix_tree`($Cases$)
**begin**

$Edges := \emptyset$;

$Vertices := \{RootNode\}$ ;

**while** $Cases \neq \emptyset$

1. select $[C_{head}, L_1, \ldots, L_n] \in Cases$;
2. $Cases := Cases - \{[C_{head}, L_1, \ldots, L_n]\}$;
3. $CurrentNode := RootNode$;
4. **for** $i = 1, \ldots, n$

   **if** $(CurrentNode, L, Next) \in Edges$ such that $L$ is unifiable with $L_i$ **then**
   **begin**
   (a) $CurrentNode :=$ `update`$(CurrentNode, i, [C_{head}, L_1, \ldots, L_n])$;
   (b) $CurrentNode := Next$;
   (c) i:= i+1;
   **end**

   **else begin**
   (a) $NewNode :=$`new_node`;
   (b) $NewNode :=$`update_node`$(NewNode, i, [C_{head}, L_1, \ldots, L_n])$;
   (c) $Vertices := Vertices \cup \{NewNode\}$;
   (d) $Edges := Edges \cup \{(CurrentNode, L_i, NewNode)\}$;
   (e) $CurrentNode := NewNode$;
   (f) $i := i + 1$;
   **end**
5. $Q := Vertices$;
6. $\Sigma := \{L|(q_i, L, q_j) \in Edges\}$;
7. $Z := \{C|\exists[C_{head}, L_1, \ldots, L_n] \in Cases$ and $C$ is a variant of $C_{head}\}$;
8. $\Delta := Edges$;
9. $q_0 := RootNode$;
10. $F = \{q|q \in Vertices$ and $\#CC(q) > 0\}$;
11. for all $q \in Q$:
    $\lambda(q) = \{C|C$ is a variant of some $C_{head}$ with $[C_{head}, L_1, \ldots, L_n] \in CC(q)\}$;

**return** $(Q, \Sigma, Z, \Delta, q_0, F, \lambda)$;
**end**

Algorithm 12: `prefix_tree`

## A.2 Algorithm `restruct_dc`

`restruct_dc`($\mathbf{P}_{\mathrm{I}}$)
**begin**

1. `restruct_init`($\mathbf{P}_{\mathrm{I}}, ToDo, Done$);
2. `restruct2_dc`($ToDo, Done1$);
3. **return** $\mathbf{P}'_{\mathrm{I}} := Done \cup Done1$;

**end**

Algorithm 13: `restruct_dc`

`restruct2_dc`($Rules, Done$)
**begin**

1. $Done := \emptyset$;
2. $ToDo := Rules$;
3. **while** there exists $C \in ToDo$ such that $C = B \leftarrow A$ or $C = B \leftarrow A_1 A_2 A_3 \ldots A_n$
   **if** $C = B \leftarrow A$ **then**
   (a) $Done := Done \cup \{C\}$;
   (b) $ToDo := ToDo - \{C\}$;

   **else**
   (a) `det_data_class`($(A_1, A_2), ALevel$);
   (b) $Constraints :=$`det_constraints`($(A_1, A_2), ALevel$);
   (c) $q :=$`new_predicate_symbol`;
   (d) $Head :=$ `new_atom`($q, ALevel, Constraints$);
   (e) $Done := Done \cup \{Head \leftarrow A_1, A_2\}$;
   (f) $ToDo :=$ `fold`($ToDo, Head \leftarrow A_1, A_2$);

**end**

Algorithm 14: `restruct2_dc`

## A.3 Auxiliary functions

`update_node`$(Node, i, [C_{head}, L_1, \ldots, L_n])$
**begin**

**if** $i < n$ **then**
  $\#SC(Node) := \#SC(Node) + 1;\ SC(Node) := SC(Node) \cup \{[C_{head}, L_1, \ldots, L_n]\};$

**else** $\#CC(Node) := \#CC(Node) + 1;\ CC(Node) := CC(Node) \cup \{[C_{head}, L_1, \ldots, L_n]\};$

**return** Node;

**end**

Algorithm 15: Auxiliary functions for `prefix_tree`

`det_constraints`$((A_1, \ldots, A_n), ALevel)$
**begin**

**switch** $ALevel$ =**BF then**

  **case BF**:
    **return** $\{\mathtt{tr}_{\mathbf{BF}} = tr_{\mathbf{BF}}(A_1), \mathtt{o}_{\mathbf{BF}} = o_{\mathbf{BF}}(A_1), \mathtt{s}_{\mathbf{BF}} = s_{\mathbf{BF}}(A_1), \mathtt{to}_{\mathbf{BF}} = to_{\mathbf{BF}}(A_n)\}$

  **case SF**:
    **return** $\{\ldots\}$;

  **case** ...

**end**

`new_atom`$(ALevel, q, \kappa_{\mathrm{ALevel}})$
**begin**

1. $A$:= generate an atom of data class $ALevel$ with predicate symbol $q$;
2. add $q$ to data class $ALevel$;
3. **return** $A\kappa_{\mathrm{ALevel}}$;

**end**

Algorithm 16: Auxiliary functions for `restruct_dc`

### A.4 Marker Passing: Example run on $PA'$

The prefix acceptor $PA'$ is illustrated in Figure 7, the EDB instance is the chain $\mathbf{P}_{\mathrm{E}} = \breve{A}^1\breve{A}^2\breve{A}^3\breve{A}^4$

$$\mathbf{P}_{\mathrm{E}} = \{a(t1, 90, s5, 1, 8), b(t1, 90, s5, 8, 10), c(t1, 90, s5, 10, 15), d(t1, 90, s5, 15, 17)\}.$$

**Input:** $\breve{A}^1 = a(t1, 90, s5, 1, 8)$:

$$\begin{aligned} m_1^1 = (1, \quad q_{b^*ab^*}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = 8\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = 8\}) \end{aligned}$$

**Input:** $\breve{A}^2 = b(t1, 90, s5, 8, 10)$:

$$\begin{aligned} m_1^2 = (1, \quad q_{b^*ab^*}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{10}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{10}\}) \\ m_2^2 = (2, \quad q_{b^*}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = 10\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = 10\}) \end{aligned}$$

**Input:** $\breve{A}^3 = c(t1, 90, s5, 10, 15)$:

$$\begin{aligned} m_1^3 = (1, \quad q_{b^*ab^*c}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{15}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{15}\}) \\ m_2^3 = (2, \quad q_{b^*c}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{15}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{15}\}) \end{aligned}$$

**Output:** $\mathbf{MP}(\breve{A}^1\breve{A}^2\breve{A}^3) = \{p_1(t1, s5, 1, 15), p_2(t1, s5, 1, 15)\}$

**Input:** $\breve{A}^4 = d(t1, 90, s5, 15, 17)$:

$$\begin{aligned} m_1^4 = (1, \quad q_{b^*ab^*cd}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{17}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 1\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{17}\}) \\ m_2^4 = (2, \quad q_{b^*cd}, \quad & \{\mathtt{tr}_{\mathbf{BF}} = t1, \mathtt{o}_{\mathbf{BF}} = 90, \mathtt{s}_{\mathbf{BF}} = s5\}, \{\mathtt{from}_{\mathbf{BF}} = \mathbf{17}\}, \\ & \{\mathtt{tr}_{\mathbf{SF}} = t1, \mathtt{s}_{\mathbf{SF}} = s5, \mathtt{from}_{\mathbf{SF}} = 8\}, \{\mathtt{to}_{\mathbf{SF}} = \mathbf{17}\}) \end{aligned}$$

**Output:** $\mathbf{MP}(\breve{A}^1\breve{A}^2\breve{A}^3\breve{A}^4) = \{p_3(t1, s5, 1, 17), p_4(t1, s5, 8, 17)\}$

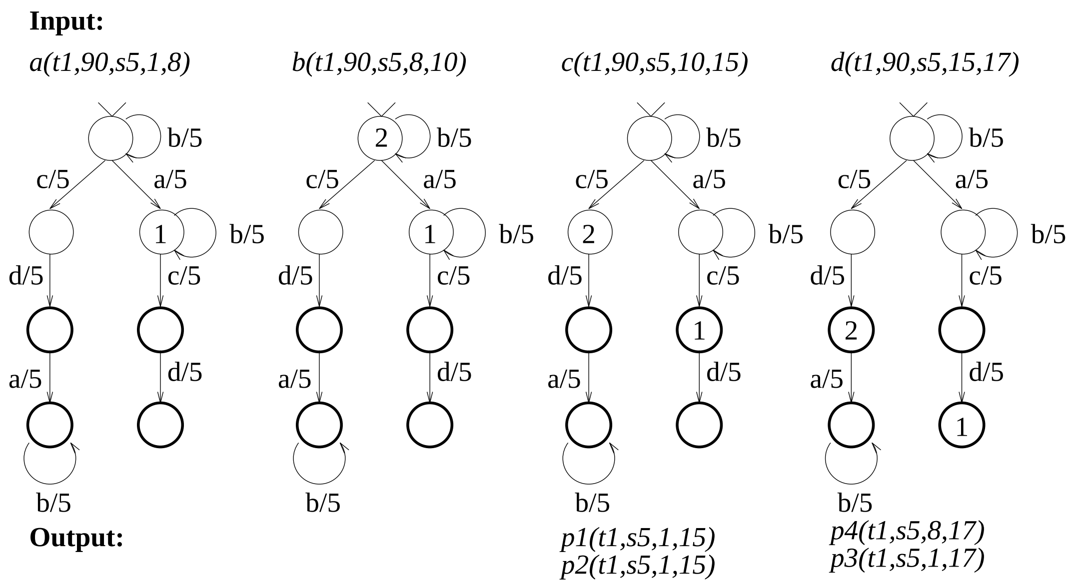


Figure 20: Example 2

## A.5 Post-Processing: Experimental Results

### A.5.1 Complexity Results

| | | No Post-processing | | | | Post-processing: Step 1 | | | | | | Step 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Train | \|Train\| | $\|Q\|$ | $\|F\|$ | $\|\Delta\|$ | Depth | $\|Q\|$ | $Q_{Red}$ | $\|F\|$ | $\|\Delta\|$ | $\Delta_{Red}$ | Depth | $\|\Delta\|$ |
| QRS | 1185 | 408 | 268 | 407 | 9 | 141 | 65.4% | 116 | 193 | 52.6% | 6 | 422 |
| PRS | 1183 | 457 | 298 | 456 | 9 | 157 | 65.6% | 130 | 219 | 52.0% | 6 | 470 |
| PQS | 1220 | 428 | 279 | 427 | 8 | 145 | 66.1% | 119 | 208 | 51.3% | 6 | 434 |
| PQR | 1272 | 459 | 305 | 458 | 9 | 154 | 66.4% | 128 | 222 | 51.5% | 6 | 461 |
| | 1215 | 438 | 288 | 437 | 9 | 149 | **65.9%** | 123 | 211 | **51.9%** | 6 | 447 |

Table 3: Prefix acceptors for basic features calculated with $Tolerance = 6$

| | | No Post-processing | | | | Post-processing: Step 1 | | | | | | Step 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Train | \|Train\| | $\|Q\|$ | $\|F\|$ | $\|\Delta\|$ | Depth | $\|Q\|$ | $Q_{Red}$ | $\|F\|$ | $\|\Delta\|$ | $\Delta_{Red}$ | Depth | $\|\Delta\|$ |
| QRS | 1163 | 307 | 228 | 306 | 7 | 123 | 59.9% | 106 | 161 | 47.4% | 5 | 368 |
| PRS | 1167 | 343 | 245 | 342 | 7 | 135 | 60.6% | 110 | 175 | 48.8% | 5 | 404 |
| PQS | 1201 | 312 | 230 | 311 | 7 | 125 | 59.9% | 102 | 166 | 46.6% | 5 | 374 |
| PQR | 1254 | 328 | 245 | 327 | 7 | 134 | 59.1% | 110 | 178 | 45.6% | 5 | 401 |
| | 1196 | 323 | 237 | 322 | 7 | 129 | **59.9%** | 107 | 170 | **47.1%** | 5 | 387 |

Table 4: Prefix acceptors for basic features calculated with $Tolerance = 8$

| | | No Post-processing | | | | Post-processing: Step 1 | | | | | | Step 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Train | \|Train\| | $\|Q\|$ | $\|F\|$ | $\|\Delta\|$ | Depth | $\|Q\|$ | $Q_{Red}$ | $\|F\|$ | $\|\Delta\|$ | $\Delta_{Red}$ | Depth | $\|\Delta\|$ |
| QRS | 1143 | 274 | 195 | 273 | 7 | 110 | 59.9% | 92 | 146 | 46.5% | 5 | 329 |
| PRS | 1146 | 304 | 211 | 303 | 7 | 125 | 58.9% | 101 | 162 | 46.5% | 5 | 374 |
| PQS | 1183 | 285 | 205 | 284 | 7 | 122 | 57.2% | 98 | 163 | 42.6% | 5 | 365 |
| PQR | 1232 | 283 | 206 | 282 | 7 | 121 | 57.2% | 98 | 163 | 42.2% | 5 | 362 |
| | 1176 | 287 | 204 | 286 | 7 | 120 | **58.3%** | 97 | 159 | **44.5%** | 5 | 358 |

Table 5: Prefix acceptors for basic features calculated with $Tolerance = 10$

| | | No Post-processing | | | | Post-processing: Step 1 | | | | | | Step 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Train | \|Train\| | $\|Q\|$ | $\|F\|$ | $\|\Delta\|$ | Depth | $\|Q\|$ | $Q_{Red}$ | $\|F\|$ | $\|\Delta\|$ | $\Delta_{Red}$ | Depth | $\|\Delta\|$ |
| QRS | 1093 | 213 | 158 | 212 | 7 | 93 | 56.3% | 80 | 125 | 41.0% | 5 | 278 |
| PRS | 1091 | 239 | 171 | 238 | 7 | 101 | 57.7% | 83 | 135 | 43.3% | 5 | 302 |
| PQS | 1121 | 232 | 172 | 231 | 7 | 103 | 55.6% | 83 | 137 | 40.7% | 5 | 308 |
| PQR | 1177 | 217 | 165 | 216 | 6 | 101 | 53.5% | 83 | 136 | 37.0% | 5 | 302 |
| | 1121 | 225 | 167 | 224 | 7 | 100 | **55.8%** | 82 | 133 | **40.5%** | 5 | 298 |

Table 6: Prefix acceptors for basic features calculated with $Tolerance = 15$

### A.5.2 Testing Results

| | | | | No PP | PP: Step 1 | | PP: Step 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Train | Test | \|Train\| | \|Test\| | $C_0$ | $C_1$ | $I_{0,1}$ | $C_2$ | $I_{0,2}$ | $I_{1,2}$ |
| QRS | P | 1185 | 435 | 63.2% | 69.9% | 6.7% | 72.6% | 9.4% | 2.7% |
| PRS | Q | 1183 | 437 | 67.5% | 74.8% | 7.3% | 75.3% | 7.8% | 0.5% |
| PQS | R | 1220 | 400 | 52.3% | 60.5% | 8.2% | 61.0% | 8.7% | 0.5% |
| PQR | S | 1272 | 348 | 56.6% | 65.8% | 9.2% | 65.8% | 9.2% | 0.0% |
| | | 1215 | 405 | 59.9% | 67.8% | **7.9%** | 68.7% | **8.8%** | **0.9%** |

Table 7: Testing results for basic features calculated with $Tolerance = 6$

| | | | | No PP | PP: Step 1 | | PP: Step 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Train | Test | \|Train\| | \|Test\| | $C_0$ | $C_1$ | $I_{0,1}$ | $C_2$ | $I_{0,2}$ | $I_{1,2}$ |
| QRS | P | 1163 | 432 | 66.2% | 72.7% | 6.5% | 75.5% | 9.3% | 2.8% |
| PRS | Q | 1167 | 428 | 70.1% | 74.8% | 4.7% | 75.2% | 5.1% | 0.4% |
| PQS | R | 1201 | 394 | 52.8% | 59.4% | 6.6% | 59.6% | 6.8% | 0.2% |
| PQR | S | 1254 | 341 | 56.3% | 63.3% | 7.0% | 63.3% | 7.0% | 0.0% |
| | | 1196 | 399 | 61.4% | 67.6% | **6.2%** | 68.4% | **7.1%** | **0.9%** |

Table 8: Testing results for basic features calculated with $Tolerance = 8$

| | | | | No PP | PP: Step 1 | | PP: Step 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Train | Test | \|Train\| | \|Test\| | $C_0$ | $C_1$ | $I_{0,1}$ | $C_2$ | $I_{0,2}$ | $I_{1,2}$ |
| QRS | P | 1143 | 425 | 65.7% | 70.6% | 4.9% | 72.9% | 7.2% | 2.3% |
| PRS | Q | 1146 | 422 | 69.2% | 73.0% | 3.8% | 73.5% | 4.3% | 0.5% |
| PQS | R | 1183 | 385 | 55.6% | 60.8% | 5.2% | 60.8% | 5.2% | 0.0% |
| PQR | S | 1232 | 336 | 54.8% | 60.4% | 5.6% | 60.4% | 5.6% | 0.0% |
| | | 1176 | 392 | 61.3% | 66.2% | **4.9%** | 66.9% | **5.6%** | **0.7%** |

Table 9: Testing results for basic features calculated with $Tolerance = 10$

| | | | | No PP | PP: Step 1 | | PP: Step 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Train | Test | \|Train\| | \|Test\| | $C_0$ | $C_1$ | $I_{0,1}$ | $C_2$ | $I_{0,2}$ | $I_{1,2}$ |
| QRS | P | 1093 | 401 | 67.1% | 71.6% | 4.5% | 73.6% | 6.5% | 2.0% |
| PRS | Q | 1091 | 403 | 70.2% | 73.0% | 2.8% | 73.0% | 2.8% | 0.0% |
| PQS | R | 1121 | 373 | 57.4% | 60.9% | 3.5% | 60.9% | 3.5% | 0.0% |
| PQR | S | 1177 | 317 | 54.9% | 59.3% | 4.4% | 59.3% | 4.4% | 0.0% |
| | | 1121 | 374 | 62.4% | 66.2% | **3.8%** | 68.2% | **4.3%** | **0.5%** |

Table 10: Testing results for basic features calculated with $Tolerance = 15$

## References

[1] D. Angluin. Inference of reversible languages. *Journal of the Association for Computing Machinery*, 29:741–765, 1982.

[2] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery*, 29:841–862, 1982.

[3] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[4] Ch.-L. Chang and R. Ch. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[5] E. Charniak. Passing markers: A theory of contextual influence in language comprehension. *Cognitive Science*, 7, 1983.

[6] G. Dong and S. Ginsburg. On the decomposition of datalog program mappings. *Theoretical Computer Science*, 75:143–177, 1990.

[7] G. Dong and S. Ginsburg. On decompositions of chain Datalog programs into $p$ (left-)linear 1-rule components. *Journal of Logic Programming*, 23:203–236, 1995.

[8] J. A. Hendler. Integrating marker-passing and problem solving. In A. Tate J. Allen, J. Hendler, editor, *Readings in Planning*, pages 275–287. Morgan Kaufmann, 1990.

[9] J. D. Ullman J. E. Hopcroft. *Introduction to Automata Theory, Languages, an Computation*. Addison-Wesley, 1979.

[10] V. Klingspor, K. Morik, and A. Rieger. Learning concepts from sensor data of a mobile robot. *Machine Learning*, 1996. to appear.

[11] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2nd edition, 1987.

[12] K. Morik, St. Wrobel, J. U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications*. Addison Wesley, 1993.

[13] S. Muggleton. Inverse entailment and Progol. *New Generation Computing Journal*, 13:245–286, 1995.

[14] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming*, 1992.

[15] S. H. Muggleton. Duce, an oracle based approach to constructive induction. In *Proc. of the 10th Int. Joint Conf. on Artificial Intelligence*, Los Altos, 1987. Morgan Kaufmann.

[16] St. Muggleton and C. Feng. Efficient induction of logic programs. In St. Muggleton, editor, *Inductive Logic Programming*, chapter 13, pages 281–298. Academic Press, 1992.

[17] V.S. Subrahmanian R. Ng. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *Journal of Automated Reasoning*, 10:191–235, 1993.

[18] A. S. Rao. Means-end plan recognition - towards a theory of reactive recognition. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 497–508, 1994.

[19] A. Rieger. Inferring probabilistic automata from sensor data for robot navigation. In M. Kaiser, editor, *Proc. of the 3rd European Workhop on Learning Robots*, 1995. also available as Research Report 18, FB Informatik LS 8, Universität Dortmund, Dortmund, Germany.

[20] A. Rieger. Learning to guide a robot via perceptions. In M. Ghallab, editor, *Procs. of the 3rd European Workshop on Planning.* IOS Press, 1996.

[21] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In *Inductive Logic Programming*, pages 63–92. Academic Press, 1992.

[22] St. Sklorz. Representing and learning operational concepts. Master's thesis, Universität Dortmund, 1995. in German.

[23] E. Sommer. FENDER : An approach to theory restructuring. In N. Lavrač and St. Wrobel, editors, *Proc. of the European Conference on Machine Learning (ECML-95)*, pages 356–359. Springer Verlag, 1995.

[24] J. D. Ullman and A. van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[25] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.

[26] St. Wessel. Learning qualitative features from numerical robot sensor data. Master's thesis, Universität Dortmund, 1995. in German.

[27] R. Wirth. Completing logic programs by inverse resolution. In K. Morik, editor, *Proc. Fourth European Workong Session on Learning (EWSL)*, pages 239–250. Morgan Kaufmann, 1989.

[28] S. Wrobel. *Concept Formation and Knowledge Revision.* Kluwer Academic Publishers, 1994.